MIDRANGE
HANDBÜCHER

7 Kapitel SQL und Datenbank

Walter Becker, Ronald Böhmer, Birgitta Hauser, Klaus-Peter Luttkus,
Carmen Rasche, Rainer Ross, Jörg Zeig

IBM POWER Systems i Softwareentwicklung und -modernisierung

Handbuch für Strategien und Leitfaden für moderne Programmierung
sowie Tipps & Tricks

ITP VERLAG

SQL und Datenbank

7.1 System i DB2 Universal Database (UDB)
7.1.1 Datenbank-Limits DB2 UDB Release V5R4

7.2 Das Datenbank-Design
7.2.1 Situationsbeschreibung „Auftragsverwaltung“
7.2.2 Die Systemanalyse
7.2.2.1 Die Analyse-Methode „Structured Analysis“
7.2.2.1.1 Datenfluss-Analyse der „Auftragsverwaltung“
7.2.2.1.2 Data Dictionary „Auftragsverwaltung“
7.2.3 Die Datenbank-Beschreibung
7.2.3.1 Begriffsdefinition
7.2.3.2 Der Datenbank-Entwurf
7.2.3.2.1 Die Normalisierung der Datenbank
7.2.3.2.2 Die Relationenbeziehungen
7.2.3.2.3 ANSI-SPARC-3 Schichtenmodell
7.2.3.2.4 Schlüsseltabellen
7.2.3.2.5 Rollenmodellierung
7.2.3.2.6 Spezifische Attribute
7.2.4 Das physische Datenbankkonzept
7.2.4.1 Datenbankintegrität
7.2.4.1.1 Entitätsintegrität
7.2.4.1.2 Domänenintegrität
7.2.4.1.3 Referentielle Datenintegrität
7.2.4.1.3.1 Definition einer referentiellen Integritätsbedingung
7.2.4.1.3.2 Überprüfung referentieller Integritätsbedingungen
7.2.4.1.3.3 Allgemeine Überlegungen zur Implementierung
7.2.4.1.3.4 Überlegungen zur Performance
7.2.4.2 Datenbankrelationen analysieren
7.2.4.2.1 Der Befehl „Display database relation (DSPDBR)“
7.2.4.2.2 Database Navigator-Masken einsetzen

7.3 Data Definition Language (DDL) Seite 3
7.3.1 Relationale iSeries-Datenbanken
7.3.2 Schemata
7.3.2.1 Systemschemata
7.3.2.2 Benutzerschemata erstellen
7.3.3 SQL-Tabellen
7.3.3.1 Der Befehl CREATE TABLE
7.3.3.2 Tabellen- und Spaltennamen
7.3.3.3 SQL-Datentypen
7.3.3.3.1 Zeichen speichern
7.3.3.3.2 Temporale Daten speichern
7.3.3.3.3 Ziffern speichern
7.3.3.3.4 DATALINK-Spalten
7.3.3.3.5 Benutzerdefinierte Datentypen
7.3.3.3.6 Identity-Spalten
7.3.3.3.6.1 ROWID-Spalten
7.3.3.3.6.2 Die Klausel AS IDENTITY
7.3.3.3.6.3 SEQUENZ-Objekte
7.3.3.4 ALTER TABLE – Änderung Ihrer Tabellen
7.3.3.5 COMMENT ON – Dokumentation Ihrer Datenbank
7.3.3.6 CREATE VIEW – logische SQL-Sichten erstellen
7.3.3.7 CREATE Alias – Alias-Namen erstellen
7.3.3.8 Der DB2 UDB for iSeries Datenbankkatalog
7.3.4 Integritätsbedingungen
7.3.4.1 UNIQUE Constraints
7.3.4.2 Primärschlüssel-Constraints
7.3.4.3 Fremdschlüssel
7.3.4.4 Check Constraints
7.3.5 SQL-INDIZES
7.3.5.1 Der Binary Radix Tree Index
7.3.5.2 Binärindexerstellung
7.3.5.3 Der Encoded Vector Index (EVI)
7.3.5.4 EVI-Index – Erstellung und Pflege
7.3.5.5 Maintained Temporary Index (MTI)

7.3.5.6 Abgeleitete Indizes Release V6R1
7.3.6 SQL-INDIZES – Einsatz und Wirkung
7.3.6.1 Das Optimierungsziel
7.3.6.2 Single Table Optimierung
7.3.6.3 Encoded Vector Index (EVI)
7.3.6.4 Index Access Only
7.3.6.5 Tabellenverknüpfungen und Indizes
7.3.6.6 Gruppierungen und Indizes
7.3.6.7 Sortierungen und Indizes
7.3.6.8 Warum werden meine Indizes ignoriert?
7.3.6.9 Überprüfung der Indexstrategie
7.3.7 Materialized Query Tables – Einsatz und Wirkung
7.3.7.1 MQTs erstellen
7.3.7.2 Systemvorbereitungen für den MQT-Einsatz
7.3.7.3 Lohnt sich der Einsatz von MQTs?
7.3.7.4 Bestehende Tabellen als MQTs nutzen
7.3.8 Der OmniFind Text Search Server für DB2
7.3.8.1 Installation
7.3.8.2 OmniFind Starten und Stoppen
7.3.8.3 Der OmniFind-Textsuchindex
7.3.8.4 Unterstützte Datentypen des Textsuchindexes
7.3.8.5 OmniFind-Verwaltungs-Tools
7.3.8.6 OmniFind-Suchoperatoren
7.3.8.7 Suche mit Synonymen
7.3.8.8 Daten aus dem IFS für die OmniFind-Suche einbinden
7.3.8.9 Textsuchgruppen
7.3.8.10 Textsuchgruppe für Source Physical Files erstellen
7.3.8.11 Textsuchgruppe für Spools erstellen
7.3.8.12 Textsuchgruppe für IFS-Dateien erstellen
7.3.9 SQL-Objekte und i5/OS-Berechtigungen
7.3.9.1 i5/OS-Berechtigungen für SQL-Schemata
7.3.9.2 i5/OS-Berechtigungen für SQL-Tabellen
7.3.9.5 Zeilen und Spaltenberechtigungen (RCAC)
7.3.9.5.1 Zeilen-/Spaltenberechtigungen (RCAC) – Voraussetzungen

7.3.9.5.2 Zeilenberechtigung erteilen
7.3.9.5.3 Zeilenberechtigungen anzeigen
7.3.9.5.4 Spaltenberechtigungen erstellen
7.3.9.5.5 Spaltenberechtigungen anzeigen
7.3.9.5.6 RCAC-Bedingungen und die SQL-Indexierung
7.3.9.5.7 RCAC und Materialized Query Tables (MQT)
7.3.9.5.8 Datenbankadministration und RCAC
7.3.9.5.9 RCAC in der Programmierung
7.3.9.5.10 Sonstige Überlegungen
7.3.10 Partitionierte Tabellen
7.3.10.1 Wofür werden partitionierte Tabellen benötigt?
7.3.10.2 Wie werden partitionierte Tabellen erstellt?
7.3.10.3 Der Migrationsprozess
7.3.10.4 Indizes und partitionierte Tabellen
7.3.10.5 Partitionierte Tabellen und der SQL-Optimizer

7.4 Grafische Datenbank-Tools
7.4.1 iSeries Navigator
7.4.1.1 Mit Datenbanken arbeiten
7.4.1.2 Mit Datenbanktabellen arbeiten
7.4.1.2.1 Datenbanktabellen über das grafische Interface erstellen
7.4.1.2.2 SQL Prozedurumgebung iSeries Navigator
7.4.1.2.2.1 SQL-Prozedurumgebung aufrufen
7.4.1.2.2.2 JDBC-Schnittstelle konfigurieren
7.4.1.2.2.3 SQL-Statement in der Skript-Umgebung ausführen
7.4.1.3 Mit Integritätsbedingungen arbeiten
7.4.1.4 Mit Statistikdaten arbeiten
7.4.2 Visual Explain
7.4.2.1 Toolbar und Menüoptionen
7.4.2.2 Visual Explain Abfrageattribute und Werte
7.4.3 SQL Performance Monitor
7.4.3.1 Detailed SQL Performance Monitor starten
7.4.3.2 Leistungsanalyse Detailmonitor
7.4.3.3 Summary Performance-Monitor starten
7.4.4 Der Plan Cache

7.4.4.1 Snapshot des SQL Plan Cache
7.4.5 Die SQL Workbench/J – Eine Alternative zum System i Navigator?
7.4.5.1 Download und Installation der SQL Workbench/J
7.4.5.2 Konfiguration des JDBC-Treibers
7.4.5.3 Konfiguration der Verbindungsprofile
7.4.5.4 SQL Workbench/J – Ein Überblick der Skriptumgebung
7.4.5.4.1 SQL Skriptumgebung – Variablen deklarieren
7.4.5.4.2 SQL Skriptumgebung – Arbeiten mit BLOB-Daten
7.4.5.4.3 SQL Skriptumgebung – Daten im Result Set verändern
7.4.5.4.4 SQL Skriptumgebung – Komfortable Skript-Funktionen
7.4.5.5 Die SQL Workbench/J – Datenbank-Explorer
7.4.6 Eclipse Datenbank-Tools
7.4.6.1 Erstellen eines Verbindungsprofils
7.4.6.2 Arbeiten mit SQL
7.4.6.2.1 Erstellen eines Filters
7.4.6.3 SQL-Erstellungsprogramm
7.4.7 RDI und die SQL-Objekterstellung
7.4.7.1 Kompilierbefehle für Quellenteildateien erstellen
7.4.7.2 Benutzeraktionen für IFS-Teildateien erstellen

7.5 Interfaces für die SQL-Ausführung

7.5.1 Der Befehl RUNSQLSTM
7.5.2 Der Befehl RUNSQL
7.5.5 Der Query Manager
7.5.5.1 Query Manager – Benutzerprofile
7.5.5.2 Querymanager – QM-Abfragen erstellen (menügeführt)
7.5.5.3 Querymanager – SQL-Abfragen erstellen und ausführen
7.5.5.4 Querymanager Formulare (*QMFORM) – einfache Berichte
7.5.5.5 Query Manager–Formulare (*QMFORM) – Gruppenwechsel
7.5.5.6 Query Manager-Abfragen parametrisieren
7.5.5.7 Query/400 (*QMQRY) konvertieren
7.5.5.8 Query Manager – Mit Tabellen arbeiten

7.6 SQL-Data Manipulation Language (DML) Seite 7
7.6.1 Abfragen mit SELECT
7.6.1.1 Spaltenauswahl
7.6.1.2 Daten selektieren
7.6.1.2.1 Reguläre Ausdrücke verwenden
7.6.1.3 SQL-Spaltenfunktionen
7.6.1.3.1 Aggregatfunktionen
7.6.1.3.2 Skalare Funktionen
7.6.1.4 Daten gruppieren
7.6.1.4.1 Daten gruppieren – HAVING
7.6.1.5 Daten sortieren
7.6.2 Wie wird ein SQL-Statement eigentlich verarbeitet?
7.6.3 Tabellenverknüpfungen (JOIN)
7.6.3.1 INNER JOIN
7.6.3.1.1 INNER JOIN mit mehr als zwei Tabellen
7.6.3.1.2 Die USING-Klausel V5R3
7.6.3.1.3 Kreuzprodukt – Cross Join
7.6.3.2 OUTER JOIN
7.6.3.3 EXCEPTION-JOIN
7.6.3.4 FULL OUTER JOIN
7.6.3.5 Self JOIN
7.6.3.6 Non-Equi-Joins
7.6.4 Vertikale Verknüpfungen – der Operator UNION
7.6.5 Der Einsatz von Unterabfragen
7.6.5.1 Skalare Unterabfrage – eine zurückgegebene Zelle
7.6.5.2 Temporäre Spalten und temporäre Tabellen erstellen
7.6.5.3 Common Table Expressions (CTE)
7.6.5.4 Korrelierende Unterabfragen
7.6.5.5 Die Prädikate EXISTS, ALL, ANY oder SOME
7.6.6 SQL-Funktionen im praktischen Einsatz
7.6.6.1 Mit Datum und Uhrzeit arbeiten
7.6.6.2 DAYOFWEEK, DAYNAME und Fallunterscheidungen
7.6.6.2.1 CASE-Funktion und Pivot-Tabellen
7.6.6.3 Jahreszahl, Monatsname, Kalenderwoche ...
7.6.6.4 Auswertungen erstellen: Datumsdifferenzen und CTE

7.6.6.5 Die Funktionen LAST_DAY, NEXT_DAY und ADD_ MONTHS
7.6.6.6 Logische Funktionen, Konvertierungs- und Zeichenkettenfunktionen
7.6.6.7 Datenverschlüsselung
7.6.6.8 ROW_NUMBER(), RANK() und DENSE_RANK() – Release V5R4
7.6.6.9 Mehrdimensionale Gruppierungen – Release i6.1
7.6.6.10 Skalare Funktionen mit regulären Ausdrücken
7.6.8 Rekursive SQL-Aufrufe
7.6.8.1 Wechselseitige Rekursionen
7.6.8.2 Rekursive SQL-Aufrufe – CONNECT BY
7.6.8.3 Optimierung rekursiver SQL-Aufrufe
7.6.10 Datensätze einfügen – Die Anweisung INSERT
7.6.10.1 INSERT INTO – Spaltenwerte
7.6.11 Datensätze aktualisieren – Die Anweisung UPDATE
7.6.11.1 Bedingte Datenaktualisierungen
7.6.12 Datensätze löschen – Die Anweisung DELETE

7.7 Prozeduren
7.7.1 Prozeduren mit dem System i Navigator erstellen
7.7.2 Die Anweisung CREATE PROCEDURE
7.7.3 Objektadressierung in Prozeduren
7.7.4 Prozeduren im Datenbankkatalog verschlüsseln
7.7.5 Der Prozeduraufruf
7.7.6 Overloading von Prozeduren
7.7.7 SQL Control Language

7.8 Funktionen Seite 9
7.8.1 Skalare SQL-Funktionen
7.8.1.1 Skalare Funktionen mit dem System i Navigator definieren
7.8.1.2 Skalare SQL-Funktionen in der Prozedurumgebung erstellen
7.8.2 Tabellenfunktionen (UDTF)
7.8.3 Pipelined Table Function
7.8.4 Abgeleitete Funktionen

7.9 Datenbanktrigger
7.9.1 Datenbanktrigger – Eigenschaften
7.9.2 Datenbanktrigger erstellen – grafisches Interface
7.9.3 Datenbanktrigger erstellen – CREATE TRIGGER
7.9.4 Triggeradministration
7.9.5 Multi-Event-Trigger (i7.1 TR6)
7.9.6 Instead-of-Trigger

7.10 Fehlerhandling in Prozeduren / Funktionen / Trigger
7.10.1 Fehlermeldungen durch die Datenbank
7.10.2 Die Anweisung GET DIAGNOSTICS
7.10.3 Globale und lokale Fehlerroutinen
7.10.4 Benutzerdefinierte Meldungen
7.10.4.1 Benutzerdefinierte Meldungen auslösen
7.10.6 Transaktionssicherung im SQL-Umfeld
7.10.6.1 Transaktionen starten und beenden
7.10.6.2 Die verschiedenen Isolationsstufen
7.10.6.3 Transaktionen identifizieren

7

Seite 10

SQL ist für die Datenbank DB2 UDB for iSeries der IBM zukunftsweisend und hat sich in den vergangenen Jahren als universelle Datenbanksprache durchgesetzt. Obwohl der Großteil aller Datenbanken auf der i5 bzw. iSeries mit DDS erzeugt wurde und wahrscheinlich die Mehrzahl der iSeries-Entwickler DDS gegenüber SQL bevorzugt, wird DDS – ähnlich wie der RPG-Programmzyklus – langsam nicht mehr praktiziert. Neue Datenbank-Funktionalitäten sind bereits jetzt nur noch SQL-basierend; eine entsprechende DDS-Syntax existiert nicht. Identity Columns, kodierte Vektorindizes und Constraints sind nur einige der neuen Features, die nicht DDS-, sondern SQL-gestützt sind. IBM hat gerade in den letzten Releases die integrierte Datenbank von Grund auf umgekrempelt und wichtige Erweiterungen implementiert. Der Schwerpunkt liegt dabei immer wieder auf SQL – mit dem Erfolg, dass die DB2 UDB for iSeries die erste Datenbank ist, die den SQL-2003-Standard 100-prozentig erfüllt. Zukünftig wird es immer mehr Neuerungen dieser Art geben, die nie in DDS aufgenommen werden. Dadurch, dass sich IBM auf eine SQL-basierte Datenbank-Definition festlegt, können die Kosten für die Verbesserung der Datenbank gesenkt werden. Das liegt daran, weil einerseits die Kosten auf alle IBM-Plattformen verteilt werden und andererseits die iSeries für Entwickler, die mit anderen Plattformen vertraut sind, attraktiver wird. Zudem kann die iSeries leichter in andere Systeme, die bereits ein SQL-Interface haben, integriert werden. Für IBM macht es einfach keinen Sinn Datenbank-Updates sowohl für SQL als auch für DDS zur Verfügung zu stellen.

Die Arbeit mit der relationalen Datenbank UDB/400 erfordert daher heute zweierlei:

- Kenntnisse der Datenbanksprache SQL und
- Grundlegende Kenntnisse des Planungs- und Entwurfsprozesses einer relationalen Datenbank.

In den folgenden Veröffentlichungen soll ein praktisches Fallbeispiel so aufbereitet werden, dass Sie über das Beispielhafte hinaus weitere Hinweise und Anregungen für Ihre tägliche Arbeit erhalten. Anhand der Fallstudie wird zunächst der Entwurfsprozess für eine Datenbank systematisch abgearbeitet. Dieser Abschnitt soll nicht unnötig komplex gestaltet werden, sondern dient vielmehr der Erläuterung allgemeiner Grundbegriffe und zeigt einen möglichen Weg zum Aufbau eines konsistenten und redundanzfreien Datenbank-Modells. Als Fallstudie habe ich eine Auftragsbearbeitung gewählt. Das Beispiel ist realistisch genug, um die Probleme relationaler Datenbanken zu verdeutlichen. Nach dieser kurzen theoretischen Einführung geht es schwerpunktmäßig um die Realisierung der Datenbank gemäß der definierten Anforderungen mittels SQL (DDL). Hierbei werden auch die beiden grafischen Schnittstellen vorgestellt, denn nicht nur DDS sondern auch PDM soll durch die grafischen Umgebungen – iSeries Navigator und WDSC – abgelöst

werden. In weiteren Beiträgen werden praktische Beispiele für die Datenmanipulation, aber auch für die Datenkontrolle im Multi-User-Umfeld folgen. Und nicht zuletzt spielt auch die Performance eine entscheidende Rolle. Denn wie schnell Ihre SQL-Statements verarbeitet werden, sollten Sie nicht dem Zufall überlassen. Und schließlich gibt es die vielfältigen Einsatzmöglichkeiten: ob als embedded SQL in RPG- bzw. COBOL-Programmen, als SQL-Prozeduren, SQL-Funktionen, SQL-Trigger oder als Query-Abfragen. SQL kennt viele Einsatzmöglichkeiten! Überzeugen Sie sich von dem vielfältigen Nutzen der Datenbanksprache SQL. Und damit die Umstellung Ihrer DDS-Datenbank auf SQL auch reibungslos funktioniert, wollen wir eventuelle Probleme, mögliche Vorgehensweisen thematisieren, damit auch in Ihrem Haus DDS zukünftig der Vergangenheit angehört!

7.1 System i DB2 Universal Database (UDB)

Eigentlich hatte die Datenbank der iSeries bis Mitte 1994 niemals einen offiziellen Namen und der Ausdruck „AS/400 Database“ entwickelte sich somit auf natürliche Weise im Laufe der Zeit. Für diejenigen, die ständig mit der iSeries zu tun hatten, war dies zu keiner Zeit ein Problem. Nur wenn wir mit Kollegen anderer Systeme sprachen, ernteten wir häufig merkwürdige Blicke für die namenlose Datenbank und stifteten erhebliche Verwirrung, wenn wir erklärten, dass die Datenbank keinen Namen benötigt, weil sie in der Architektur des Servers integriert sei. Letzteres ist auch heute noch so: Die Datenbank wird standardmäßig mit dem Betriebssystem i5/OS ausgeliefert, bereits Mitte 1994 hatte IBM die „AS/400 Database“ zum offiziellen Teil der relationalen Datenbankprodukte der DB2-Familie erklärt. Heute ist die „DB2 UDB for iSeries“ eine der leistungsfähigsten Datenbanken am Markt. Sie unterstützt neben einer Vielzahl weiterer Funktionen Trigger, Stored Procedures und dynamische Bitmap Indizes, so dass die unterschiedlichsten Applikationen denkbar sind. Angefangen von traditionellen Host-basierenden Anwendungen über Client/Server-Lösungen bis hin zu intelligenten Business-Applikationen ist alles denkbar. Die Datenbank verfügt hierfür über zwei unabhängige Schnittstellen:

Die native Schnittstelle stammt noch vom Erbe des Systems/38. Sie enthält die Data Description Specifications (DDS), einige Command Language-Befehle, um Datenbankobjekte zu erzeugen und zu verwalten, sowie Abfrage-Tools (Query/400). Sie arbeitet allerdings nicht SQL-orientiert. Die native Schnittstelle wird seitens IBM nicht mehr weiterentwickelt. Neue Datenbankfunktionalitäten implementiert IBM heute nur noch SQL-basierend; eine entsprechende DDS-Syntax existiert nicht.

Die AS/400 erhielt bereits zur Markteinführung neben der nativen Schnittstelle eine unabhängige SQL-Schnittstelle. Heute ist SQL die sowohl von IBM als auch von der Industrie anerkannte relationale Datenbanksprache. Precompiler und eine Reihe anderer Tools ermöglichen es Ihnen, SQL-Statements in ihre HLL-Programme zu integrieren sowie Ad-hoc-Abfragen und Reports zu erstellen. Insbesondere in den letzten Releases wurden Datenbank und SQL-Schnittstelle erheblich modifiziert – mit dem Erfolg, dass die „DB2 UDB for iSeries“ die erste Datenbank ist, die die SQL-2003-Standards zu 100 Prozent erfüllt.

7.1 Seite 2

7.1.1 Datenbank-Limits DB2 UDB Release V5R4

Um ein Maximum an Portabilität und Flexibilität zu erreichen, wurden im Release V5R4 einige Datenbank-Limits erneut erweitert. Nachfolgend ein Überblick über die aktuellen Datenbank-Limits V5R4:

Beschreibung	Maximumwert
Anzahl Bytes je Zeile	32.766 (32.740 bei Verwendung variabler Spalten)
Anzahl Zeilen je Member	4 294 967 294 Zeilen
Anzahl Member je physischer Datei	32.767
Anzahl Bytes je Member	1 869 162 846 624 Bytes (ca. 1,7 TB)
Anzahl Spalten je Zeile	8.000 Spalten
Anzahl Spalten je Schlüsseldefinition	120 Felder
Anzahl logischer Dateien je physischer Datei	3686 logische Dateien
Anzahl Bytes je Zugriffspfad	1 099 511 627 776 Bytes (1 TB)
Anzahl verknüpfter Tabellen je SQL-Statement	1000
Verknüpfte Tabellen in Views/MQTs	256
Anzahl gleichzeitig geöffneter Tabellen je Zugriff	ca. 500.000
Anzahl Constraints je Tabelle	300
Anzahl Trigger je Tabelle	300
Anzahl rekursiver Update- und Insert-Trigger	200

7.1.1 Seite 2

7.2 Das Datenbank-Design

Wenn die Idee entsteht, für ein bestimmtes Aufgabengebiet IT-Unterstützung zu schaffen, wird ein komplexer Prozess in Gang gesetzt, an dessen Ende möglicherweise eine fertige IT-Lösung steht. Dieser Prozess besteht aus einer Reihe von Tätigkeiten und Zwischenresultaten. Welche und wie viele Phasen für die Entstehung der Lösung notwendig sind, lässt sich nicht allgemeingültig bestimmen, aber jede Entwicklungsphase schließt mit einem definierten und dokumentierten Resultat ab. Man nennt dieses Phasenmodell auch „Software-Life-Cycle“, womit ausgedrückt wird, dass eine Software-Applikation nie richtig fertig sein wird, denn das fertige Endprodukt lässt neue Bedürfnisse nach Erweiterungen oder Verbesserungen entstehen. Der Benutzer entdeckt immer neue Anforderungen, so dass der Entwicklungsprozess häufig mit einer modifizierten, erweiterten Problemstellung von neuem beginnt.

In einem solchen Phasenmodell ging man früher stets von einem Anwendungsprogramm als Endprodukt aus. Ein Anwendungsprogramm, egal wie umfangreich dessen Funktionen auch sind, ist aber nur eine Ansammlung verschiedener Daten und deren Operationen. Wenn sich die Betrachtung auf eine einzelne Anwendung beschränkt, kann man die benötigten Datenstrukturen sicherlich relativ leicht bestimmen und gegebenenfalls auch optimieren. Ergeben sich aus der praktischen Arbeit heraus neue Anforderungen, müssen das Programm erweitert und die Daten dementsprechend angepasst werden, sofern dies überhaupt möglich ist. Wenn aber diese Daten in unterschiedlichen Kontexten genutzt werden, ist eine gemeinsame Verwaltung der zugrunde liegenden Datenbasis unumgänglich. Denn die anwendungsspezifische Sichtweise auf die Daten wird früher oder später ein inkonsistentes Verhalten des Informationssystems nach sich ziehen, da identische Informationen redundant verwaltet werden.

In einem relationalen Datenbankkonzept wird die Organisation und Speicherung der Daten daher unabhängig von jeder Anwendung realisiert. Die Datenbank bildet den Kern der betrieblichen Informationsverwaltung. In Verbindung mit einer Datendefinitions- und Datenmanipulationssprache wie SQL wird die Verwaltung des betrieblichen Datenbestands zu einer zentralen Aufgabe. Daneben gibt es interaktive Zugriffe auf die Daten. Ad hoc können so Daten nach unterschiedlichen Kriterien abgefragt und manipuliert werden, ohne dass Software entwickelt werden müsste. Folgerichtig müssen der Entwurf und die Realisierung der Datenbank eigenständig erfolgen – ohne Berücksichtigung bestehender oder zukünftiger Programme. Die Entwicklung einer relationalen Datenbankstruktur wird heute ähnlich der Software-Entwicklung in verschiedene Phasen unterteilt. Allen voran steht die Analyse, deren Ergebnis die Anforderungsdefinition ist. Aufgrund der Anforderungsdefinition entsteht die Beschreibung der Datenbank. Diese Phase nennt man Datenbankentwurf. Erst zum Schluss wird die Datenbank realisiert. Ergebnis der Realisierungsphase ist die leere Datenbank.

7.2
Seite 2

7.2.1 Situationsbeschreibung „Auftragsverwaltung“

In meinen Beiträgen verwende ich zum besseren Verständnis durchgängig nur eine Fallstudie und entwickle gemeinsam mit Ihnen eine Auftragsverwaltung. Ich reduziere den Bereich der Auftragsverwaltung bewusst auf ein für unsere Zwecke ausreichendes Maß. Dabei sollen die Probleme jeder Auftragsverarbeitung angesprochen werden, ohne dass wir uns in Details oder Erklärungen einiger Spezialprobleme verlieren. Die Einfachheit des Beispiels dient der Verständlichkeit. Daher verzichte ich auch auf eine Artikelverwaltung. Eine Artikelverwaltung ist vom Prinzip her einer Kundenverwaltung sehr ähnlich, so dass sich von daher keine wesentlichen neuen Aspekte ergeben. Anhand der Aufgabenstellung verdeutliche ich zunächst beispielhaft Begriffe und Aufgabe des Datenbank-Designs und entwickle anschließend gemeinsam mit Ihnen – anhand der vorliegenden Situationsbeschreibung – die Datenbank.

Aufträge zu bearbeiten, ist eine allgemeine Beschreibung für verschiedene verwaltende Tätigkeiten, die in jedem Betrieb anfallen, der mit Waren oder Dienstleistungen handelt. Unter den Begriff „Auftragsbearbeitung“ fällt alles, was von der Akquisition über die Produktion und Lieferung bis hin zur Rechnungsstellung und anschließender Zahlungseingangskontrolle an Informationen zu verarbeiten ist. Unserer Fallstudie liegt das Modell eines Consulting-Unternehmens zugrunde. Aufträge haben in diesem Zusammenhang einen sehr individuellen Charakter und sind hinsichtlich Inhalt und Konditionen nicht standardisierbar. Es werden unterschiedliche Dienstleistungen erbracht, die individuell auf den Kunden abgestimmt sind und die nach einem auftragsspezifisch festgelegten Preis abgerechnet werden. Dienstleistungen können sein: Beratungen, Schulungen, Gutachten, Zeitschriftenartikel und Bücher. Aufgrund der unterschiedlichen Leistungen muss mit unterschiedlichen Mehrwertsteuer-Sätzen fakturiert werden. Folgende Grundfunktionen sind gefordert:

Aufträge verwalten

Der Zweck hierbei ist die Erfassung, Editierung und Auswertung von Aufträgen. Es soll eine Auftragsbestands- und Auftragswerteliste erzeugt werden, die darüber Aufschluss gibt, welche Aufträge für einen bestimmten Zeitraum vorliegen, wie groß deren Volumen ist und welchen Status sie haben. Bei Auftragserteilung werden die Plan-Zahlen bzgl. Fertigstellung und Ergebnis erfasst. Damit sind die zu erwartenden Plan-Einnahmen abschätzbar. So ist einerseits eine Grundlage für die Annahme und Ablehnung von Aufträgen vorhanden und andererseits eine Nachkalkulation der Aufträge möglich.

7.2.1
Seite 2

Leistungen verwalten

Jede erbrachte Leistung wird manuell erfasst. Die Liste „Leistungsübersicht“ gibt Aufschluss über die in einem bestimmten Zeitraum erbrachten Leistungen und ihrer zu berechnenden Werte. Es soll unterschieden werden zwischen berechneten und nicht berechneten Leistungen, damit das abgerechnete und zu berechnende Auftragsvolumen ermittelbar ist. Die Leistungen eines Auftrags müssen jederzeit darstellbar sein. Es sind Zwischensummen je Auftrag gefordert.

Rechnungen erstellen

Die Aufträge sollen für einen frei wählbaren Zeitraum fakturierbar sein. Dabei sind diejenigen Aufträge auszuwählen, die in den angegebenen Zeitraum fallen und bislang noch nicht berechnet wurden. Für diese Aufträge sind die Rechnungen und eine entsprechende Rechnungsausgangsliste zu erstellen. Pro Leistung errechnet sich der Nettoeinzelpreis, indem die Anzahl der geleisteten Ist-Einheiten mit dem festgelegten Satz multipliziert wird. Jede berechnete Leistung soll als solche gekennzeichnet werden. Weiterhin muss die Endsumme über alle Nettobeträge ausgegeben werden. Der ermittelte Nettobetrag erhöht den Kundenumsatz entsprechend. Ebenso ist mit den einzelnen Mehrwertsteuer-Beträgen und der gesamten Mehrwertsteuer zu verfahren. Alles zusammen ergibt den Bruttobetrag.

Kunden verwalten

Es sollen sowohl Kundendaten als auch zugehörige Ansprechpartnerdaten gepflegt werden. Eine Umsatzauswertung einzelner Kunden ist ebenfalls gefordert. Die Umsatzstatistik soll nach unterschiedlichen Kriterien sortierbar sein. Hierzu gehört auch die Auswertung der Kunden ohne Umsatz. Zusätzlich werden selbstverständlich entsprechende Adresslisten gefordert.

Sie sehen, wir realisieren eine einfache Auftragsverwaltung mit wenigen Funktionen. Zentrales Resultat ist dabei die Rechnungserstellung. Ein kleines Beratungsunternehmen hätte hierdurch bereits einigen Nutzen. Man wüsste jederzeit, welche Aufträge zu erledigen sind, welche Termine beachtet werden müssen, welche Leistungen erbracht, aber noch nicht abgerechnet wurden. Einige wichtige Bereiche bleiben in der Fallstudie bewusst ausgeklammert; so die Verwaltung der Ein- und Ausgaben, Meldungen an das Finanzamt usw. Doch dadurch zeichnet sich das relationale Datenbank-Prinzip ja aus: die vorhandene Datenbank kann später jederzeit problemlos erweitert werden.

Doch wie fangen wir an? Der Entwurf eines Informationssystems ist auf den ersten Blick vergleichbar mit dem Schreiben eines Buches. Man sitzt vor einem leeren Blatt Papier und überlegt, wie das Buch inhaltlich zu füllen ist. Nun könnte man einige wichtige Dinge einfach aneinanderreihen und darauf hof-

fen, dass es nachher irgendwie passt. Bei dieser Vorgehensweise ist die Gefahr groß, sich im Detail und den eigentlichen Zweck aus den Augen zu verlieren. So etwas kann auch bei der Entwicklung einer Datenbank-Anwendung geschehen. Meist ist die Zeit ohnehin knapp und meist gibt es ja auch bereits ähnlich realisierte Anforderungen. Derartige Überlegungen schränken allerdings die Entwurfsfreiheit stark ein. Außerdem wird die Formulierung der Anforderungen durch „Vorlieben“ der Programmierer beeinflusst. Hinzu kommt, dass der Aufwand und die Kosten für Modifikationen umso größer werden, je später der Wunsch oder die Notwendigkeit erkannt wird und je weiter das Projekt schon realisiert ist. Es spricht also einiges dafür, am Anfang des Projekts – bei der Analyse und den Entwurfsphasen – genügend Zeit einzuplanen. Ein Hauptanliegen dieses ersten Beitrags ist es, hier einige praktische Anleitungen und Hilfestellungen zu geben.

7.2.1
Seite 4

7.2.2 Die Systemanalyse

Ausgangspunkt für das Datenbank-Design ist die Systemanalyse. In dieser Phase werden u. a. die folgenden Fragen geklärt:

- Welche Daten benötigt das User-Interface?
- Welche Aufgaben sind gefordert?
- Welche Daten bezüglich der Kunden, Aufträge und anderer Objekte müssen wir speichern?
- Welche Daten geben die Anwender ein?
- Welche Veränderungen der Eingabedaten sind erforderlich, um die geforderten Ausgabedaten zu erhalten?

Basis sind dabei die mehr oder weniger vorhandenen nebulösen Vorstellungen der Benutzer über das zukünftige Produkt. Sie ergeben sich aus der täglichen Büroarbeit, bei der die Ideen geboren werden, was man alles automatisieren müsste. Handelt es sich um mehrere Personen, die ein Sachgebiet bearbeiten und dabei – wie auch immer – Daten austauschen, so werden Sie sehr häufig unterschiedliche, einander widersprechende Anforderungen an die Lösung erhalten. Zudem umfasst beispielsweise der Begriff „Auftragsbearbeitung" für die damit befassten Sachbearbeiter eine Selbstverständlichkeit, die als Ganzes gesehen wird. Will man das „Ganze" jedoch als Informationssystem abbilden, so muss man es in überschaubare abgegrenzte Objekte und Tätigkeiten zerlegen. Es ist Aufgabe der Systemanalyse in einer Anforderungsdefinition, die seitens der Anwender geäußerten Vorstellungen in eine aussagekräftige, sinnvolle und geordnete Form zu bringen. Die Anforderungsdefinition ist ein Dokument, das genau festlegt, **was** das System leisten soll. Jede zu erbringende Leistung und alle dafür notwendigen Daten müssen qualitativ beschrieben werden. Alle Details über das **Wie** der Realisierung haben in der Anforderungsdefinition nichts zusuchen. Der Benutzer braucht sich sicherlich nicht darum zu kümmern – und der Programmierer tut gut daran, die Details zum jetzigen Zeitpunkt auszuklammern. Logischerweise ist erst einmal zu klären, was benötigt wird, um sich dann in einem zweiten Schritt zu überlegen, wie man es erhält.

Hierfür muss eine gemeinsame Sprache gefunden werden, die es erlaubt Anforderungen so zu definieren, dass sowohl der Anwender als auch der IT-Spezialist versteht, was gemeint ist. Außerdem muss es möglich sein, das Ergebnis auf Vollständigkeit zu prüfen. Für den ersten Punkt könnte eine rein verbale Beschreibung der Lösung ausreichen. Allerdings ist ein Textdokument nur schwer auf Vollständigkeit und Konsistenz zu prüfen. Eine Kombination aus grafischer und verbaler Darstellung könnte den Widerspruch lösen.

7.2.2
Seite 2

Fassen wir kurz zusammen. Eine Anforderungsdefinition sollte mindestens folgende Informationen enthalten:

- Beschreibung der Gesamtaufgabe
- Beschreibung der Leistungen, die erbracht werden sollen
- Beschreibung der Daten, die bereitzuhalten oder einzugeben sind
- Beschreibung wesentlicher Konsistenzbedingungen (Wertebereiche, Plausibilitäten)
- Beschreibung der Zugriffsrechte auf die Daten
- Beschreibung der Dateneingaben durch den Benutzer
- Beschreibung der Schnittstellen für Datenein- und -ausgaben

Das Resultat der Systemanalyse muss demnach eine exakte für Fachleute und Nicht-Fachleute gleichermaßen verständliche Formulierung der Anforderungen sein, aus der sich letztendlich Daten und Programmstrukturen ableiten lassen.

7.2.2.1 Die Analyse-Methode „Structured Analysis"

Für die Analyse unserer Fallstudie stütze ich mich auf die Methode „Structured Analysis", die erstmals 1978 von Tom de Marco beschrieben wurde („Structured Analysis and System Specification"). Kern der Methode ist die Analyse der Datenflüsse. Die Daten werden im Hinblick auf Entstehung, Verwendung und Veränderung hin untersucht und dargestellt. Alle Implementierungsdetails – wie Algorithmen oder der Datenbankentwurf – bleiben hierbei zunächst unberücksichtigt. Die einzelnen Datenfluss-Diagramme werden als Baum in einem hierarchischen Modell angeordnet. Innerhalb eines SA-Modells müssen die Datenflüsse zwischen Kind- und Elterndiagramm ausbalanciert sein. Das heißt: Die Datenintegrität muss gewährleistet sein.

Natürlich gibt es vielfältige Werkzeuge und Methoden, die allerdings häufig sehr umfangreich sind und damit den Rahmen des Buches sprengen würden. Außerdem sind die Tools meistens sehr teuer, so dass sich der Einsatz eher für größere Projekte lohnt. Außerdem ist eine gewisse Einarbeitungszeit erforderlich, um die Methoden und Werkzeuge zu beherrschen. „Structured Analysis" kommt dagegen mit wenig Sprachkonstrukten und Symbolen aus und ist somit leicht zu erlernen. Genau richtig, um unsere Fallstudie zu analysieren und zu dokumentieren, denn schließlich soll es in dem Buch nicht vorrangig um das Thema „Systemanalyse", sondern vielmehr um das Thema „Datenbankdesign und -nutzung" gehen.

7.2.2.1.1 Datenfluss-Analyse der „Auftragsverwaltung“

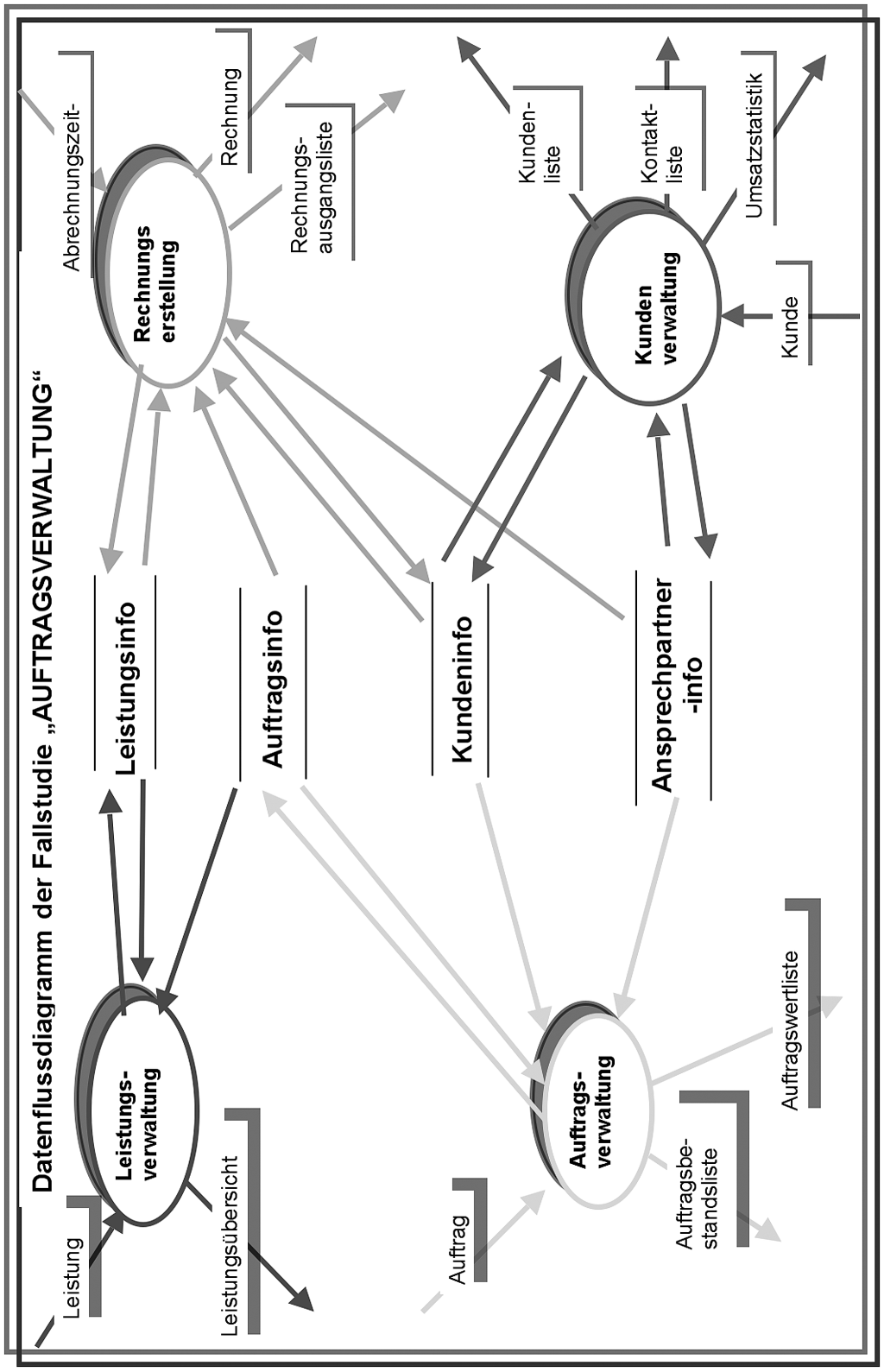
Ein Datenfluss ist ein „Kanal“, durch den Informationen von bekanntem Aufbau fließen. Die grafische Darstellung ist ein benannter Pfeil.

Länge und Form des Pfeils sind unerheblich. Die Spitze weist zum Ziel – gibt also die Richtung an. Der Name des Datenpfeils (DF) muss eindeutig sein und deutet den Inhalt an. Im zugehörigen Datenwörterbuch erhält jeder Datenpfeil eine formalisierte Beschreibung seines Inhalts. Daten können in einen Prozess hineinfließen, aus ihm herauskommen oder von einem Prozess an einen anderen übergeben werden. Danach lässt sich zwischen inneren und äußeren Datenflüssen unterscheiden. Als äußeren Datenfluss bezeichnet man einen selbstständigen Datenfluss, der in das System hinein- oder hinausfließt. Äußere Datenpfeile sind also Schnittstellen zur Umgebung. Ein interner Datenfluss fungiert innerhalb des Systems als Schnittstelle zwischen Prozessen und/oder Datenspeichern. Alle komplexen Aktionen bezeichnet man als Transformation (Prozess), die einen oder mehrere Ausgangsdatenflüsse umformt. Grafisch wird die Transformation als Kreis dargestellt, der eindeutig benannt ist.

Während der Anforderungsdefinition werden alle Formen von Informationsablagen als Datenspeicher bezeichnet. Hierbei kann es sich um eine einzelne Datei, aber auch um eine Verknüpfung mehrerer Datenbanktabellen handeln. Jeder Datenspeicher wird im Datenwörterbuch beschrieben. Jeder Datenspeicher hat mindestens einen eingehenden und einen ausgehenden Datenpfeil, denn ansonsten ergibt sich kein praktischer Sinn: Denn die Tabelle würde keine Daten enthalten oder diese würden nicht verwendet werden, wären also überflüssig.

Durch den Einsatz der Methode „Structured Analysis“ während der Analyse werden Sie somit zu einer einheitlichen und präzisen Darstellung des gewünschten Datenflusses gezwungen. Konsistenzregeln helfen zudem Fehler zu vermeiden. Sie können mit dieser Methode große Systeme in kleine, überschaubare Teilsysteme aufspalten. Betrachten wir nun anhand dieser Darstellungsmethode unsere Fallstudie „Auftragsverwaltung“.

7.2.2
Seite 4


Datenflussdiagramm der Fallstudie „AUFTRAGSVERWALTUNG“
Abrechnungszeit-
Rechnung
Rechnungs-
ausgangsliste
Rechnungs
erstellung
Kunden-
liste
Kontakt-
liste
Umsatzstatistik
Kunden
verwaltung
Kunde
Leistungsinfo
Auftragsinfo
Kundeninfo
Ansprechpartner
-info
Leistungs-
verwaltung
Leistung
Leistungsübersicht
Auftrags-
verwaltung
Auftrag
Auftragsbe-
standsliste
Auftragswertliste

7.2.2.1.2 Data Dictionary „Auftragsverwaltung“

Eine wichtige Information für die Datenbank sind später die Metadaten über die gespeicherten Objekte und deren Strukturen. Diese Metadaten werden auch als Data Dictionary bezeichnet. Das Data Dictionary enthält Metadaten zur Dokumentation der in der Datenbank gespeicherten Datenobjekte und -attribute. Auch Informationen zur Datenverwendung, durch Anwendungen, Benutzer und Transaktionen können hier dokumentiert werden. Ich verwende ein solches Data Dictionary zunächst, um die ermittelten Datenflüsse verbal zu erläutern. Als Zugeständnis an die spätere Implementierung verwende ich jetzt bereits Namen, die auch als Spaltennamen einer Datenbank-Tabelle in Frage kommen.

Für die Definition der Anforderung bedeutet dies keine wesentlichen Einschränkungen, erleichtert aber die spätere Zuordnung von Datenbank-Spalten zu Datenelementen in der Definition. Bei der Namensgebung sollten Sie bereits jetzt einige Regeln beachten, die allerdings nicht zwingend sind. Grundsätzlich sollten Sie sich für kurze, aber dennoch sprechende Namen entscheiden. Da man beim Umgang mit der Datenbank ständig genötigt ist, Datenelemente – später Spaltennamen – anzugeben, erscheint jedes zusätzliche Zeichen als zusätzliche Mühe. Versuchen Sie gleichartige Datenelemente, die in unterschiedlichen Zusammenhängen genutzt werden, mit dem Anfangsbuchstaben des übergeordneten Datenflusses zu kennzeichnen. Das Kürzel AKURZ ist Bestandteil des Datenelements AUFTRAG, das Kürzel KKURZ gehört zum Datenelement KUNDE. Die Datenelemente definiere ich zu diesem Zeitpunkt ausschließlich durch ihre Namensgebung und falls erforderlich durch ergänzende Kommentare. Erst sehr viel später werden Datentypen, Längen und Wertebereiche festgelegt.

Außerdem müssen Sie noch einige syntaktische Vorschriften beachten:

Dem Namen des Datenflusses folgt ein Gleichheitszeichen. Aufzählungen einzelner Datenelemente werden durch ein Pluszeichen (+) voneinander getrennt:

Beispiel Datenfluss KUNDE


```
KUNDE =
      FIRMA
   +  STRASSE
   +  ORT
   +  {ANSPRECHPARTNER}
   +  (HOBBY)                    @ guter Kunde @
```

Grundwerk 4/2007

Wiederholungen von Datenelementen werden in geschweiften Klammern angegeben. Solche mehrfach auftretenden Datenelemente werden anschließend zusätzlich dokumentiert. Optionale Datenelemente kennzeichnet man durch runde Klammern. Kommentare sind in @@ eingeschlossen. Eine Aufzählung alternativer Werte stellen Sie durch eckige Klammern dar und trennen die einzelnen Werte durch geraden Querstrich (|). Querverbindungen zwischen den Datenflüssen vermerke ich mit dem Hinweis „used in *name des Datenflusses*".

Dokumentieren wir nun einige der Datenflusselemente exemplarisch in unserem Data Dictionary. Das vollständige Datenwörterbuch der Anforderungsdefinition finden Sie im Anhang. Bei der Beschreibung unterschiedlicher Datenflüsse müssen Attribute, die in anderen Datenflüssen genutzt werden, natürlich berücksichtigt werden. So wird in der Definition des Datenelementes RECHNUNG beispielsweise das Attribut AKURZ verwendet.

RECHNUNG =

	RNR	@eindeutige Rechnungsnummer@
+	**AKURZ**	**@Auftragskennzeichen@**
+	KKURZ	@Kundenkennzeichen@
+	FIRMA	
+	ZUSATZ	
+	NACHNAME	@Name des Ansprechpartner@
+	STRASSE	
	:	
	:	

Dieses muss sich folglich im Datenfluss AUFTRAG wieder finden.

AUFTRAG =

	AKURZ	**@ eindeutiger Schlüssel @**
+	**KKURZ**	**@ Kundenkennzeichen @**
+	**APKURZ**	**@ Kennzeichen Ansprechpartner @**
+	ADATUM	@ Auftragseingangsdatum @
+	FERTIGIST	@ tatsächliche Fertigstellung; der Auftrag ist fertig, wenn die letzte Leistung fertig gestellt ist@
	:	
	:	

Die Daten TEL, FAX, MAIL, PAGE und INFO sowie UMSATZ ergeben sich aus der Anforderung für die Kunden- und Umsatzstatistiken. Das Attribut PARTNER erscheint in geschweiften Klammern. Das bedeutet, dass es mehrfach pro Kunde auftreten kann. Die Anzahl der Ansprechpartner variiert je Kunde.

```
KUNDE =
      KKURZ                    @Schlüssel @
  +   FIRMA
  +   ZUSATZ                  @ zum Firmennamen @
  +   STRASSE
  +   LKZ                      @ Länderkennzeichen @
  +   PLZ
  +   ORT
  +   TEL
  +   FAX
  +   MAIL
  +   PAGE
  +   INFO                     @ beliebiger freier Text @
  +   UMSATZ
  +   {PARTNER}                @ evtl. mehrere Ansprechpartner @
```

Jedes mehrfach vorkommende Datenelement wird separat im Data Dictionary erläutert.

```
PARTNER =
      @    Verwaltung der Ansprechpartner von Firmen.
           Ein Ansprechpartner kann mehrere Firmen vertreten @

           PKURZ                          @ Partnernummer @
  +        VORNAME
  +        NACHNAME
  +        ANREDE
  +        {APA}                    @ firmenspezifische Daten @
```

used in: KUNDE

Außerdem weist das Element PARTNER eine weitere Besonderheit auf: Wenn Sie sich das Element AUFTRAG anschauen, werden sie feststellen, dass auch dort eine Verbindung über das Attribut PKURZ zu diesem Datenelement existiert. Dies ist auch richtig, wenn wir davon ausgehen, dass in einer Firma mehrere Ansprechpartner existieren, und somit festgehalten werden soll, mit wem der jeweilige Auftrag abgesprochen wurde. Dennoch muss PARTNER natürlich als Element von KUNDE verwaltet werden und nicht etwa über das Datenelement AUFTRAG.

Eine weitere Überlegung zum Datenelement PARTNER führte zu dem Schluss, dass ein- und dieselbe Person, Ansprechpartner für verschiedene Firmen sein kann. Zwar ist dies nicht die Regel, es muss aber möglich sein, diese Fälle korrekt abzubilden. Die firmenspezifischen Daten „Abteilung“, „Rufnummer bzw. Telefondurchwahl“ sowie die Funktion der Ansprechpartner könnten also mehrfach vorkommen und müssten in geschweiften Klammern stehen. Nur

7.2.2
Seite 8

bilden diese Daten in Bezug auf den Kunden eine Einheit. Wir fassen daher die kundenspezifischen Angaben als eigenständige Einheit unter dem Namen APA zusammen und kennzeichnen es als wiederholendes Datenelement im Element PARTNER.

```
APA =
          KKURZ          @ Bezug zur Firma @
    +     TEL            @ Rufnummer/Durchwahl @
    +     FUNKTION       @ Stellung in der Firma @
    +     INFO           @ allgemeine Kurzinfo als freier Text @
    +     ABTLG

used in: PARTNER
```

Alles scheint soweit in Ordnung. Nur, wenn wir jetzt den Datenfluss KUNDE auflösen, ergibt sich ein Widerspruch.

```
KUNDE =
          KKURZ                    @ Kundennummer @
          :
          :
          {PARTNER}
               PKURZ               @ Partnernummer @
               :
               :
               {APA}
                    KKURZ          @ spezifische Kundendaten des
                    :                Ansprechpartners @
                    :
```

Sie können folgenden Sachverhalt erkennen: Ein Kunde kann mehrere Ansprechpartner haben. Soweit so gut. Einem Ansprechpartner können mehrere firmenspezifische Daten zugeordnet sein, weil die Ansprechpartner ja verschiedene Firmen vertreten können. Und hier liegt das Problem, denn das würde bedeuten, dass für den Kunden 112 die Funktion eines Angestellten, der einen anderen Kunden betreut, festgehalten wird. Dies ergibt natürlich keinen Sinn. Der Widerspruch lässt sich nur auflösen, indem APA getrennt vom PARTNER und vom KUNDEN gesehen wird, wobei der Bezug zu beiden Datenelementen klar sein muss.

Als Letztes fehlen die Beschreibungen der Datenspeicher. Erinnern Sie sich? Alle benötigten Datenablagen werden als Datenspeicher bezeichnet und müssen im Data Dictionary separat beschrieben werden. Bedenken Sie aber, dass wir noch nicht in der Datenbank-Realisierung sind und daher die hier beschriebenen Datenspeicher in dieser Form nicht als Datenbank-Tabelle erstellt werden! Ich verzichte an dieser Stelle auf eine vorläufige Beschreibung und komme auf das Data Dictionary nach dem Datenbank-Design zurück. Erst dann beschreibe ich meine Relationen.

Wollte man ein Anwendungsprogramm für die geschilderte Ausgangssituation entwickeln, müssten auch die Transformationen (Prozesse) jetzt weiter verfeinert werden – solange bis sogenannte Elementarfunktionen übrig bleiben, die sich nicht weiter teilen lassen. Das ist jedoch nicht unser Ziel! Unsere Definition für den Entwurf einer Datenbank ist ausreichend.

Später können wir die geforderten Leistungen durch interaktive SQL-Abfragen umsetzen. An einigen Stellen bietet es sich auch an, Prozeduren, Funktionen oder embedded SQL-Programme zu entwickeln.

Doch zunächst geht es darum, die vorliegende Anforderungsdefinition in ein relationales Datenmodell umzusetzen.

7.2.2
Seite 10

7.2.3 Die Datenbank-Beschreibung

Die Anforderungsdefinition legt die Problemstellung im Einzelnen fest. Die praktischen Anforderungen stehen im Zentrum der Analyse. Jetzt geht es darum, den Wirklichkeitsausschnitt in ein Modell abzubilden, das sich so verhalten kann wie die realen Objekte der Beobachtung.

Um dieses Ziel zu erreichen, müssen zunächst die von den Benutzern formulierten unterschiedlichen Datensichten auf Überschneidungen und eventuelle Widersprüche untersucht werden. Daraus ergibt sich das „konzeptionelle Schema", das auch „logisches Schema" genannt wird. Das „konzeptionelle Schema" ist bereits normalisiert und somit redundanzfrei und enthält die verschiedenen Entitätsmengen mit ihren Beziehungen zueinander.

Anschließend wird in einem weiteren Schritt das „Relationenschema" festgelegt. Es enthält die Namen der Relationen und ihrer Attribute, sowie Angaben über deren Wertebereiche und eventuelle Integritätsbedingungen.

Erst zum Schluss wird das „Implementierungsschema" erstellt, das die zu erwartenden Datenzugriffe und gegebenenfalls deren Optimierungen durch Indexe definiert.

Das konzeptionelle Schema dient gemeinsam mit dem Relationenschema sowie dem Implementierungsschema als Datenbank-Beschreibung. Anhand dieser Beschreibung kann anschließend die Datenbank realisiert werden – einschließlich aller Tabellen und deren Indizes sowie der Zugriffsberechtigungen. Das Resultat ist eine leere Datenbank, ein Gerüst, dem nur noch die Daten fehlen! Der Datenbank-Entwurf bildet zudem die Grundlage für den Anwendungsentwurf. Erst jetzt können Masken, Menüs, Berichte und verarbeitende Programme erstellt und zu einem Gesamtsystem zusammengefügt werden.

7.2.3.1 Begriffsdefinition

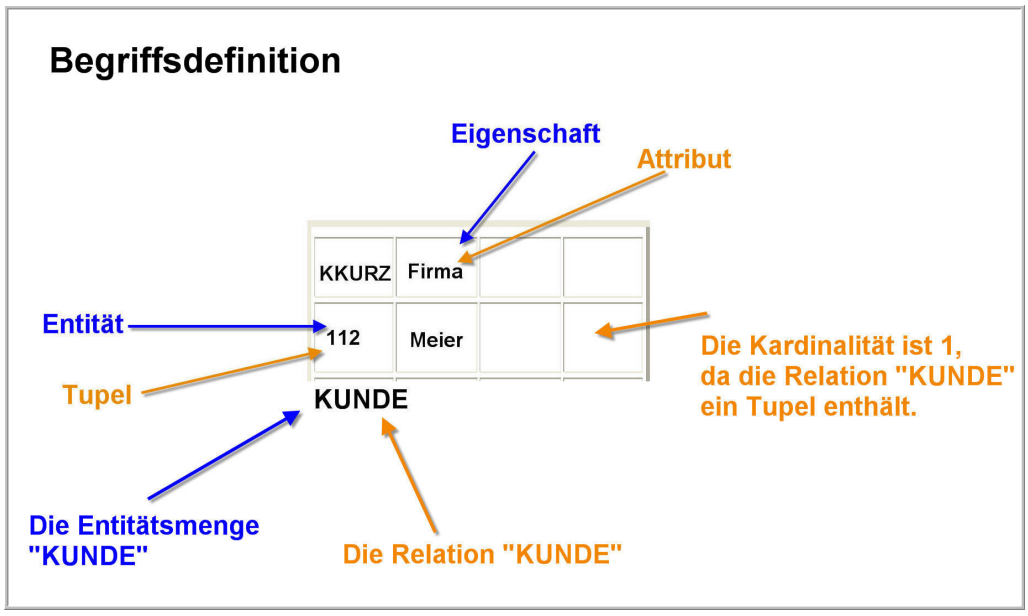
Wenn eine Anforderungsdefinition vorliegt, stellt man sich sehr schnell die Fragen: „Wie sollen die Daten jetzt strukturiert werden?" und „Welche Datenfelder bilden eine Einheit?" Der Grundgedanke beim Entwurf einer Datenbank ist die Abbildung des zu entwerfenden Systems in Form von sogenannten Entitäten bzw. Relationen. Etwas weniger technisch gesprochen: Man versucht ein System durch Datenobjekte und logische Einheiten zu beschreiben und untersucht die Verbindungen zwischen diesen Einheiten. Das Konzept wurde erstmals 1976 von P.P.-S. Chen formuliert. Mittlerweile existieren aber in den entsprechenden Fachdiskussionen vielfältige Auslegungen und eine standardisierte Symbolik, die wir an dieser Stelle sicherlich nicht detailliert besprechen müssen.

7.2.3
Seite 2

Vielmehr möchte ich Ihnen anhand unserer Fallstudie nur einen gangbaren Weg aufzeigen, um die Daten in eine entsprechend strukturierte Form zu bringen. Auf dieser Stufe des Entwurfs werden keine Überlegungen angestellt, welche Felder mit welchen Datentypen und dieser oder jener Anzahl von Dezimalstellen benötigt werden. Diese Überlegungen gehen zu weit in die Tiefe des zukünftigen DB-Systems. Sehr detaillierte Definitionen zu diesem Zeitpunkt erschweren die Arbeit des weiteren Entwurfs. Es besteht die Gefahr, sich früh in Feinheiten zu verlieren. Besser ist es, sich zunächst einmal einen Überblick über das künftige System zu verschaffen. Der Datenzusammenhang innerhalb der Anwendung ist unbedeutend. Was wir brauchen, sind Entitätsmengen.

Eine Entität (engl. Entity) ist lediglich ein wohlunterscheidbarer Sachverhalt aus der Realität. Jeder Kunde ist ein solch unterscheidbares Subjekt, aber auch jeder AUFTRAG lässt sich voneinander unterscheiden. Mehrere gleichartige Entitäten werden zu einer Entitätsmenge zusammengefasst. Solch eine inhaltslose Blackbox kann eigentlich jedes Subjekt oder Objekt sein. Eine Entität – d. h. ein Dateneintrag – wird durch Eigenschaften (engl. Property) gekennzeichnet. Die Eigenschaften haben einen Namen und einen Wertebereich. Die in der Anforderungsdefinition zugehörigen Datenelemente KKURZ, FIRMA, TEL usw. sind also Eigenschaften der Entität KUNDE.

Jede Entitätsmenge lässt sich aber auch als Relation abbilden. Eine Relation entspricht somit einer Menge von Daten. Die Relation erhält einen Namen und verbindet einzelne Eigenschaften zu Elementen – sogenannten Tupel. Eine Entität entspricht also einem Element oder besser gesagt einem Tupel. Eine Relation kann ein, kein oder viele Tupel (Elemente) enthalten. Die Anzahl vorhandener Tupel wird als Kardinalität bezeichnet. Die Kardinalität ist Null (0), wenn in der Relation keine Tupel vorhanden sind.



Begriffserläuterung

Verschiedene Entitäten – also die einzelnen Kunden – derselben Entitätsmenge KUNDE können identische Eigenschaftswerte ausweisen. So kann es mehrere Kunden geben, die in Bremen wohnen. Wird aber der Kunde A, der in Bremen wohnt, mehrfach mit seinen spezifischen Attributen gespeichert, liegt Redundanz vor. Solche Wiederholungen sind einerseits speicherintensiv, andererseits eine große Fehlerquelle, denn Sie müssten in diesen Fällen sicherstellen, dass alle Redundanzen stets entsprechend aktualisiert sind. Wenn die redundanten Informationen aber in verschiedenen Kontexten unterschiedliche Werte annehmen, führt dies dazu, dass das Informationssystem für gleichartige Abfragen in den betrachteten Kontexten unterschiedliche Antworten liefert. In einem solchen Fall spricht man von Daten-Inkonsistenz (engl. data inconsistency) oder von Mutationsanomalien.

Außerdem muss jedes Tupel (Entität) derselben Relation eindeutig unterscheidbar sein. Entweder geschieht dies durch die Kombination aller Attribute oder einer hinreichenden Teilmenge davon: dem Relationenschlüssel. Eine minimale Attributkombination pro Relation nennt man Primärschlüssel (engl. primary key). Oft werden dafür künstliche Attribute festgelegt. Ein solcher Schlüsselwert repräsentiert demzufolge ein ganzes Tupel (Element). Das Attribut KKURZ ist Primärschlüssel der Relation KUNDE. Jede Relation benötigt einen Primärschlüssel. Meist müssen dafür künstliche Attribute eingeführt werden, weil die natürlichen Attribute schlecht kontrollierbar sind. Falls Sie auch nur vermuten, dass ein natürliches Attribut mehrfach vorkommen oder für einen Eintrag fehlen könnte, ist das Attribut als Primärschlüsselkandidat ungeeignet.

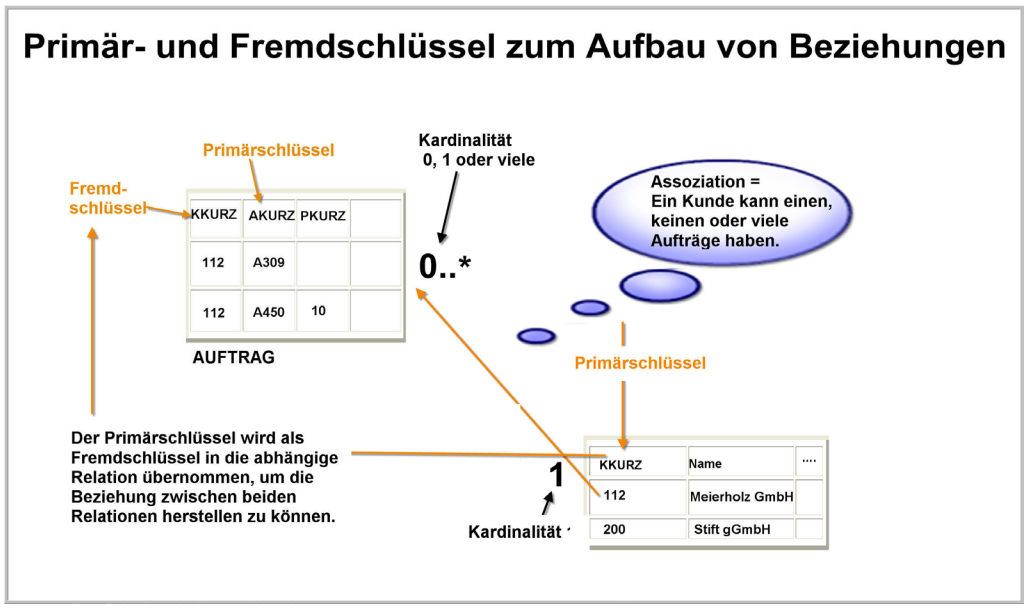
Des Weiteren müssen Sie bedenken, dass verschiedene Entitäten in Kontakt zu einander stehen. Der Kunde A kann zum Beispiel im Laufe der Zeit mehrere Aufträge erteilen. Solche regelmäßig anzutreffenden Verbindungen werden als Beziehung (engl. Relationship oder Assoziation) definiert.

Die wichtigsten Abhängigkeiten sind **funktionale Abhängigkeiten** oder **Inklusionsabhängigkeiten**. Von einer funktionalen Abhängigkeit (engl. functional dependency) spricht man, wenn zwei Attribute derselben Entität voneinander abhängig sind.

Inklusionsabhängigkeiten (engl. inclusion dependency) bestehen hingegen zwischen den Attributen unterschiedlicher Relationen. Durch Inklusionsabhängigkeiten können Verknüpfungen zwischen Relationen realisiert werden. So kann man bestimmen, dass für jeden Kunden kein, ein oder mehrere Ansprechpartner vorhanden sind. Inklusionsabhängigkeiten stellen demzufolge sicher, dass sich die Beziehungen zwischen den Entitäten auflösen lassen. Sind in einer Datenbank alle Abhängigkeiten korrekt erfüllt, ist die Datenbank konsistent.

Um Inklusionsabhängigkeiten aufzulösen oder besser gesagt, um von einer Entität auf eine andere Entität zu verweisen, wird der Primärschlüssel der Relation A als Fremdschlüssel in der Relation B hinterlegt. Beide – Primär- und Fremdschlüssel – sollten den selben Attributnamen erhalten. Soll nun auf eine

andere Relation Bezug genommen werden (welcher Kunde gehört zum Auftrag 4711?) reicht die Verwendung seines Schlüsselwerts für den Zugriff aus – das heißt: Für die Assoziation wird das Attribut KKURZ in der Relation AUFTRAG verwendet. KKURZ in AUFTRAG hat somit nur die Funktion, auf eine andere Relation zu verweisen. In diesen Fällen spricht man von einem Fremdschlüssel (engl. foreign key).



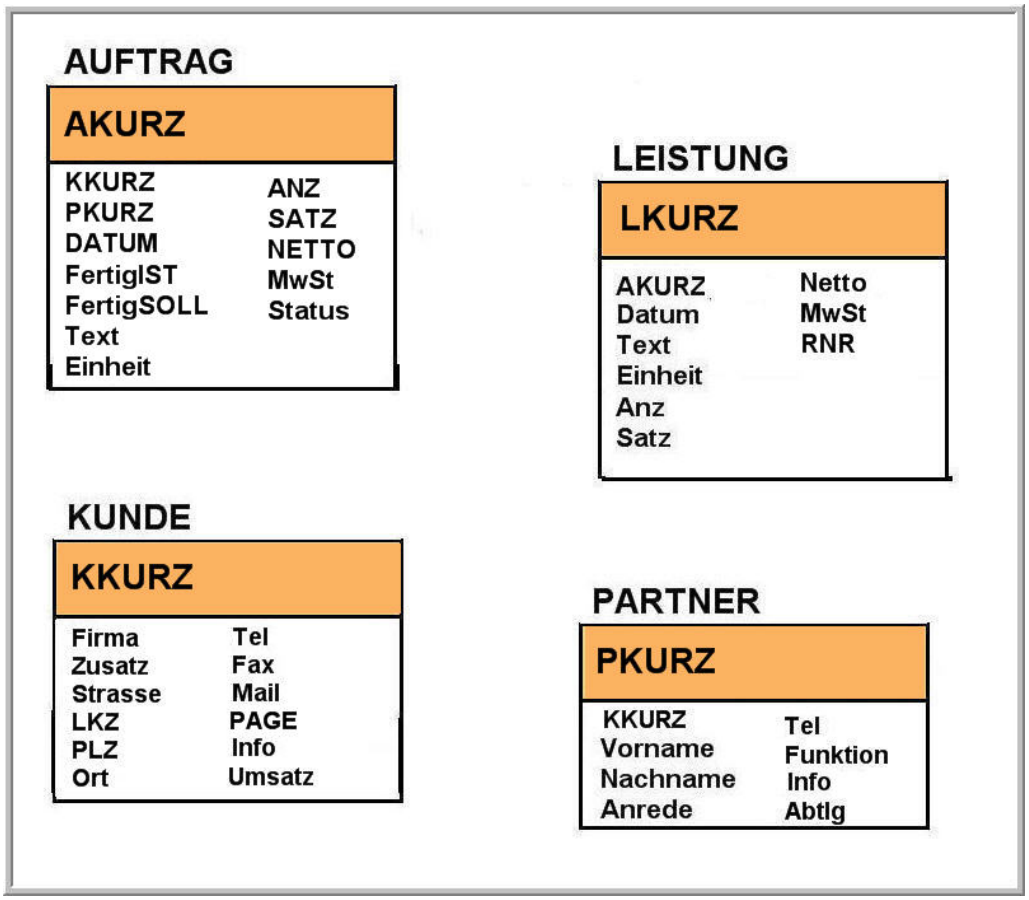
Primär- und Fremdschlüsseldefinition

Sind Sie jetzt von den vielen Begriffen verwirrt? Und fragen sich, was das alles mit unserer Datenbank-Beschreibung zu tun hat? Nun ja, die Frage ist sicherlich berechtigt. Zumal in der Praxis die Begriffe des Entity-Relationship-Konzeptes und Begriffe des Relationen-Konzeptes oftmals beliebig ausgetauscht werden. So spricht der Eine von Assoziationen, der Nächste von den Kardinalitäten und der Übernächste verwendet den Begriff Relation anstelle von Entitätsmenge. Dies spielt jedoch für unseren Datenbank-Entwurf keine Rolle und es ist egal, welche Begriffe Sie nutzen. Nur eines sollten Sie zu diesem Zeitpunkt vermeiden: Relationen werden als Tabelle abgebildet, so dass man schnell versucht sein könnte, eine Relation als Tabelle oder ein Tupel als Zeile zu bezeichnen. Doch diese Begriffe sollten ausschließlich der späteren Realisierungsphase vorbehalten sein.

7.2.3.2 Der Datenbank-Entwurf

Wir benötigen als Erstes mögliche Kandidaten für unsere Entitäten. Oft liefert eine verbale Problembeschreibung bereits erste Ansätze. Schauen Sie sich bei einer verbalen Problembeschreibung einfach die Hauptworte an. Wir haben es einfacher! Aufgrund unserer Anforderungsdefinition können wir bereits erste

Entitäten abbilden. Wir wollen uns an dieser Stelle noch keine Gedanken darüber machen, ob die entdeckten Entitäten das System vollständig beschreiben, zumal es auch aus mathematischer Sicht durchaus zweifelhaft ist, ob Systeme ab einer gewissen Komplexität noch vollständig beschreibbar sind. Setzen wir uns das Ziel, eine weitgehende Abdeckung zu erreichen. Bisher war nur die Rede von Entitäten. Noch fehlen uns die Eigenschaften. Doch auch da haben wir in der Anforderungsdefinition vorgesorgt, so dass sich für uns die folgende Ausgangssituation ergibt.



Die Ausgangssituation

Bei dem von mir gewählten Top-Down-Ansatz fragt man sich, welche Objekte, Subjekte oder Akteure an dem gefundenen Vorgang beteiligt sind beziehungsweise wer Vorgangsdaten erzeugt und wen sie betreffen. Häufig verwendet man für diesen Ansatz Entity-Relationship-Diagramme. Allerdings ist es unerheblich, ob man ER-Diagramme benutzt oder eine andere Notation.

Unsere Ausgangssituation scheint bereits sehr komfortabel. Nur ist der Entwurf von Datenbanken kein linearer Prozess nach einem simplen Rezept. Es müssen ständig syntaktische und semantische Validierungen vorgenommen werden. Mehrere Entwurfsschritte sind noch offen, bevor wir unsere Datenbank implementieren können.

Da das Entity-Relationship-Modell in der Objektsicht schwerfällig und schnell unübersichtlich wird, benutze ich zusätzlich die Schreibweise des Relationenmodells. Wegen seiner einfachen Handhabbarkeit ist es als Zwischenschritt weiterhin nützlich. Letztendlich ist die Form der Darstellung unerheblich. Sie entscheiden, welche Darstellungsweise Ihren Ansprüchen genügt, um in einem nächsten Entwicklungsschritt die Daten auf redundante Speicherungen hin zu prüfen. Hier setzt das Verfahren der Datenbank-Normalisierung ein.

7.2.3.2.1 Die Normalisierung der Datenbank

Für die Definition von Relationen existiert ein umfangreiches formales Regelwerk. Hierzu gehört auch das analytische Verfahren der Datenbank-Normalisierung. Bei der Normalisierung (engl. normaliziation) eines relationalen Schemas werden die Attribute derart auf die Relationen verteilt, dass beim Einfügen, Ändern oder Löschen von Datensätzen keine Inkonsistenzen auftreten können. Die Normalisierung hat somit zum Ziel, inkorrekte oder fehleranfällige Konstruktionen bereits beim Entwurf der Relationen zu verhindern. Der Prozess der Normalisierung führt in der Regel zu einer Aufspaltung und/oder Neudefinition von Relationen. Stellen Sie sich zunächst einmal die Attribute Ihrer Entitäten als Tabelle vor. Ich verwende zunächst die Entitätsmenge KUNDE.

KUNDE

KKURZ	Firma	...	LKZ	Strasse	PLZ	...
100	Versicherungspartner		D	Hauptweg 8		

In der **ersten Normalform** dürfen in jeder Relation nur atomare Attribute gespeichert werden – d. h.: Sie erhalten Inhalte, die nicht weiter teilbar sind. Daher wird jetzt auch das Attribut STRASSE vom Attribut HAUSNR getrennt.

KUNDE

KKurz	Firma	...	Strasse	HausNr	PLZ	...
100	Versicherungspartner		Hauptweg	8		

Schauen wir uns als nächstes die Relation Partner an:

PARTNER

PKURZ	KKURZ	Vorname	Nachname	Anrede	Tel	Abtlg	Funktion
1	100	Klaus	Willde	Herr	04792-616	Verkauf	Sachbearbeiter
1	500	Klaus	Willde	Herr	04281-1577	Verkauf	Sachbearbeiter
1	250	Klaus	Willde	Herr	-37	Verkauf	Sachbearbeiter

Ein Ansprechpartner kann laut Anforderungsdefinition mehrere Firmen repräsentieren. Wir müssten also demnach entweder die Namen der Ansprechpartner und gegebenenfalls weitere kennzeichnende Attribute redundant speichern oder die firmenspezifischen Attribute mehrfach in einer Zelle hinterlegen. Beides ist unzulässig, denn in der **zweiten Normalform** muss jedes Nicht-Schlüsselattribut einer Relation voll funktional abhängig vom Schlüssel derselben Relation sein. Unser Primärschlüssel ist die Attributkombination PKURZ und KKURZ. Wenn jetzt aber die Kontaktdaten eines Ansprechpartners geändert, gelöscht oder neu hinzugefügt werden, kann es zu Mutationsanomalien kommen, denn theoretisch wäre es möglich einen neuen Ansprechpartner mit einer bereits bestehenden Nummer (PKURZ) zu speichern, wenn der Primärschlüssel eindeutig bleibt.

PARTNER

PKURZ	KKURZ	Vorname	Nachname	KoTel	Abteilung	Funktion	...
1	100	Klaus	Willde	04792-616	Verkauf	Sachbearbeiter	
1	500	Klaus	Willde	04281-1577	Verkauf	Sachbearbeiter	
1	250	Klaus	Willde	-37	Verkauf	Sachbearbeiter	
1	300	Marlies	Knauff	089-99875	Verkauf	Sachbearbeiter	

Nur weil die Schlüsselkombination PKURZ = 1 und KKURZ = 300 noch nicht in der Relation existiert, kann der Eintrag gespeichert werden. Das gleiche gilt natürlich auch für den umgekehrten Sachverhalt. Solche Mutationsanomalien deuten stets darauf hin, dass in einer Relation verschiedene selbstständige Sachverhalte abgebildet werden, die sich unabhängig voneinander entwickeln können.

Daraus folgt, dass wir zwei unabhängige Relationen bilden müssen. Die Relation PARTNER erhält den Primärschlüssel PKURZ. Dadurch wird die mehrfache Verwendung der Nummer PKURZ ausgeschlossen.

PARTNER

PKURZ	Vorname	Nachname	Anrede
1	Klaus	Willde	Herr
2	Marlies	Knauff	Frau

Die neue Relation APA dient als Beziehungsmenge. Sie stellt die Verbindung zwischen der Relation KUNDE und PARTNER her.

7.2.3
Seite 8

APA

APKURZ	PKURZ	KKURZ	Tel	Funktion	...
1	1	100	04792-616	Sachbearbeiter	
2	1	500	04281-1577	Verkäufer	
3	1	250	-37	Verkäufer	
4	3	300	089-99875	ReWe	

Gleichzeitig könnten beide Fremdschlüssel – PKURZ und KKURZ – gemeinsam den Primärschlüssel der Relation APA bilden. Sie sollten als Primärschlüssel allerdings – wie schon zuvor erwähnt – einen eigenen künstlichen Schlüssel definieren. Selbstverständlich muss trotzdem sichergestellt werden, dass die Schlüsselkombination PKURZ und KKURZ eindeutig bleibt. Dies lässt sich später mit einer UNIQUE-Schlüsseldefinition einfach realisieren.

Als Nächstes betrachten wir die Relation AUFTRAG:

AUFTRAG

AKURZ	APKURZ	...	Einheit	Anz	Satz	Netto	MwSt	Status
A18	3		STD	10	50	500	0.19	A

Das Attribut NETTO ist das Produkt aus ANZ *SATZ. Es ist somit nicht voll funktional abhängig vom Primärschlüssel, sondern vom Inhalt der genannten Attribute. Die Abhängigkeit der NICHT-Schlüsselfelder untereinander verstößt gegen die **dritte Normalform**, die besagt, dass alle NICHT-Schlüsselfelder von keinem anderen lokalen Attribut transitiv abhängig sein dürfen. Das heißt: Alle Attribute, die nicht zum Schlüssel gehören, müssen wechselseitig unabhängig voneinander sein. Das transitiv abhängige Attribut muss aus der Relation beseitigt werden. Das Attribut NETTO ist transitiv abhängig von einem algebraischen Ausdruck. Erlaubt die Datenbanksprache – also SQL – eine adäquate Formulierung, dann wird das abhängige Produkt als virtuell bezeichnet und aus dem logischen Datenbank-Entwurf entfernt. Analog verhält es sich natürlich mit dem Attribut NETTO in der Entitätsmenge LEISTUNG.

Lässt sich die transitive Abhängigkeit nicht mit einem Ausdruck beschreiben, wird das Attribut zusammen mit dem transitiv abhängigen Attribut in eine eigene Relation gestellt. Der Schlüssel der neuen Relation verbleibt als Fremdschlüssel in der alten Relation. Hier sollten wir uns einmal die Beziehung der PLZ zur Ortsbezeichnung in der Entitätsmenge KUNDE anschauen.

KUNDE

KKurz	Firma	...	Strasse	HausNr	PLZ	ORT	Umsatz
100	Versicherungspartner		Hauptweg	8			20000
250	Müller Werke GmbH		Korn-blumenweg	112	28219	Bremen	450000
500	Bremer Wohnungsbau		Korn-blumenweg	112	28213	Bremen	10000

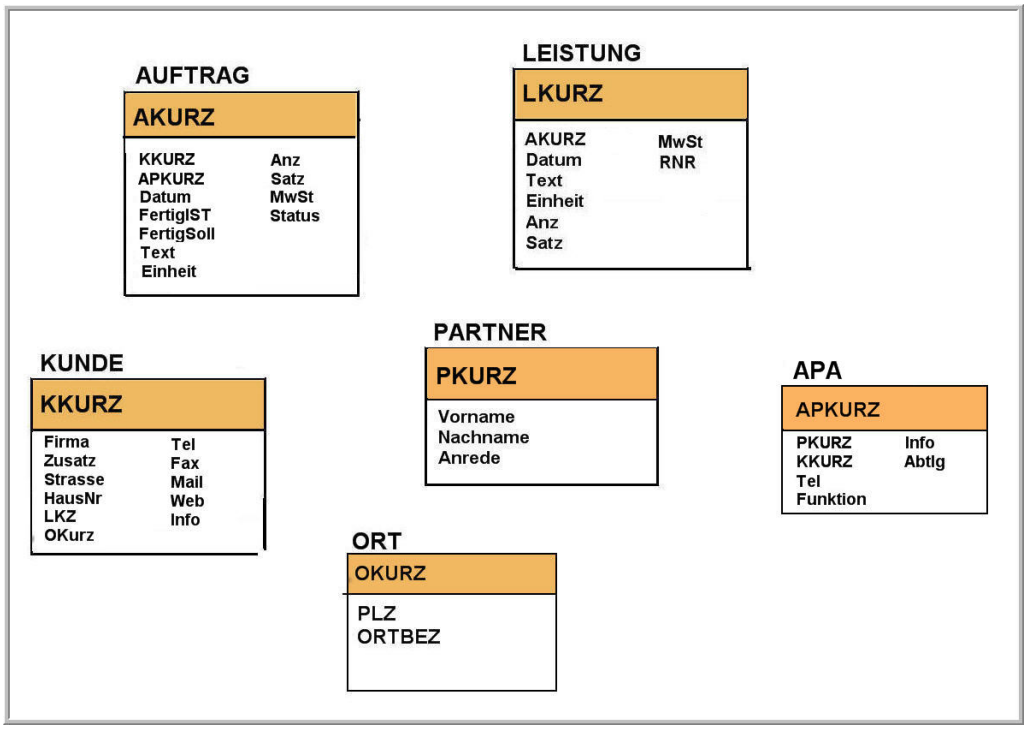
Bisher haben wir die PLZ und die Ortsbezeichnung gemeinsam gespeichert. Für den Bereich der Bundespost gilt, dass zu einer PLZ mehrere Orte gehören und einige Orte auch mehr als eine PLZ haben. Es besteht also eine komplexe Assoziation beziehungsweise haben wir es hier auch mit einer transitiven Abhängigkeit zu tun. So gibt es in „Bremen" den „Kornblumenweg" in zwei verschiedenen Stadtteilen. Die Stadtteile haben aber unterschiedliche Postleitzahlen. Nur durch die PLZ ist eine Unterscheidung möglich. Wir benötigen daher eine Beziehungsmenge. Für „überschaubare Datenbestände" verzichtet man häufig auf eine weitere Entitätsmenge. Nur wo fängt Überschaubarkeit an und wo hört sie auf?

Auch das Attribut UMSATZ in der Relation KUNDE sollten wir einmal hinterfragen. Das Attribut verstößt zunächst einmal nicht gegen die Regeln der Normalisierung, denn das Attribut ist nur einfach assoziiert mit dem Schlüssel KKURZ. Dennoch liegt in der Datenbank ein Problem vor! Der Umsatzwert wird prozedural über die Addition der in Rechnung gestellten Leistungen ermittelt. Diese Prozedur muss nach jeder Rechnung vollständig durchgeführt werden; sie darf nur genau einmal stattfinden. Bei Rechnungsänderungen müssen alle zugehörigen Kumulationen zunächst rückgängig gemacht werden. Programmabstürze, und menschliche Unaufmerksamkeit stellen die Richtigkeit der Werte zudem in Frage. Weil aber eine solche Zahl für betriebswirtschaftliche Auswertungen wichtig sein kann, muss aus Konsistenzgründen das Attribut in eine View verlagert werden.

Wir haben bereits viel über Schlüssel gesprochen, ohne uns über den Aufbau von Schlüsseln Gedanken gemacht zu haben. Oft trifft man auf Schlüsseldefinitionen, die selbst noch informativ sein sollen. So werden Abteilungsnummern in Personalnummern integriert oder Warengruppennummern in der Artikelnummer verborgen. Derartige „sprechende Schlüssel" sind häufig alphanumerisch und nur für kleine Entitätsmengen geeignet. Verwenden Sie lieber einfache, fortlaufende, numerische Schlüssel. Auf keinen Fall dürfen Ihre sprechenden Schlüssel inhaltlich auf andere Attribute der Entität verweisen, da sonst die zweite Normalform verletzt wird, auch wenn sich alles nur innerhalb eines Attributs abspielt. Der Beweis für meine Aussage ist einfach: Wenn Sie die Struktur Ihrer Warengruppen ändern, müssten auch alle Schlüsselwerte aktualisiert werden – oder aber die aufgebauten Bezüge geraten durcheinander.

Wechselt ein Mitarbeiter die Abteilung, müssten Sie die Personalnummer ändern!

Und nun wieder ein Blick auf die Fallstudie. Unsere Ausgangssituation hat sich aufgrund der Normalisierung bereits verändert.



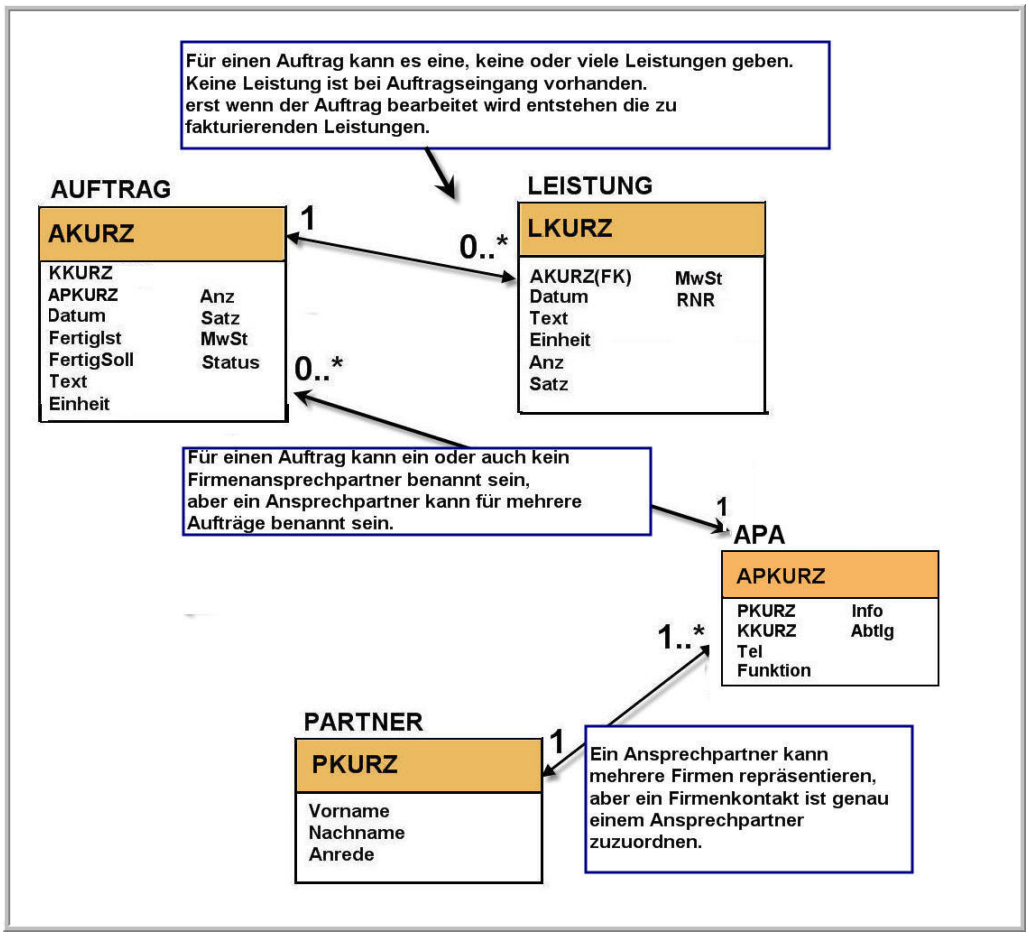
Das normalisierte Realitätsabbild

7.2.3.2.2 Die Relationenbeziehungen

Die Erstellung des konzeptionellen Schemas umfasst neben der Festlegung der Attribute auch die Darstellung der Beziehungen zwischen den Relationen. Die Beziehungen zwischen den Attributen einer Entität lassen sich in verschiedene Assoziationsformen beschreiben:

- Eine einfache Assoziation (TYP 1) liegt vor, wenn ein Wert der einen Menge zu genau einem Wert einer anderen Menge gehört.
- Von einer konditionellen Assoziation (TYP C) spricht man, wenn in der Verbindung zweier Mengen höchstens ein, also unter Umständen auch kein Wert in der anderen Menge vorhanden ist.
- Unter einer komplexen Assoziation (TYP M) ist zu verstehen, dass zu einem Wert kein oder beliebig viele Werte assoziiert werden können.

Beschreibt man die Assoziation vollständig, muss man auch die inversen Beziehungen berücksichtigen. Das heißt: Während zu einer Auftragsnummer mehrere Leistungen gehören, ist eine Leistung genau einem Auftrag zuzuordnen.



Vollständige Assoziationen am Beispiel: Auftrag / Leistung

Untersucht man die vollständigen Assoziationen, stellt man häufig komplexe Beziehungen (engl. many-to-many) fest. Solch eine komplexe Assoziation erhalten wir, wenn wir die Entitätsmenge KUNDE (KKURZ) und die Entitätsmenge PARTNER (PKURZ) in Beziehung setzen würden. Ein Kunde kann mehrere Ansprechpartner benennen, aber ein Ansprechpartner kann auch mehrere Firmen repräsentieren. Komplexe Assoziationen führen immer zu neuen Entitätsmengen. Das heißt: Der TYP M muss – wie bereits durch die Relation APA geschehen – mit einer neuen Beziehungsmenge aufgelöst werden.

Damit DBMS die Datenintegrität gewährleisten kann, müssen wir später Integritätsregeln in der Datenbank implementieren. Grundsätzlich lassen sich vier verschiedene Integritätsdefinitionen unterscheiden.

Für jeden Primärschlüssel benötigen wir eine **Entitäts-Integritätsbedingung**: Jede Relation hat genau einen Primärschlüssel. Die Werte solcher Primärschlüssel müssen eindeutig sein. Beispiel: KKURZ ist der Primärschlüssel der Relation KUNDE.

7.2.3
Seite 12

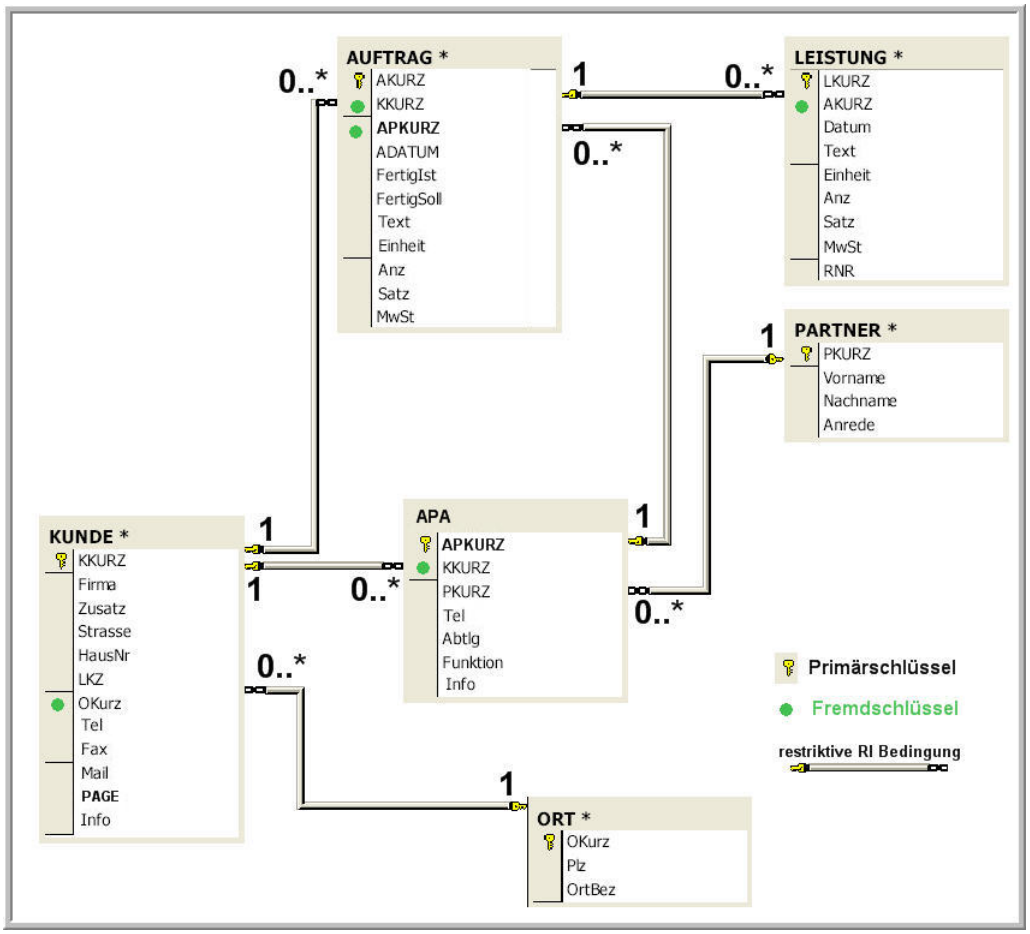
Für alle eindeutigen Attributkombinationen, die nicht Primärschlüssel sind, definieren wir **eindeutige Integritätsregeln**:
Jede Relation kann weitere Schlüsseldefinitionen enthalten, deren Werte eindeutig unterscheidbar sind. Für die Attributkombination PKURZ und KKURZ in der Relation APA benötigen wir solch eine eindeutige Integritätsdefinition.

Für jede Assoziation zwischen den Relationen benötigen wir **referentielle Integritätsbedingungen**:
In jedem Einzelfall ist festzulegen, was in der Datenbank passieren soll, wenn ein Primärschlüsselwert verändert oder gelöscht wird. Das heißt: Was soll beispielsweise passieren, wenn eine Auftragsnummer gelöscht wird, die in Beziehung zu Entitäten der Relation LEISTUNG steht? Mögliche Maßnahmen könnten sein:

- Änderungen oder Löschungen von Primärschlüsselwerten werden in der Entitätsmenge zurückgewiesen (restriktiv).
- Änderungen oder Löschungen von Primärschlüsselwerten werden akzeptiert und in die korrespondierenden Relationen stufenweise weitergegeben (kaskadierend).
- Löschungen von Primärschlüsselwerten werden akzeptiert und die korrespondierenden Werte erhalten den Wert NULL (SET NULL).

Hinzu kommen **benutzerdefinierte Integritätsbedingungen**, die es erlauben, Daten auf eine bestimmte Anzahl von Werten zu begrenzen.

Das Auftragseingangsdatum DATUM soll zum Beispiel vor dem Fertigstellungsdatum FERTIGIST liegen.



Die Beziehungen der Relationen zueinander

Sind alle Relationen der konzeptionellen Ebene in der dritten Normalform berücksichtigt, gilt das Realitätsabbild als voll normalisiert. Das heißt: In den Nicht-Schlüsseldomänen liegt keine Redundanz vor.

Unsere Datenbank ist ein Abbild der realen Welt und verhält sich wie diese.

7.2.3.2.3 ANSI-SPARC-3 Schichtenmodell

Durch die Normalisierung haben wir ein konzeptionelles Schema entwickelt, das keine redundanten Speicherungen erlaubt. Die Datenintegrität ist gewährleistet. Die Dreischema-Architektur, das -Konzept oder -Modell geht darüber hinaus.

Ziel des Dreischichtenmodells ist die Entkopplung des konzeptionellen Entwurfs von der physischen Speicherung, um die Unabhängigkeit der Anwendungsprogramme zu gewährleisten. Einfacher ausgedrückt: Was passiert mit unserem Datenbank-Modell, wenn wir für einen Kunden neben der Rechnungsadresse auch noch eine Lieferadresse benötigen oder zusätzliche Telefonnummern speichern wollen? Ganz einfach, wenn sich das Abbild der realen

Welt ändert, müssten Sie auch Ihre Datenbank verändern. Datenbank-Tabellen ändern, ist natürlich heute kein Problem mehr. Leicht lassen sich neue Spalten mit dem Befehl CHGPF (Physische Dateien ändern) oder mit der SQL-Anweisung ALTER TABLE ändern. Aber was passiert jetzt? Ihre bestehenden Programme passen nicht mehr zu den Tabellen. Natürlich gibt es die Möglichkeit, LVLCHK(*NO) zu verwenden oder alle Sourcen auf Verwendung der Tabelle zu durchzusuchen oder die Programme erneut zu kompilieren, und vielleicht haben Sie auch Change-Management-Software im Einsatz, die diese Arbeit für Sie übernimmt, aber dann doch externe Datenstrukturen mit Feldüberlappungen oder abhängige Datendefinitionen übersieht. Früher oder später verursacht die Veränderung Ihrer Tabellenstrukturen zusätzliche Arbeit.

Aus solchen Überlegungen heraus leitet sich die 3-Schichten Architektur einer Datenbank ab. Bereits 1975 wurde dieser DB-Architekturvorschlag veröffentlicht. Die 3-Ebenen-Architektur wurde als Strukturierungsmittel in der DB-Welt eingeführt, um die Sicht der Benutzer und der Applikationen auf die Daten von der Art der internen Organisation und Speicherung zu separieren. Einige Motive für die Trennung in drei unterschiedliche Ebenen sind:

- Verschiedene Benutzer haben verschiedene Sichten derselben Datenmenge. Die Art, wie ein Benutzer die Daten sieht, kann sich im Laufe der Zeit ändern.
- Der Benutzer soll von den Problemen der Datenbank-Strukturierung abgeschirmt werden.
- Der DB-Administrator sollte die logische Struktur der Datenbank reorganisieren können, ohne dass die Benutzer davon tangiert werden.
- Der DB-Administrator sollte die Speicherungsstruktur ändern können, ohne dass die logische Struktur davon beeinflusst wird.

Aus diesen Gründen definierte das ANSI-SPARC-Komitee die 3-Schema-Architektur für die Beschreibung der einzelnen Ebenen folgendermaßen:

Externe Sichten (engl. external schema)

Unterschiedliche Anwendungsprogramme beziehungsweise unterschiedliche Anwender benötigen im Normalfall nicht alle im konzeptionellen Modell entwickelten Objekttypen und -attribute, sondern nur einen anwendungsrelevanten Ausschnitt. Entsprechend soll jede Anwendung auch nur Zugang zu bestimmten Daten erhalten. Ausschnitte des konzeptionellen Modells werden Benutzersichten (engl. Views) genannt. Ein externes Schema ist somit eine abgegrenzte, anwendungs- und benutzerspezifische Sicht auf eine Datenbank, die jeweils genau an die individuellen Bedürfnisse der Benutzer oder Programme angepasst ist.

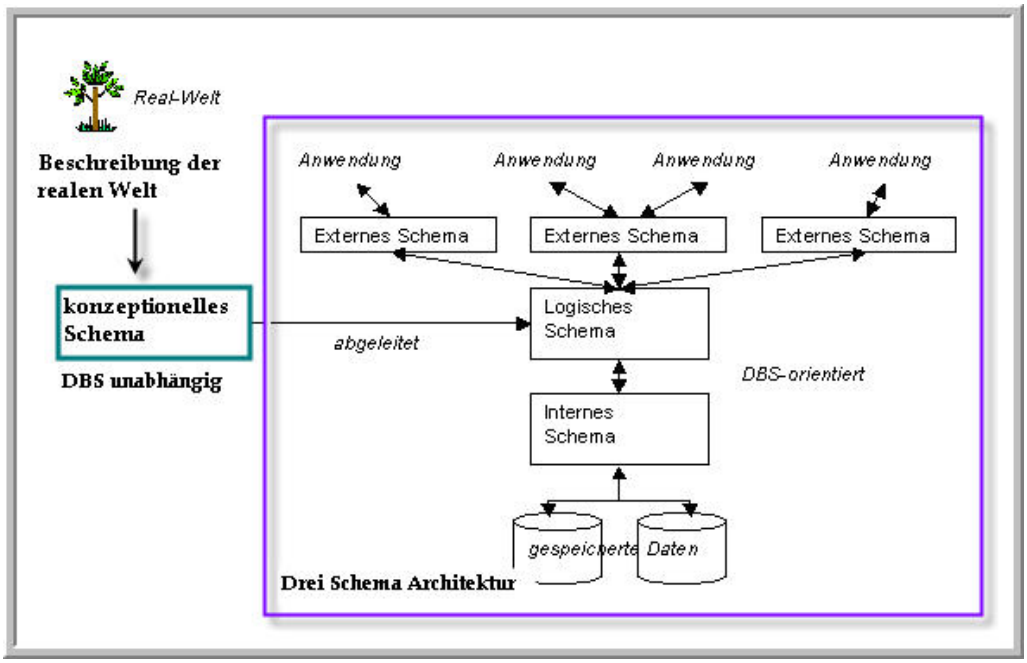
Konzeptionelle Sicht (conceptual schema)

Das konzeptionelle Schema ist das Ergebnis der Abbildung eines konzeptionellen Datenmodells in ein konkretes Datenmodell, das in einem bestimmten Datenbanksystem implementiert werden kann. Für die Erstellung des konzeptionellen Datenmodells kann man sich spezieller Datenmodellierungsmethoden wie beispielsweise des Entity-Relationship-Modells bedienen. Das konzeptionelle Modell ist unabhängig von der Realisierung in einem speziellen Datenbanksystem.

Das **logische Schema** wird je nach Begriffsdefinition mit dem konzeptionellen Schema gleichgesetzt. Häufiger aber kann es auch als Umsetzung des konzeptionellen in das interne Schema angesehen werden, was dann mittels einer konkreten Methode und einer konkreten Sprache beispielsweise mit SQL geschieht.

Interne Sicht (internal level)

Die interne Sicht bestimmt die physische Datenorganisation und legt die Zugriffspfade für die Daten fest. Die Zielsetzung ist hierbei eine minimale Zugriffszeit bei möglichst optimaler Speicherplatzausnutzung.



3-Schichten-Architektur

Während das interne Schema somit die physische Gruppierung der Daten und die Speicherplatzbelegung beschreibt sowie technologie- und produktorientiert ist, gibt das konzeptionelle Schema die sachlogischen Zusammenhänge der Daten wieder – es ist also systemneutral. Mehrere externe Schemata enthalten bestimmte Teilbereiche des konzeptionellen Schemas, die auf die jeweiligen Anwenderanforderungen zugeschnitten sind. Je nach Sichtweise verstehen wir unter Datenbeschreibung also etwas anderes.

Wenn die Unabhängigkeit der Ebenen real ist, lassen sich Änderungen in den Ebenen durchführen ohne Rückwirkungen auf die nächsthöheren oder tieferen Ebenen. Durch ein solch dreigliedriges Schema ist die physische und logische Datenunabhängigkeit gewährleistet.

Was bedeutet in diesem Zusammenhang **physische Unabhängigkeit**?
Das logische Schema kann unverändert bleiben, wenn sich beispielsweise aus Gründen der Optimierung oder Reorganisation der Speicherort oder die Speicherform einzelner Daten ändert. Von **logischer Unabhängigkeit** spricht man, wenn bei den meisten Änderungen des logischen Schemas die externen Schemata unverändert weiter bestehen. Dies ist besonders deshalb erwünscht, weil dadurch Anwendungsprogramme nicht modifiziert oder neu übersetzt werden müssen.

Soweit die Theorie. Was aber können wir praktisch tun, um unsere Datenbank abstrakter zu modellieren, d.h. wie kommen wir zu einer 3-Schichten-Architektur?

7.2.3.2.4 Schlüsseltabellen

Betrachten wir Spalten wie den Auftragsstatus oder die Anrede, stellen wir fest, dass – falls Änderungen vorgenommen werden – der gesamte Datenbestand durchsucht und die Inhalte der betreffenden Spalte auf den neuesten Stand gebracht werden müssen. So weit sind Ihnen die Dinge eigentlich bekannt und vermutlich auch die Lösung mit diversen Schlüsseltabellen. Wir könnten in AUFTRAG eine Auftragsstatusnummer als Fremdschlüssel einführen und definieren eine eigene Relation. Das Vorgehen ist einfach, nur welche Schlüsseltabellen werden benötigt und wie viele und welche Schlüsseltabellen existieren bereits? Hier besteht die Möglichkeit, der Schlüsselflut Herr zu werden.

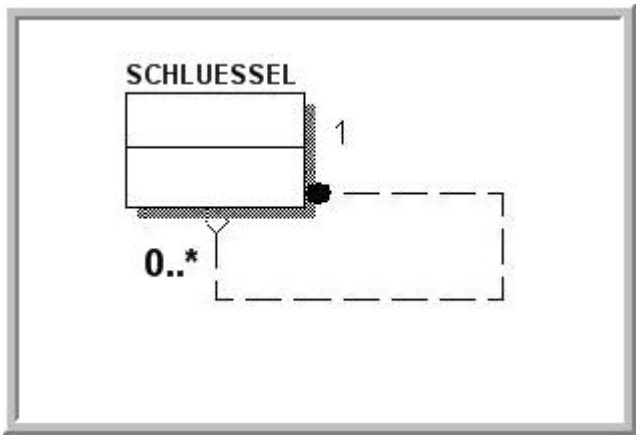
Schauen wir uns an, wie sich ganz allgemein ein Schlüssel darstellt. Erstens gibt es den Schlüsselbegriff an sich. Also die Anrede, den Auftragsstatus, das Länderkennzeichen und so fort. Zweitens stellt man fest, dass die Schlüssel natürlich bestimmte Werte annehmen. Zum Schlüsselbegriff ANREDE existieren die Werte „Frau“, „Herr“, „Firma“, der Schlüsselbegriff STATUS weist die Werte „Geplant“, „Vertrag“, „Erledigt“ und „Ablage“ auf. Wir bekommen somit zwei eigenständige Relationen. Es stellt sich die Frage, ob es möglich ist, in einer Relation alle Schlüsseltabellen zu vereinen ohne „sprechende Schlüssel“ in den Relationen zu verwenden. Die Lösung ist relativ einfach:

KURZ	BZKURZ	BZLANG
Status	GE	Geplant
Anrede	Frau	
Status	ER	Erledigt
LK	A	Argentinien
Status	VE	Vertrag
Anrede	Firma	
Status	AB	Ablage
LK	D	Deutschland
EH	Std.	Stunde

Wenn Sie auf diese Art und Weise mit Ihren Schlüsseltabellen verfahren, ist es nicht mehr zwingend notwendig, neue Schlüsseltabellen aufzubauen, wenn neue Schlüssel auftauchen.

Die Attribute KURZ und BZKURZ wären der Primärschlüssel der Relation. Allerdings haben wir zuvor alphanumerische Schlüssel für ungeeignet erklärt. Zu schnell könnte es auch hier aufgrund von Schreibfehlern zu redundanten Speicherungen kommen. Außerdem könnte es zukünftig Schlüssel geben, die untergeordnete Schlüssel enthalten. So kann der Schlüsselbegriff WARENGRUPPE unterteilt werden in NON-FOOD sowie FOOD und auch diese ließen sich weiter unterteilen. Es kann, muss aber nicht eigene Relationen innerhalb der Relation geben.

Falls dieser Fall eintritt, muss eine Projektion der Relation auf sich selbst möglich sein.



Schlüssel mit Unterschlüsseln

Aus diesen Gründen ist es besser, wir planen zunächst eine „Master-Schlüsselrelation“.

7.2.3
Seite 18

MASTERKEY

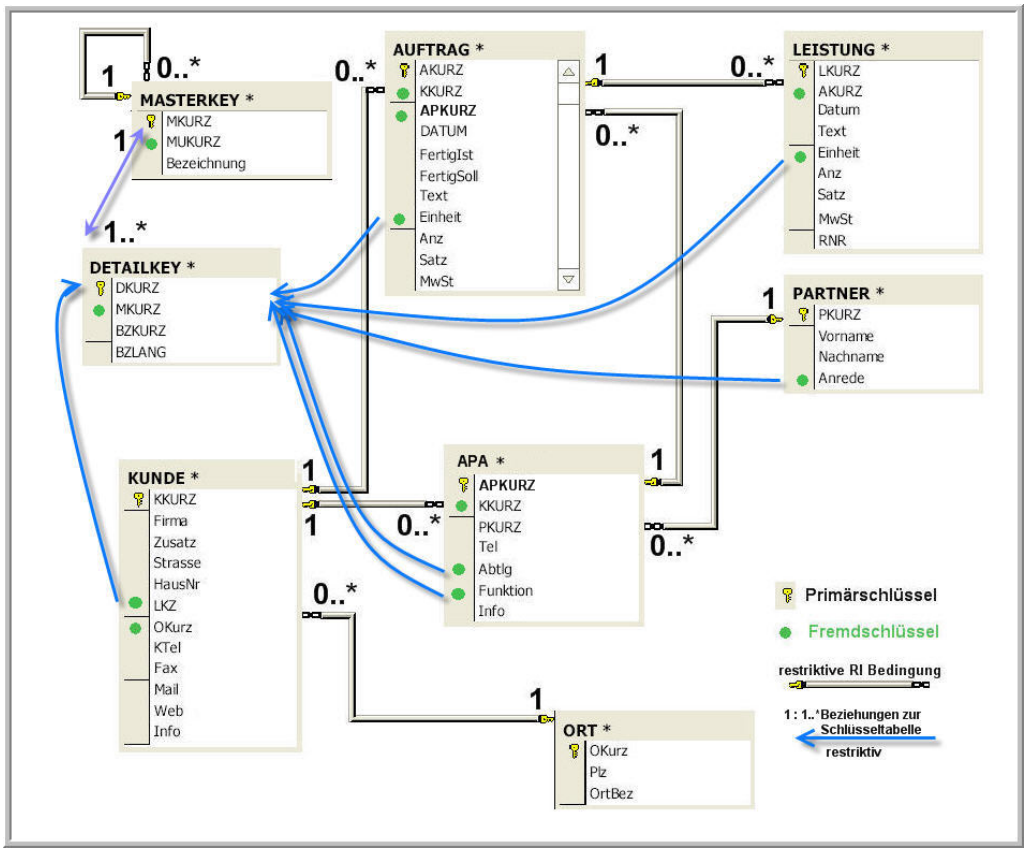
MKURZ	MUKURZ	Bezeichnung
100	0	Auftragsstatus
200	0	Anrede
300	0	Länderkennzeichen
400	0	Warengruppe
410	400	NON-Food
420	400	FOOD
430	410	Spielwaren

In der Relation MASTERKEY werden alle Schlüssel verzeichnet. Sie gibt jederzeit Aufschluss über die aktuell verwendeten Schlüsseltabellen. Das Attribut MUKURZ wird als Fremdschlüssel in der Relation aufgenommen, um jederzeit eine Hierarchisierung der Schlüsseltabellen zu ermöglichen. Die Inhalte der eigentlichen Schlüsseltabellen legen wir in eine zweite Relation, deren Entitäten mit der Relation MASTERKEY natürlich in Verbindung stehen müssen. Es ergibt sich über den Fremdschlüssel MKURZ in der Relation DETAILKEY eine komplexe Assoziation (1:N).

DETAILKEY

DKURZ	MKURZ	BZKURZ	BZLANG
1	100	GE	Geplant
2	200	Frau	
3	100	ER	Erledigt
4	300	A	Argentinien
5	100	VE	Vertrag
6	200	Firma	
7	100	AB	Ablage
8	300	D	Deutschland
10	600	Std	Stunde

Wie wirkt sich dieses Vorgehen auf unsere Fallstudie aus?



Datenbankbeschreibung mit Schlüsselrelationen

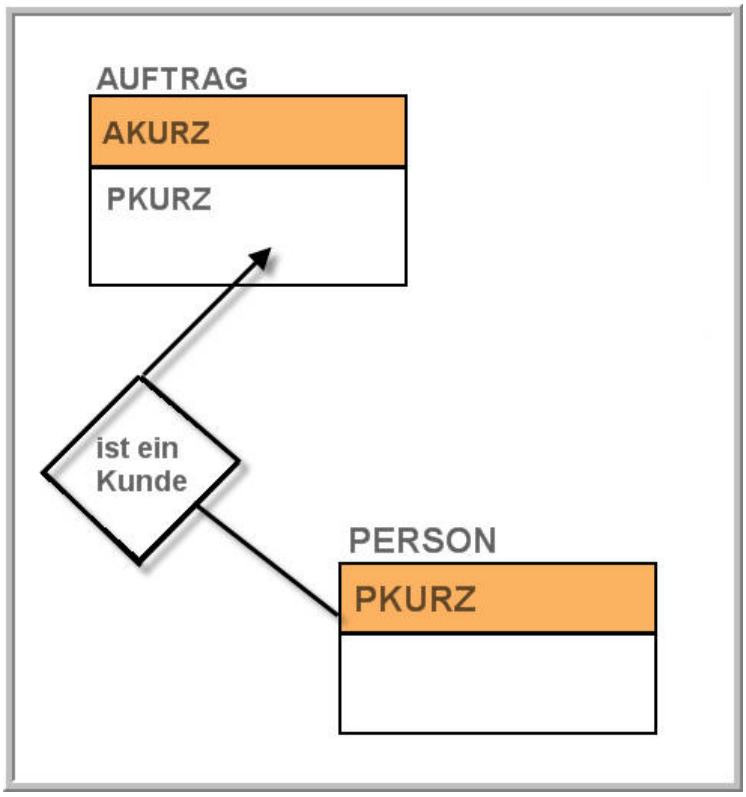
Sie sehen die Komplexität der Beziehungen nimmt deutlich zu. Auch wenn das Schaubild im ersten Augenblick verwirrt, werden Sie zu einem späteren Zeitpunkt feststellen, dass die Arbeit mit der Datenbank nicht kompliziert ist. Und um die Beziehungen der Relationen müssen Sie sich nicht kümmern. Dies übernimmt das DBMS für Sie. Letztendlich führt diese Vorgehensweise dazu, dass Sie später immer weniger in bestehende Datenbankstrukturen eingreifen oder ergänzen müssen. Je abstrakter Sie Ihre Datenbank modellieren, je stabiler ist ihr Aufbau. Neue abzubildende Schlüsseltabellen müssen jetzt nicht durch die Erstellung von Datenbank-Objekten gelöst, sondern lediglich durch ein Paket von Pflege beziehungsweise Konfigurationsprogrammen angelegt werden.

7.2.3.2.5 Rollenmodellierung

Vielleicht warten Sie schon an dieser Stelle auf den Aufbau der Tabellen, also die Definition der einzelnen Spalten. Aber noch sind wir nicht so weit.

Betrachten wir erneut unsere Kunden- und Partnerrelation. Es ist leicht nachvollziehbar, dass die Entitäten gewisse Gemeinsamkeiten besitzen. Wenn man im ersten Schritt einmal vernachlässigt, dass für die Entität KUNDE Firmen stehen können, so sieht man unmittelbar, dass sich hinter den Begriffen eigentlich Personen verbergen. Und damit kommen wir zum Begriff der Rolle. Wenn im Sprachgebrauch der Modellbeschreibung Sätze der Form „… ist ein(e) …“ auftauchen, so deutet das auf eine Rolle hin. Der Kunde **ist eine** Person. Dieser Satz ist natürlich nur ein Hinweis und seine Formulierung sollte auch recht flexibel ausgelegt werden. Ich zeige Ihnen anhand der Relation KUNDE, was gemeint ist. Vorausgesetzt es gibt keine Entität KUNDE sondern nur eine Entität PERSON. Aber was passiert dann mit dem Kunden, der die Aufträge erteilt?

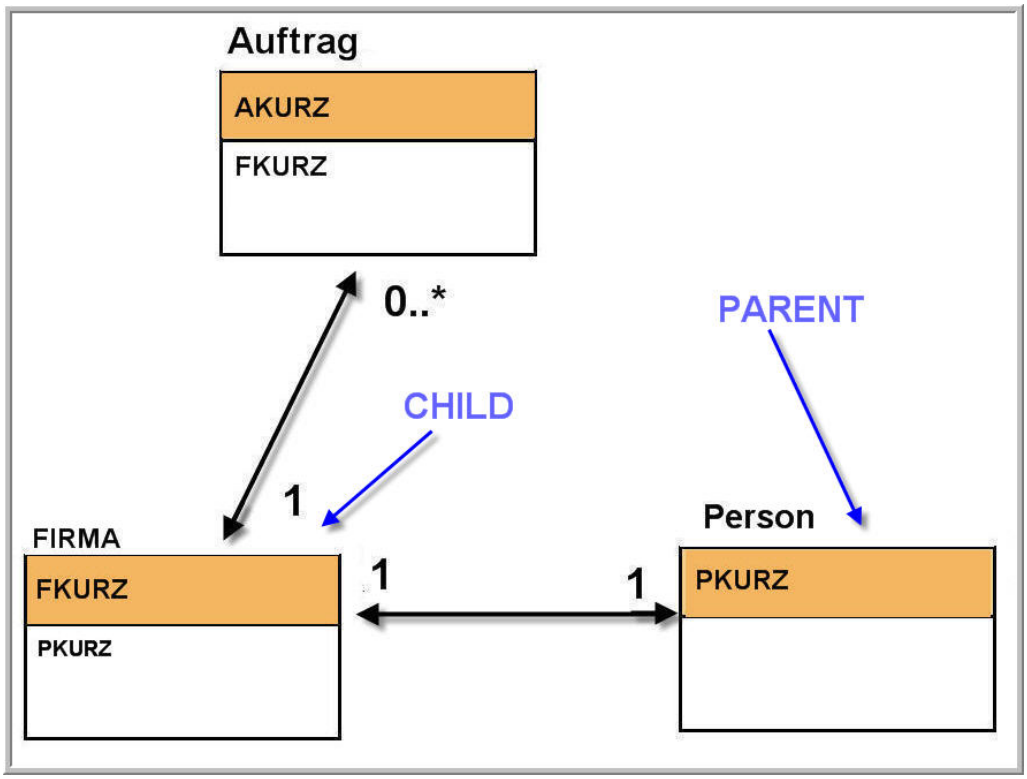
Nun, wenn der Kunde eigentlich eine Person ist, nur eben mit anderen oder zusätzlichen Attributen ausgestattet, so definieren Sie einfach, dass die Person in Bezug auf die Relation AUFTRAG die Rolle eines Kunden bzw. einer Firma spielt. Dieser Sachverhalt stellt sich dann im Diagramm wie folgt dar.



Die Person und ihre Rolle

Anstelle der Entität KUNDE steht die Entität PERSON. Eine Person spielt also in der Beziehung zum Auftrag die Rolle eines Kunden. Bis hierher handelt es sich lediglich um eine Änderung der Schreibweisen innerhalb des Diagramms. Es ist sozusagen eine andere logische Sicht der Dinge. Wie lässt sich dieser Sachverhalt der Realität in einem konzeptionellen Modell abbilden?

Stellen Sie zwischen die Relationen PERSON und AUFTRAG eine Rollenrelation. Ich nenne sie FIRMA. Es muss nun noch festgestellt werden, welche Kardinalitäten vorliegen. Wenn ich von einer Rolle FIRMA ausgehe, so sollte diese Rolle genau einer Person zugeordnet sein. Es ergibt sich eine 1:1-Beziehung.

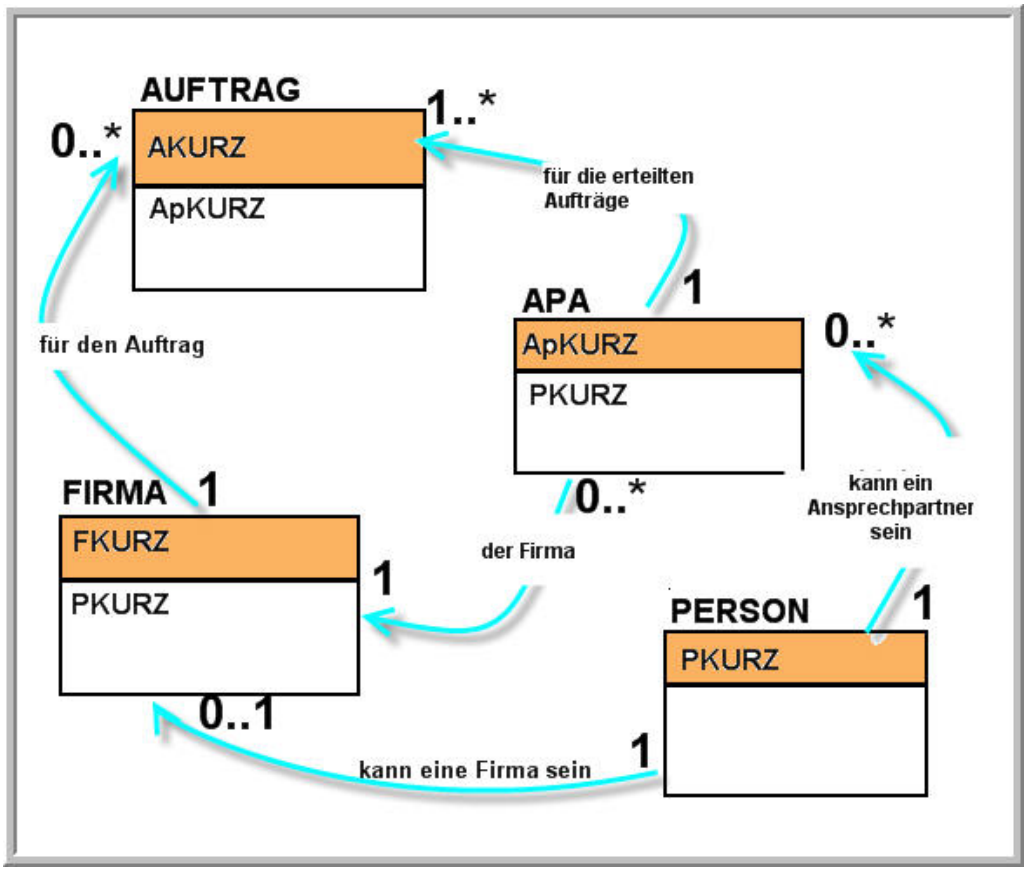


Die Rollenmodellierung

Jetzt muss geklärt werden, welche der beiden beteiligten Entitätsmengen die Parent-Funktion und welche die Child-Funktion erhält. Es geht also um die Frage, welche Tabelle übernimmt die Aufgabe der führenden Tabelle und welche ist die geführte Tabelle. Bei einer 1:1-Beziehung gibt es dafür keine Regel. Es bleibt Ihnen überlassen! Und dennoch sollten Sie die Relation FIRMA als Child-Relation sehen. Sie spielt die Rolle und enthält ergänzende Daten zur PERSON. Außerdem verhindern Sie durch diese Sichtweise, dass die Relation PERSON in ihrer Struktur geändert werden muss, wenn neue Rollen abgeleitet werden. Im anderen Fall müsste PERSON jedes mal um einen Fremdschlüssel für die neuen Relationen ergänzt werden. Das aber bedeutet Änderung in bestehenden Tabellen, die höchstwahrscheinlich schon Daten beinhalten.

7.2.3
Seite 22

Lassen Sie uns noch eine weitere Betrachtung zum Thema der Rollengestaltung machen. Wir haben unsere Ansprechpartner. Auch die Ansprechpartner sind letztendlich Personen. Wir haben also die Möglichkeit, auch hier den Rollengedanken umzusetzen und können vielleicht eine der bisherigen Entitäten ersetzen.



Die neue Rolle Ansprechpartner

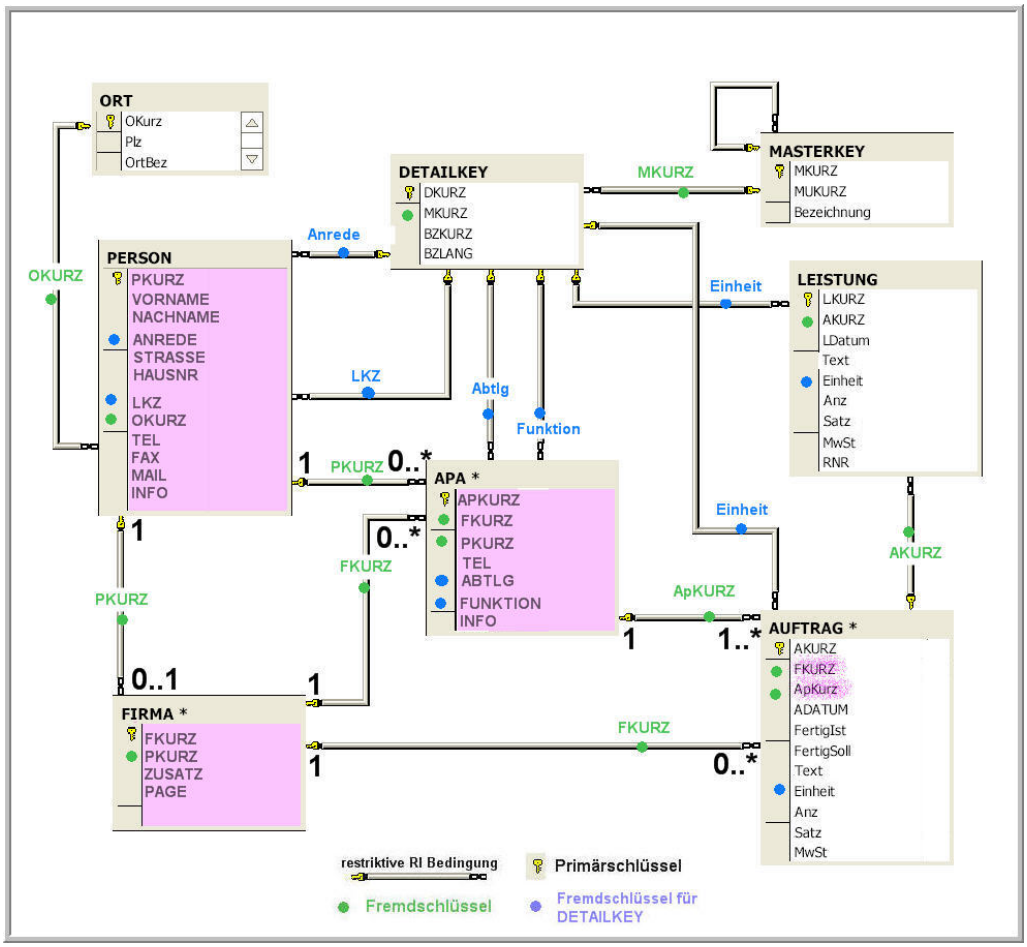
Im Grunde genommen hat sich wenig geändert. Nur benötigen wir unsere Entität PARTNER nicht mehr. Sie wurde durch die universelle Entität PERSON ersetzt, und die Relation APA bleibt als Rollenrelation erhalten.

Betrachten wir abschließend die Beziehungen der Relationen. Eine Firma kann einen, keinen oder viele Ansprechpartner haben. Ein Ansprechpartner lässt sich allerdings immer genau einer Firma zuordnen. Ähnlich verhält es sich mit den Aufträgen. Ein Ansprechpartner kann für einen oder viele Aufträge verantwortlich sein. Der Auftrag lässt sich allerdings genau einem Ansprechpartner zuordnen.

Beachten Sie, das sich die Kardinalität bei FIRMA von „1 zu 0..1“ geändert hat. In diesem Kontext würde die Kardinalität 1 bedeuten, dass zu jeder Person zwingend eine Rolle als FIRMA existieren muss! Hier liegt aber eine logische ODER-Verwandtschaft vor. Entweder die Person besitzt die Rolle FIRMA oder die Rolle APA. Auch die Variante, dass beide Rollen oder keine der beiden in Anspruch genommen werden, ist möglich.

Wo ist der Vorteil der Rollendefinitionen? Sollten sich die Ansprüche an die Datenbank im Laufe der Zeit ändern, indem zum Beispiel auch Mitarbeiter verwaltet werden, handelt es sich auch hier um Personen. Sie müssten lediglich Ihre Datenbank um die Rolle „Mitarbeiter“ ergänzen. Alle bestehenden Relationen bleiben unverändert. Falls Sie hingegen weitere Attribute für die Personen einpflegen wollen, wird nur die Relation PERSON erweitert. Die Änderungen wirken sich anschließend universell aus, ohne dass die einzelnen Rollenrelationen von den Änderungen betroffen sind.

Betrachten wir auch jetzt wieder unsere Gesamtsituation:



Der aktuelle Datenbankentwurf mit Rollen

Sie sehen, während wir zunächst glaubten, dass sich aufgrund unserer Normalisierung nur wenig ändert, stellen wir jetzt fest, dass sich die Struktur unserer Datenbank deutlich komplexer darstellt.

7.2.3.2.6 Spezifische Attribute

Ein letzter Blick gilt nun erneut den Attributen. Unsere Attribute haben wir aufgrund unserer Anforderungsdefinition sehr schnell zugeordnet. Diese Attribute beschreiben die Entität; Sie sollten darauf achten, dass sie wirklich spezifisch sind. Nehmen wir als Beispiel die Person. Es ist sicherlich spezifisch für eine Person, dass sie einen Namen, eine Anrede, eventuell einen Titel, Geschlecht und Körpergröße besitzt. Diese und weitere Attribute machen eine Person aus – sind kennzeichnend für sie. Im täglichen Gebrauch einer Datenbank wird im Zusammenhang von Personen auch die Rede von der Telefonnummer sein. Wenn einer Person nur eine Telefonnummer zugeordnet wird, kann man datenbanktechnisch diese Informationen wohl in einer entsprechend aufgebauten Personentabelle unterbringen. Aber eine Telefonnummer ist kein kennzeichnendes Merkmal wie das Alter oder der Geburtstag. Abgesehen davon wird man einer Person später eventuell auch mehrere Telefonnummern zuordnen müssen. Sie kennen das – TEL1, TEL2, TEL3…? Und wie sieht es mit der E-Mail-Adresse oder der Faxnummer aus? Was passiert? Sie müssen Ihre Tabelle erweitern und somit ändert sich auch Ihre Tabellenstruktur! Ähnlich verhält es sich mit den Adressdaten? Wie viele Adressen hat die Person? Vielleicht muss man irgendwann zwischen Wohn-, Rechnungs-, Lieferadresse unterscheiden?

Die Ausführungen zeigen, dass für eine semantische Validierung der Attribute eine Vielzahl von Überlegungen anzustellen ist. Schematische Datenbank-Normalisierungen reichen nicht aus, obwohl die Normalformen bei originären Daten beachtet werden müssen. Als wirksamste Prüffrage gegen semantische Fehler hat sich folgende Frage heraus kristallisiert:

Ist das Attribut x wirklich eine Eigenschaft von Entitätstyp E oder etwa Eigenschaft eines anderen Entitätstyps?

Aufgrund der erneuten semantischen Validierung unserer Attribute erhalten wir weitere eigenständige Entitäten. Beginnen wir mit den Kommunikationsdaten. Kommunikationsdaten sind nicht kennzeichnend für die Person als solches. Sie sind ihr lediglich für einen Zeitraum zugeordnet, können sich aber jederzeit verändern. Bilden wir diesen Sachverhalt ab, erhalten wir eine eigenständige Relation KOMMUNIKATION.

KOMMUNIKATION

KTab	KKEY	Medium	Nummer

Die Entität erhält wie alle anderen Relationen zunächst einen künstlichen Primärschlüssel, das Attribut KOKURZ. KKEY wird als Fremdschlüssel benötigt, um zu den Entitätsmengen FIRMA und APA eine beliebige Anzahl Beziehungen aufbauen zu können. Das Attribut Mediumdient zur Unterscheidung der Kommunikationsdaten in Telefon, Fax, Email etc. Hier handelt es sich natürlich wieder um eine Schlüsseltabelle, die wir mit unserer Entität DETAILKEY problemlos abbilden können.

Ebenso verhält es sich mit den Zusatzinformationen, die sowohl für die Firmen als auch für deren Ansprechpartner hinterlegt werden. Auch hierbei handelt es sich nicht um spezifische Eigenschaften der Entität FIRMA oder APA. Wenn es sich hier um freie Texte handelt, benötigen diese Informationen Platz, aber wie viel Text ist notwendig und werden tatsächlich für jede Firma und jeden Ansprechpartner Zusatzinformationen festgehalten? Schon allein um Plattenplatz zu sparen, sollten wir auch an dieser Stelle eine eigene Entität bilden.

ZUSATZINFO

ZKURZ	ZTab	ZKEY	Info

Aufgrund des künstlichen Primärschlüssels ist es mir jetzt sogar möglich, für eine Firma mehrere Informationen festzuhalten – falls der Platz doch einmal nicht ausreichend sein sollte. Unsere Entitäten FIRMA und APA ändern sich natürlich entsprechend in unserem Datenbankdesign.

FIRMA

FKurz	Zusatz	Page

APA

APKURZ	PKURZ	KKURZ	Abtlg
1	1	100	100
2	2	100	150
3	3	100	200

Stellen Sie sich jetzt einmal vor, Sie möchten zukünftig auch Zusatzinformationen zu Aufträgen ablegen. Was ist zu tun? Die Antwort lautet: Sie müssen gar nichts an der Datenbank ändern. Sie speichern lediglich die Auftragsnummer im Fremdschlüssel ZKEY und legen die gewünschten Informationen ab.

7.2.3
Seite 26

Schauen wir uns auch abschließend die Adressdaten an, die zurzeit noch in der Entität PERSON gespeichert sind.

PERSON

PKurz	Vorname	Nachname	...	OKURZ	Strasse	HausNr
100				1	Am Heuberg	7
120				2	Waldweg	123

Auch die Adressdaten sind wie bereits erwähnt keine kennzeichnenden Daten der Person.

PERSON

PKURZ	Vorname	Nachname	Anrede

Anschließend bauen wir die Entität ADRESSE wieder auf:

ADRESSE

ADKURZ	ATab	AKEY	AdrArt	OKURZ	Strasse	HausNr

Das Attribut ART dient lediglich der Unterscheidbarkeit der Adressen in Lieferadresse(L), Rechnungsadresse(R) und Privatadresse(P). Natürlich könnten Sie auch hier wieder auf unsere Schlüsseltabelle Bezug nehmen. Anschließend kann ich für die Firmen beliebig viele Adressen hinterlegen und auch in Bezug zum Ansprechpartner ist die Welt wieder in Ordnung. Für die Ansprechpartner sollen lt. Anforderungsdefinition zurzeit keine Adressen hinterlegt werden. Wir hätten also die Adressfelder der Relation PERSON für die Ansprechpartner nicht genutzt. Jetzt können wir jederzeit – falls gewünscht – Adressen der Ansprechpartner vermerken, wir müssen es aber nicht! Ihre Datenbank ist von diesen Veränderungen nicht betroffen.
Die letzte Überprüfung gilt den Entitäten AUFTRAG und LEISTUNG. Beide Entitäten enthalten Preis- bzw. Kalkulationsdaten. Auch hier drängt sich der Verdacht auf die Informationen als eigenständige Entität PREIS abzubilden.

PREIS

PRKURZ	PTab	PKEY	KalkArt	Einheit	Anz	Satz	MwSt

Warum? In Bezug auf die Entität Leistung übernimmt die Entität PREIS die Rolle der Abrechnungsdaten. Die gleichen Informationen benötigt die Entität AUFTRAG für die Auftragseingangskalkulation. Diese ursprünglichen Kalkulationsdaten müssen aber bei Auftragsfertigstellung keinesfalls dem tatsächlichen Auftragswert entsprechen. Außerdem könnte es zukünftig erforderlich sein, für einen Auftrag unterschiedliche Kalkulationen zu erstellen. Spätestens dann wären wir gezwungen die Kalkulationsdaten in eine eigenständige Relation zu stellen, um nicht gegen die Normalformen zu verstoßen.

So ergibt sich das folgende Realitätsabbild für die Entitäten AUFTRAG und LEISTUNG:

AUFTRAG

AKURZ	FKURZ	APKURZ	Datum	FertigIST	FertigSOLL	Text

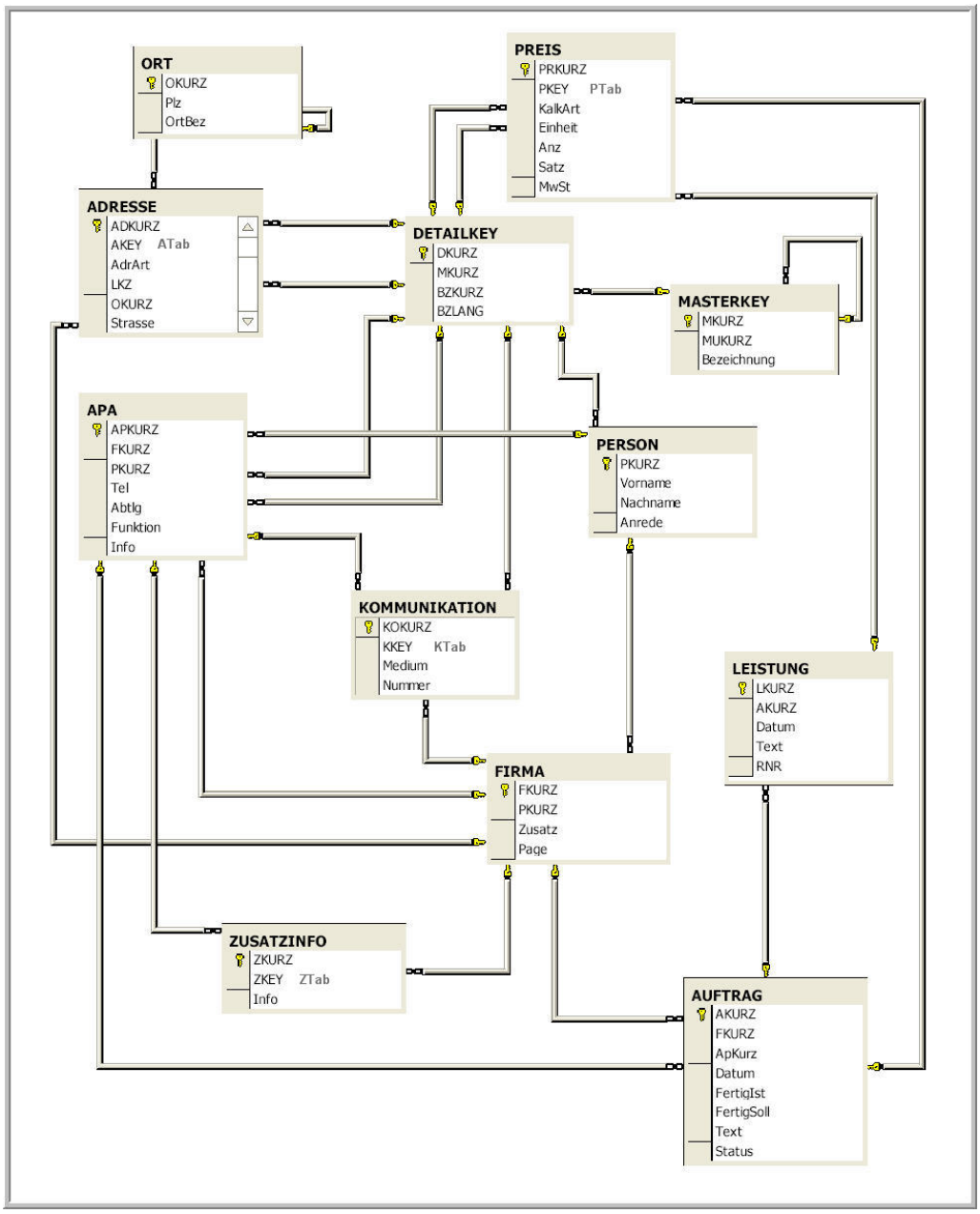
LEISTUNG

LKURZ	AKURZ	Datum	Text	RNR

Wir sind fertig mit dem konzeptionellen Entwurf und so sieht unser Abbild der Realität jetzt aus:

Das Ergebnis ist erstaunlich! Die Struktur unseres konzeptionellen Entwurfs hat mit dem ersten Entwurf nur noch wenig gemeinsam und dennoch spiegelt die Datenbank immer noch unseren Realitätsausschnitt wieder, ist aber im Hinblick auf zukünftige Änderungen weitestgehend abstrakt modelliert.

7.2.3
Seite 28



Der fertige konzeptionelle Entwurf

Diese Art des Modellentwurfs erscheint zunächst recht aufwendig. Auch der Zugriff auf die Tabellen ist nicht besonders reizvoll. Und natürlich stellt sich die Frage, ob diese Datenbank „schnell“ genug ist. Diese Fragen werden sich im weiteren Verlauf sicherlich beantworten lassen, denn wir bauen eine komplette Anwendung auf und betrachten in diesem Zusammenhang selbstverständlich auch die Performance der Datenbank und nicht nur mit ein oder zwei Tabellenzeilen, sondern mit einer Datenbasis, die ebenfalls ein Realitätsabbild sein könnte. Lassen Sie sich überraschen! Für den externen Zugriff auf die Datenbank, d.h. für alle Anwendungs- und Benutzersichten lassen sich später jeder-

zeit komplexe Views modellieren, die den Datenzugriff erheblich vereinfachen und um die Pflege der Datenbank mit den definierten Beziehungen kümmert sich schließlich die DB2 UDB for iSeries.

Die Erfahrung hat gerade gezeigt, dass sich der Designaufwand und die Abstraktion zu Beginn lohnen, da die Risiken und die Arbeit bei späteren Erweiterungen und Änderungen ganz erheblich reduziert werden können. Es wird nicht gelingen diesen Spätaufwand auf Null zu senken, aber eine Erleichterung in hohem Maß ist realistisch!

7.2.3
Seite 30

7.2.4 Das physische Datenbankkonzept

Ich gehe davon aus, dass Ihnen SQL-Datentypen und SQL-Namenskonventionen bekannt sind. Andernfalls finden Sie eine detaillierte Beschreibung in den nachfolgenden Kapiteln.

Hier soll es darum gehen, den konzeptionellen Entwurf zunächst in ein logisches Schema umzusetzen, um danach dieses Konzept mit SQL als Datenbank zu implementieren.

Ausgangspunkt ist unser konzeptioneller Datenbankentwurf. Schauen wir ihn uns also nochmals fertig in Kapitel 7.2.3, Seite 28 an. Dort finden wir unsere Entitäten. Ihnen ordnen wir in einem ersten Schritt Tabellen- und Systemnamen zu. Auch einen beschreibenden Text, der unter anderem der DB-Dokumentation dienen soll, dürfen wir nicht vergessen.

Entität	Tabellenname (SQL und i5/OS) physisches Konzept	Beschreibung
ADRESSE		Adressen der Kunden und Ansprechpartner
AUFTRAG	ORDERHDR	Auftragskopfdaten
APA		Ansprechpartner für Kundenaufträge
DETAILKEY	DETAIL	Universelle Schlüsseltabelle
FIRMA		Kundenstammdatendaten
KOMMUNIKATION	KONTAKT	Telefon- und Fax-Nummern, e-Mail-Adressen der Ansprechpartner und Kunden
LEISTUNG	ORDERPOS	Auftragspositionen (=Leistungen)
MASTERKEY	MASTER	Schlüsseltabellenübersicht
ORT		Ortsnamen und dazugehörige Postleitzahlen
PERSON		Spezifische Personenkennzeichen
PREIS	WERT	Kalkulations- und Abrechnungsdaten der einzelnen Auftragspositionen
ZUSATZINFO	ZINFO	Enthält frei formulierte Info-Texte für Kunden und Ansprechpartner

Sie sehen, ich lege im ersten Schritt die Tabellennamen fest. Bei einigen Tabellen entscheide ich mich dafür, die Namen zu verkürzen, um keine abweichenden Systemnamen zu erhalten und um mir späteren Schreibaufwand zu ersparen.

In einem zweiten Schritt ordne ich den Tabellen die Attribute zu. Auch hier prüfe ich die Länge der Spaltennamen. Zusätzlich entscheide ich mich für zusätzliche Spalten, um den Zeitpunkt der Datenerfassung und der -änderung zu protokollieren.

Attribute der Entität	SQL Name	I5/OS Name	Datentyp	Beschreibung
ADRESSE (Entität Adresse)				
ADKURZ			INTEGER	Künstlicher **Primärschlüssel**
ATAB	Wird aufgrund des SEQUENZ-Objekts SEQ_ADR nicht benötigt.			
AKEY			INTEGER	**Fremdschlüssel** für die Parent-Tabellen FIRMA + APA
ADRART	AdressArt	ADRART	INTEGER	Rechnungs-, Lieferadresse etc., **Fremdschlüssel** für die Parent-Tabelle DETAIL
LKZ				Länderkennzeichen, **Fremdschlüssel** für die Parent-Tabelle DETAIL
OKURZ			INTEGER	PLZ + Ortsname, **Fremdschlüssel** für die Parent-Tabelle ORT
STRASSE				Straßenname
HAUSNR				Hausnummer
	ADRINSERTDATE	ADRINSDAT	Zeitstempel	Eingefügt am
	ADRINSERTUSER	ADRINSUSR	CHAR(10)	Eingefügt von
	ADRUPDATEDATE	ADRUPDDAT	Zeitstempel	Verändert am
	ADRUPDATEUSER	ADRUPDUSR	CHAR(10)	Verändert von
ORT(Entität ORT)				
OKURZ			INTEGER	Künstlicher **Primärschlüssel**
PLZ				Postleitzahl
ORTBEZ	ORTSNAME	ORTNAM		Ortsname
	ORTINSERTDATE	ORTINSDAT	Zeitstempel	Eingefügt am
	ORTINSERTUSER	ORTINSUSR	CHAR(10)	Eingefügt von
	ORTUPDATEUSER	ORTUPDDAT	Zeitstempel	Verändert am
	ORTUPDATEDATE	ORTUPDUSR	CHAR(10)	Verändert von

APA(Entität APA)				
APKURZ			INTEGER	Künstlicher **Primärschlüssel**
FKURZ			INTEGER	Kundenummer, **Fremdschlüssel** für die Parent-Tabelle FIRMA
PKURZ			INTEGER	Personendaten (Vorname, Nachname), **Fremdschlüssel** für die Parent-Tabelle PERSON
FKURZ + PKURZ müssen eindeutig sein.				
ABTLG			INTEGER	Abteilungsname, **Fremdschlüssel** für die Parent-Tabelle DETAIL
FUNKTION			INTEGER	Bezeichnung für die Funktion innerhalb der Firma (Kunde); **Fremdschlüssel** für die Parent-Tabelle DETAIL
INFO			INTEGER	Beliebige Zusatzinformationen, **Fremdschlüssel** für die Parent-Tabelle, ZINFO
	APAINSERTDATE	APAINSDAT	Zeitstempel	Eingefügt am
	APAINSERTUSER	APAINSUSR	CHAR(10)	Eingefügt von
	APAUPDATEDATE	APAUPDDAT	Zeitstempel	Verändert am
	APAUPDATEUSER	APAUPDUSR	CHAR(10)	Verändert von
HDROrder(Entität AUFTRAG)				
AKURZ			INTEGER	Künstlicher **Primärschlüssel**
FKURZ			INTEGER	Kundenummer **Fremdschlüssel** für die Parent-Tabelle FIRMA
APKURZ			INTEGER	Ansprechpartner **Fremdschlüssel** für die Parent-Tabelle APA
DATUM	EingangsDATUM	EINDAT	Datumswert	Datum des Auftrageingangs
FERTIGSOLL	SollDatum	SOLLDAT	Datumswert	Geplante Fertigstellung
FERTIGIST	ISTDATUM	ISTDAT	Datumswert	Tatsächliche Fertigstellung
TEXT				Globaler Auftragstext
STATUS			INTEGER	Auftragsstatus **Fremdschlüssel** für die Parent-Tabelle DETAIL
	HDRORDERINSERTDATE	HDRINSDAT	Zeitstempel	Eingefügt am
	HDRORDERINSERTUSER	HDRINSUSR	CHAR(10)	Eingefügt von
	HDRORDERUPDATEDATE	HDRUPDDAT	Zeitstempel	Verändert am
	HDRORDERUPDATEUSER	HDRUPDUSR	CHAR(10)	Verändert von

7.2.4
Seite 4

Sie können bereits zum jetzigen Zeitpunkt erkennen, dass jede Tabelle einen Primärschlüssel und eine Vielzahl an Fremdschlüsseln enthält. In der Entwurfsphase haben wir darüber gesprochen, dass es vorteilhaft ist, künstliche Primärschlüssel zu bilden. Wenn wir uns darauf einigen, die Primärschlüssel als Integer-Wert zu definieren, legen wir automatisch auch den Datentyp für die zugehörigen Fremdschlüssel fest, da die Fremdschlüssel in Länge und Typ dem Primärschlüssel entsprechen müssen.

Neben den Primär- und Fremdschlüsselspalten gibt es klassische Zeichenspalten – so zum Beispiel die Attribute „Vorname“, „Nachname“ und „Strasse“. Auch hier lässt sich eine globale Definition finden: Ich lege fest, dass derartige Namensfelder als variable Zeichenfelder der Länge 150 definiert werden sollen, wobei ich aus Erfahrung weiß, dass die meisten Bezeichnungen nicht mehr als 50 Zeichen benötigen. Also teile ich 50 Byte als feste Spaltengröße zu. So kann ich sicherstellen, dass nur wenige Daten im Overflow-Bereich der Tabelle gespeichert werden, was den Zugriff auf die Daten deutlich beschleunigt. Diesen Datentyp verwende ich auch für die Spalten „Nummer“ und „Zusatz“. Vielleicht werden Sie sich über meine Entscheidung für die Spalte „Nummer“ wundern. Laut konzeptionellem Entwurf dürfen in dieser Spalte neben „Telefon- und Faxnummern“ auch „e-Mail-Adressen“ gespeichert werden, also brauche ich ein Zeichenfeld; 150 Zeichen müssten ausreichend sein.

Für kurze Zeichenspalten lege ich einen Datentyp CHAR 10 fest. Die i5/OS-Benutzernamen, die ich bei INSERT und UPDATE zur Dokumentation speichern möchte, sind i5/OS-Objektnamen, deren Länge nicht wesentlich variieren kann. „Postleitzahl“, „Hausnummer“ und ähnliche Werte sollen immer in diesem Datentyp untergebracht werden.

Unsere Datenbank hat allerdings weitere Textfelder. In der Tabelle ZUSATZINFO existiert die Spalte INFO. Hier sollen freie Texte gespeichert werden. Wie lang diese Texte sein werden, ist derzeit noch unklar. Ich entscheide mich, die Spalte als Character-Large-Objekt mit 1 MB zu definieren, werde aber im Satz 2.000 Zeichen fest zuordnen – in der Annahme, dass viele Kurzinfos nicht mehr als 2.000 Zeichen benötigen.

Außerdem gibt es die typischen Datumsfelder: „Auftragseingangsdatum“, „Fertigstellungsdatum“ etc. Hier verwenden wir natürlich den Datentyp DATE.

Die Zeitstempel, die den Zeitpunkt von UPDATE und INSERT dokumentieren sollen, werde ich als Timestamp-Spalten deklarieren, die später mit dem jeweils aktuellen Zeitwert initialisiert werden.

Zu guter Letzt gibt es dann noch die EURO-Beträge: „Stundensätze“, „Positionswerte“ usw. Auch hier definiere ich einen globalen Datentyp: DEC(11,2).

Zusätzlich benötigen wir noch Ganze Zahlen, um die Anzahl der geleisteten Stunden oder Tage abzubilden. Dafür sollte der Datentyp SMALLINT ausreichend sein.

Schauen sie sich jetzt noch mal den Entwurf der Tabelle „Firma“ an. Dort finden Sie die Spalte PAGE. Diese Spalte enthält die URL der entsprechenden Homepage. Hier bietet es sich also an, den Datentyp DATALINK zu verwenden.

Wenn ich mich an die oben beschriebenen Konventionen halte, würde ich folgende Referenzdatei erhalten – vorausgesetzt, dass ich meine Datenbank mit DDS beschreibe:

Spaltenverwendung	SQL-Datentyp	Erläuterung
Datumsspalte	DATE	Einfache Datumswerte: Eingangsdatum, Soll-Datum, Ist-Datum
i5/OS-Objektnamen Kleine Zeichenfelder	CHAR(10)	Benutzerprofilnamen, PLZ, Hausnummer NOT NULL
Zeichenfelder	CHAR(150) Allocate(50)	Klassische Zeichendaten – wie Vorname, Nachname, Straßenname etc. NOT NULL
Große Zeichenfelder	CLOB 1mb Allocate(10000)	NOT NULL
Zeitstempel	Timestamp	Zeitpunkt des Insert oder Update
Euro-Betrag	Dezimal 11,2	EUR-Spalten (z. B. Positionspreis) NULL erlauben (wenn nichts kalkuliert wurde, dann ist die Spalte NULL)
Ganze Zahl	SMALLINT	Kleine ganze Zahlen für die Anzahl der geleisteten Einheiten usw.
Prim_Sequenz	Integer	Primärschlüssel mit späterem SEQUENZ-Objekt Default NOT NULL
Prim_AS_Identity	Integer	Von 10 bis 2.147.483.647 Erhöhung um 10
Fremdschlüssel	Integer	Jeder Fremdschlüssel NOT NULL
DataLink	DataLink	Typ URL, NO Control

Und so sieht dann das vollständige logische Konzept aus:

Tabellenname	SQL Spaltenname	I5/OS Spaltenname	Datentyp	Schlüssel/ Bedingung	Standardwert
ADRESSE					
	ADKURZ	ADKURZ	Prim_AS_Identity	PK	Fortlaufende eindeutige Nummer AS IDENTITY
	Identische Schlüssel in den Tabellen Firma und APA werden durch den Einsatz eines SEQUENZ-Objekts verhindert. Die Summe der Primärschlüssel der Tabellen FIRMA +APA können somit keine doppelten Werte enthalten! Die Spalte ATAB zur Identifikation der Tabelle wird nicht benötigt.				
	AKEY	AKEY	Fremdschlüssel	FK/RI (Firma(FiKURZ) APA(APKURZ)	Default NOT NULL
	AdressArt	ADRART	Fremdschlüssel	FK/RI (DETAIL(DeKURZ))	Default NOT NULL
	LandesKennzeichen	LKZ	Fremdschlüssel	FK/RI (DETAIL(DeKURZ))	Default NOT NULL
	ORKURZ	ORKURZ	Fremdschlüssel	FK/RI (ORT(OrKURZ))	Default NOT NULL
	STRASSE	STRASSE	Zeichenspalte		
	HausNr	Hausnr	Kurze Zeichenspalte		
	ADRESSEINSERTDATE	ADRINSDAT	Zeitstempel		Default: Aktueller Timestamp
	ADRESSEINSERTUSER	ADRINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	ADRESSEUPDATEDATE	ADRUPDDAT	Zeitstempel		Default: Aktueller Timestamp
	ADRESSEUPDATEUSER	ADRUPDUSR	Kurze Zeichenspalte		Benutzerprofilname
ORT					
	ORKURZ	ORKURZ	Prim_AS_Identity	PK	Fortlaufende eindeutige Nummer AS IDENTITY
	PLZ	PLZ	Kurze Zeichenspalte		
	ORTSNAME	ONAME	Zeichenspalte		
	ORTINSERTDATE	ORTINSDAT	Zeitstempel		Default: Aktueller Timestamp
	ORTINSERTUSER	ORTINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	ORTUPDATEDATE	ORTUPDDAT	Zeitstempel		Default: Aktueller Timestamp
	ORTUPDATEUSER	ORTUPDUSR	Kurze Zeichenspalte		Benutzerprofilname

APA					
	APKURZ	APKURZ	Prim_Sequenz	PK	Fortlaufende eindeutige Nummer durch SEQUENZ Objekt FIR_APA
	FIKURZ	FIKURZ	Fremdschlüssel	FK/RI(Firma/FiKURZ)	Default NOT NULL
	PEKURZ	PEKURZ	Fremdschlüssel	FK/RI(Person/PeKURZ)	Default NOT NULL
	ABTLG	ABTLG	Fremdschlüssel	FK/RI(Detail/DeKURZ)	Default NOT NULL
	FUNKTION	FUNKTION	Fremdschlüssel	FK/RI(Detail DeKURZ)	Default NOT NULL
	APAINSERTDATE	APAINSDAT	Zeitstempel		Default: Aktueller Timestamp
	APAINSERTUSER	APAINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	APAUPDATEDATE	APAUPDDAT	Zeitstempel		Default: Aktueller Timestamp
	APAUPDATEUSER	APAUPDUSR	Kurze Zeichenspalte		Benutzerprofilname
ZINFO (Entität ZusatzInfo)					
	ZIKURZ	ZIKURZ	Prim_AS_Identity	PK	Fortlaufende eindeutige Nummer AS IDENTITY
	ZKEY	ZKEY	Fremdschlüssel	FK/RI Firma(FiKURZ) FK/RI Apa(APKURZ))	Fremdschlüssel für Zusatzinformationen über Ansprechpartner und Firmen
	Identische Schlüssel in den Tabellen Firma und APA werden durch den Einsatz eines SEQUENZ-Objekts verhindert. Die Summe der Primärschlüssel der Tabellen FIRMA +APA können somit keine doppelten Werte enthalten! Die Spalte ZTAB zur Identifikation der Tabelle wird nicht benötigt.				
	Info	INFO	Lange Zeichenspalte		
	INFOINSERTDATE	INFINSDAT	Zeitstempel		Default: Aktueller Timestamp
	INFOINSERTUSER	INFINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	INFOUPDATEDATE	INFUPDDAT	Zeitstempel		Default: Aktueller Timestamp
	INFOUPDATEUSER	INFUPDUSR	Kurze Zeichenspalte		Benutzerprofilname
PERSON					
	PEKURZ	PEKURZ	Prim_AS_Identity	PK	Fortlaufende eindeutige Nummer AS IDENTITY
	Vorname	VNAME	Zeichenspalte		
	Nachname	NNAME	Zeichenspalte		
	SEX	SEX	Kurze Zeichenspalte	FK/RI(DETAIL(DeKURZ)	Default NOT NULL

	ANREDE	ANREDE	Fremdschlüssel	FK/RI(DETAIL(DeKURZ)	Default NOT NULL
	PERSONINSERTDATE	PERINSDAT	Zeitstempel		Default: Aktueller Timestamp
	PERSONINSERTUSER	PERINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	PERSONUPDATEDATE	PERUPDDAT	Zeitstempel		Default: Aktueller Timestamp
	PERSONUPDATEUSER	PERUPDUSR	Kurze Zeichenspalte		Benutzerprofilname
KONTAKT (Entität Kommunikation)					
	KOKURZ	KOKURZ	Prim_AS_Identity	PK	Fortlaufende eindeutige Nummer AS IDENTITY
	Identische Schlüssel in den Tabellen Firma und APA werden durch den Einsatz eines SEQUENZ-Objekts verhindert. Die Summe der Primärschlüssel der Tabellen FIRMA +APA können somit keine doppelten Werte enthalten! Die Spalte KTAB zur Identifikation der Tabelle wird nicht benötigt.				
	KKEY	KKEY	Fremdschlüssel	FK/RI (APA(APKURZ)) + (FIRMA(FiKURZ))	Default NOT NULL
	MEDIUM	MEDIUM	Fremdschlüssel	FK/RI DETAIL(DeKURZ)	Default NOT NULL
	Nummer	Nummer	Zeichenspalte		Telefonnummer, Faxnummer, e-Mail-Adresse etc
	KONTAKTINSERTDATE	KONINSDAT	Zeitstempel		Aktueller Timestamp durch Insert
	KONTAKTINSERTUSER	KONINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	KONTAKTUPDATEDATE	KONUPDDAT	Zeitstempel		aktueller Timestamp durch UpDate
	KONTAKTUPDATEUSER	KONUPDUSR	Kurze Zeichenspalte		Benutzerprofilname
FIRMA					
	FIKURZ	FIKURZ	Prim_Sequenz	PK	Fortlaufende eindeutige Nummer durch SEQUENZ Objekt FIR_APA
	PEKURZ	PEKURZ	Fremdschlüssel	FK/RI (Person(PeKURZ))	Default NOT NULL

	FirmenZusatz	FZusatz	Zeichenspalte		
	Page	Page	DataLink		URL, No Control
	FIRMAINSERTDATE	FIRINSDAT	Zeitstempel		Default: Aktueller Timestamp
	FIRMAINSERTUSER	FIRINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	FIRMAUPDATEDATE	FIRUPDDAT	Zeitstempel		Default: Aktueller Timestamp
	FIRMAUPDATEUSER	FIRUPDUSR	Kurze Zeichenspalte		Benutzerprofilname
HDROrder (Entität Auftrag)					
	HDKURZ	HDKURZ	Prim_Sequenz	PK	Fortlaufende eindeutige Nummer durch SEQUENZ Objekt HDR_POS
	FIKURZ	FIKURZ	Fremdschlüssel	FK/RI (Firma(FiKURZ))	Default NOT NULL
	APKURZ	APKURZ	Fremdschlüssel	FK/RI (APA(ApKURZ))	Default NOT NULL
	EingangsDATUM	EINDAT	Datum		
	SollDatum	SOLLDAT	Datum		
	ISTDATUM	ISTDAT	Datum		
	Text	Text	Zeichenspalte		
	Status	Status	Ganze Zahl	FK/RI (DETAIL(DeKURZ))	Default NOT NULL Stati: 1 = geplant 2 = Vertrag 3 = erledigt 4 = Ablage
	HDRORDERINSERTDATE	HDRINSDAT	Zeitstempel		Default: Aktueller Timestamp
	HDRORDERINSERTUSER	HDRINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	HDRORDERUPDATEDATE	HDRUPDDAT	Zeitstempel		Default: Aktueller Timestamp
	HDRORDERUPDATEUSER	HDRUPDUSR	Kurze Zeichenspalte		Benutzerprofilname

WERT (Entität PREIS)					
	WEKURZ	WEKURZ	Prim_AS_Identity	PK	Fortlaufende eindeutige Nummer AS IDENTITY
	Identische Schlüssel in den Tabellen OrderPos und OrderHDR werden durch den Einsatz eines SEQUENZ-Objekts verhindert. Die Summe der Primärschlüssel der Tabellen OrderHDR +OrderPios können somit keine doppelten Werte enthalten! Die Spalte PTAB zur Identifikation der Tabelle wird nicht benötigt.				
	WKEY	WKEY	Fremdschlüssel	FK/RI (OrderHDR(HDKURZ)) (Orderpos(POKurz))	Default NOT NULL
	KALKART	KALKART	Fremdschlüssel	FK/RI (DETAIL(DeKURZ))	Default NOT NULL
	EINHEIT	EINHEIT	Fremdschlüssel	FK/RI (DETAIL(DeKURZ))	Default NOT NULL
	Anzahl	Anzahl	Ganze Zahl		Anzahl berechneter bzw. kalkulierter Einheiten
	Betrag	Betrag	Euro Betrag		Nettowert einer berechneten oder kalkulierten Einheit
	MWST	MWST	Ganze Zahl		MwSt-Satz: 0.16, 0.19, 0.07
	WERTINSERTDATE	WERINSDAT	Zeitstempel		Aktueller Timestamp Insert
	WERTINSERTUSER	WERINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	WERTUPDATEDATE	WERUPDDAT	Zeitstempel		Aktueller Timestamp UpDate
	WERTUPDATEUSER	WERUPDUSR	Kurze Zeichenspalte		Benutzerprofilname
POSORDER (Entität Leistung)					
	POKURZ	POKURZ	Prim_Sequenz	PK	Fortlaufende eindeutige Nummer durch SEQUENZ Objekt HDR_POS
	ORKURZ	ORKURZ	Fremdschlüssel	FK/RI (OrderHDR(OrKURZ))	Default NOT NULL

	IstDatum	ISTDATUM	Datum		Datum der Fertigstellung
	Text	Text	Zeichenspalte		Individuell beschreibender Text,
	RENR	RENR	Ganze Zahl		Rechnungsnummer der Fakturierung Einzelpositionen sollen fakturierbar sein auch wenn der gesamte Auftrag noch nicht abgeschlossen ist
	ReDat	ReDat	Datum		Datum der Fakturierung
	POSORDERINSERTDATE	POSINSDAT	Zeitstempel		Aktueller Timestamp Insert
	POSORDERINSERTUSER	POSINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	POSORDERUPDATEDATE	POSUPDDAT	Zeitstempel		Aktueller Timestamp UpDate
	POSORDERSUPDATEUSER	POSUPDUSR	Kurze Zeichenspalte		Benutzerprofilname
MASTER (Entität ASTERKEY)					
	MAKURZ	MAKURZ	Prim_AS_Identity	PK	Fortlaufende eindeutige Nummer AS IDENTITY
	HGRP	HGRP	Fremdschlüssel	FK(Master(MaKURZ))	zugehörige Hauptgruppe
	BZName	BZNAME	Zeichenspalte		Bezeichnung
	MASTERINSERTDATE	MASINSDAT	Zeitstempel		Aktueller Timestamp
	MASTERINSERTUSER	MASINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	MASTERUPDATEDATE	MASUPDDAT	Zeitstempel		Aktueller Timestamp
	MASTERUPDATEUSER	MASUPDUSR	Kurze Zeichenspalte		Benutzerprofilname
DETAIL (Entität ETAILKEY)					
	DEKURZ	DEKURZ	Prim_AS_Identity	PK	Fortlaufende eindeutige Nummer AS IDENTITY

7.2.4
Seite 12

	MAKURZ	MAKURZ	Fremdschlüssel	FK(Master(MaKURZ))	
	BZKURZ	BZKURZ	Zeichenspalte		Kurzbezeichnung
	BZLANG	BZLANG	Zeichenspalte		Langbezeichnung
	DETAILINSERTDATE	DETINSDAT	Zeitstempel		Aktueller Timestamp INSERT
	DETAILINSERTUSER	DETINSUSR	Kurze Zeichenspalte		Benutzerprofilname
	DETAILUPDATEDATE	DETUPDDAT	Zeitstempel		Aktueller Timestamp UPDATE
	DETAILUPDATEUSER	DETUPDUSR	Kurze Zeichenspalte		Benutzerprofilname

7.2.4.1 Datenbankintegrität

Als Datenbankintegrität oder Datenbankkonsistenz wird die Richtigkeit der Daten, die in der Datenbank gespeicherten sind, bezeichnet. Dazu gehört die technische beziehungsweise physische Integrität. Darüber hinaus sollen die Daten aber auch den Abhängigkeiten und Bedingungen genügen, denen die Daten unterworfen sind. Man spricht in diesem Zusammenhang von der logischen Konsistenz der Datenbank.

Logische Datenkonsistenz ist kein neues Problem! Vielmehr existiert es seit Herman Hollerith, der seinerzeit die Lochkarten erfand. Mittlerweile wurde aber zur Lösung dieses Problems von allen Datenbankherstellern ein Konzept entwickelt: die Integritätsregeln!

Integritätsregeln sollen verhindern, dass fehlerhafte Eingaben die Datenbank zerstören. Konsistenzregeln und daraus abgeleitete Aktionen, die die Integrität der Daten sicherstellen, werden in der Praxis jedoch oftmals innerhalb der Anwendungsprogramme definiert. Auf diese Weise verteilen sich Behandlung und damit auch Verantwortung für die Konsistenz der Daten unter Umständen auf viele verschiedene Applikationen und deren Programmierer. Das bedeutet, dass jedes Anwendungsprogramm, das Daten aktualisiert, löscht oder einfügt, diese Datenabhängigkeiten in entsprechenden Algorithmen berücksichtigen muss. Jeder Programmierer muss somit die Relationen zwischen den Tabellen genau kennen, um die Datenintegrität zu gewährleisten. Was passiert aber, wenn Sie mit Tools auf die Datenbank zugreifen, die diese Algorithmen nicht beinhalten? Es gibt keine Möglichkeit, die Integritätsprüfungen in Dienstprogrammen zu aktivieren, wie zum Beispiel in Data File Utility (DFU) oder CPYF (Copy File). Und was ist mit all den Programmierern, die neu in das Team kommen und die Datenbank nicht aufgebaut haben?

Es ist daher schon seit längerem Stand der Technik, Integritätsregeln innerhalb der Datenbank und somit zentral an einer Stelle zu verwalten, so dass der Anwendungsprogrammierer vom Ballast der Konsistenzprüfung weitgehend befreit ist.

Ich arbeite bereits seit vielen Jahren im iSeries-Umfeld. Während dieser Zeit, konnte mir bisher kein Kunde beweisen, dass der Schutz der Daten auf Anwendungsebene ausreichend ist. Immer wieder stießen wir in Schulungen auf korrupte Daten. Mal waren es nur ein oder zwei Datensätze – oftmals aber auch wesentlich mehr. Die Gründe hierfür blieben häufig verborgen.

Wenn man nachfragt, warum viele Kunden noch nicht die Integritätsregeln in der Datenbank verankert haben, erhält man sehr vielfältige Antworten – wie zum Beispiel:

- Man benötigt SQL.
- Die Tabellen müssen journalisiert werden.
- Integritätsregeln machen die Datenbank langsam.
- Der Aufwand lohnt sich nicht.
- u. v. m.

Tatsache ist, dass Sie weder mit SQL noch mit Journalisierungen arbeiten müssen, um die Integritätsprüfungen auf Datenbankebene zu verlagern. Ich beschränke ich mich daher in den nachfolgenden Kapiteln zunächst einmal auf die i5/OS-Befehle. Die entsprechende SQL-Syntax finden Sie im Abschnitt 7.3.3.4. Auch der iSeries Navigator bietet nützliche Funktionen, um Integritätsbedingungen zu definieren und zu verwalten.

Ob Ihre Datenbankanwendung langsamer wird, hängt davon ab, welche Regeln Sie implementieren. Ob sich der Aufwand lohnt, können nur Sie allein entscheiden. Zugegeben, es gibt weniger wichtige Tabellenrelationen. Wenn diese Relationen korrupte Daten aufweisen, kann man vielleicht damit leben – abgesehen davon, dass eine „schiefe“ Datenlage ein Hinweis auf fehlerhafte Anwendungsprogramme sein könnte. Was geschieht aber, wenn Rechnungskopfdaten verloren gehen und wenn Sie die zugehörigen Positionsdaten nicht fakturieren können? Sie entscheiden selbst, wann sich der Aufwand lohnt!

7.2.4.1.1 Entitätsintegrität

Durch die Entitätsintegrität wird eine Zeile als eindeutige Entität innerhalb einer bestimmten Tabelle definiert mithilfe von UNIQUE-Schlüsseln, UNIQUE Constraints oder PRIMARY KEY Constraints. Eine Tabelle verfügt normalerweise über eine Spalte oder eine Kombination aus Spalten, die Werte enthalten, die jede Zeile in der Tabelle eindeutig identifizieren. Mithilfe von UNIQUE-Schlüsseln können Sie sicherstellen, dass keine doppelten Werte in bestimmte Spalten eingegeben werden. UNIQUE-Schlüssel sollten Ihnen hinlänglich aus DDS-Zeiten bekannt sein.

Schauen Sie sich das nachfolgende Beispiel für die Tabelle „ORT“ an:

```
A
                              UNIQUE
A                              R ORTFMT
A                                 OKURZ          .....
A                                 PLZ                 .....
A                                 ONAME          .....
A                                 .....

A                              K PLZ
A                             K ONAME
```

Die Kombination der Spalten „PLZ“ und „ONAME“ muss eindeutig sein. Nur so können Sie die doppelte Erfassung einzelner Orte vermeiden.

Verwechseln Sie aber bitte nicht einen UNIQUE-Schlüssel mit einem UNIQUE Constraint. Eine UNIQUE-Bedingung (=Constraint) können Sie nicht über DDS, sondern nur mit dem Befehl ADDPFCST (Add Physical File Constraint) einer physischen Datei hinzufügen. ADDPFCST beschreibt und definiert Datenbankbedingungen.

UNIQUE Constraint definieren

Ähnlich wie bei einem UNIQUE-Schlüssel muss die Kombination der Spalten „PLZ“ und „ONAME“ in der oberen Abbildung eindeutige Werte enthalten. Außerdem erhält die Bedingung einen Namen. Falls Sie den Integritätsnamen nicht festlegen, wird das System einen Namen generieren. Die systemgenerierten Integritätsnamen sind allerdings häufig nicht sprechend und erschweren daher die Verwaltung der Integritätsregeln. Der Name ist frei wählbar, muss aber innerhalb einer Bibliothek eindeutig sein. Sie können maximal 128 Zeichen verwenden. Wenn Sie den Integritätsnamen allerdings in Anführungszeichen setzen, können Sie sogar bis zu 256 Zeichen benutzen.

Sicherlich fragen Sie jetzt nach dem Unterschied zwischen einem UNIQUE Key, der beispielsweise über die DDS-Beschreibung generiert wurde, und dem

UNIQUE Constraint? Auf den ersten Blick ist der Unterschied nicht deutlich, aber später werden Sie feststellen, dass Sie die Beziehungen zwischen den Tabellen nur mit Hilfe der Integritätsbedingungen herstellen können. Ihre UNIQUE-Schlüssel können dabei nicht eingesetzt werden!

Eine Tabelle kann durchaus mehrere Spalten oder Spaltenkombinationen aufweisen, die eindeutig sein müssen. Eine dieser Spalten ist aber in der Regel als Hauptschlüssel zu verstehen – wie zum Beispiel: Artikelnummer in der Artikelstammdatei oder Kundennummer in der Kundenstammdatei.

Diese Spalte oder Kombination aus Spalten wird als Primärschlüssel (PK = Primary Key) der Tabelle bezeichnet. Primärschlüssel definieren Sie als PRIMARY-KEY-Einschränkung. DDS kann dabei nicht eingesetzt werden. Grundsätzlich kann eine Tabelle nur über eine einzige PRIMARY-KEY-Bedingung verfügen. In Spalten, die sich an der PRIMARY-KEY-Einschränkung beteiligen, sind NULL-Werte im übrigen nicht zulässig. Wenn Sie eine PRIMARY-KEY-Einschränkung für eine Tabelle angeben, erzwingt DBMS die Eindeutigkeit der Daten innerhalb dieser Spalte, indem ein eindeutiger Zugriffspfad für die Primärschlüsselspalten erstellt wird. Dieser Zugriffspfad ermöglicht darüber hinaus den schnellen Zugriff auf Daten, wenn der Primärschlüssel in Abfragen verwendet wird. Wenn eine PRIMARY-KEY-Einschränkung für mehrere Spalten definiert wird muss jede Kombination aus den Werten aller Spalten, die in der Definition der PRIMARY-KEY-Einschränkung enthalten sind, eindeutig sein. Schauen wir uns auch hierzu ein Beispiel an:

Primary Key Constraint Definition

In der oberen Abbildung definiere ich für die Tabelle „ORT“ eine Primärschlüsselbedingung. Der Primary Key dieser Tabelle ist gemäß Datenbankentwurf die Spalte „OKURZ“. Häufig werden künstliche Spalten als Primärschlüssel vereinbart, um den Datenpfad für den Schlüssel möglichst klein zu halten.

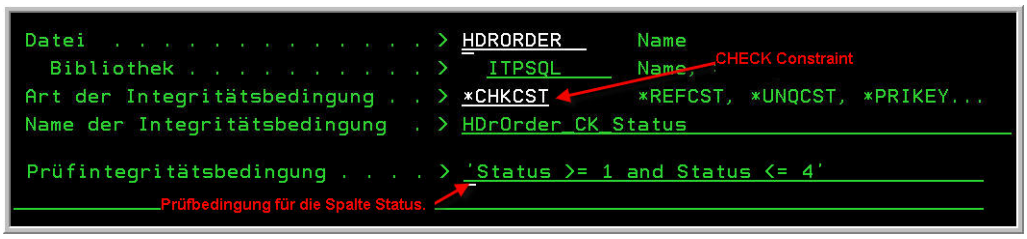
Abschließend stellt sich die Frage: Wodurch unterscheiden sich UNIQUE Constraints von Primärschlüsselbedingungen? Die nachfolgende Tabelle soll die Unterschiede verdeutlichen:

UNIQUE-Bedingung	PRIMARY KEY-Bedingung
Eindeutige Identifikation von Datensätzen	Eindeutige Identifikation von Datensätzen
NULL-fähige Spalten sind zulässig	NULL-fähige Spalten sind NICHT zulässig
Es kann mehrere UNIQUE Constraints in einer Tabelle geben	Eine PRIMARY KEY-Bedingung je Tabelle
	Spezielle Ausprägung eines UNIQUE Constraints

7.2.4.1.2 Domänenintegrität

Domänenintegrität bezeichnet die Gültigkeit der Einträge einer bestimmten Spalte. Sie definiert letztendlich den gültigen Wertebereich innerhalb einer Spalte. Sie kann durch den Datentyp, durch DEFAULT- und/oder NOT-NULL-Definitionen und durch CHECK-Constraints erzwungen werden.

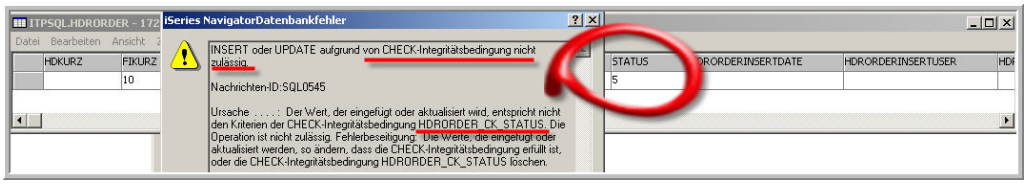
In unserer Beispieldatenbank legen wir für viele Spalten durch den Datentyp bereits die Wertebereiche der Spalten fest. Aus diesem Grund wurden zum Beispiel alle Datumsfelder mit dem Datentyp DATE definiert. Verhindert werden dadurch fehlerhafte Datumseingaben. In einer solchen Spalte kann der Anwender nicht versehentlich den „30.02.2007“ erfassen. Der Wert würde dem Datentyp widersprechen und von der Datenbank abgelehnt werden. Darüber hinaus gibt es benutzerdefinierte Wertebereiche, die wir nicht durch entsprechende Datentypen realisieren können. Nehmen wir beispielsweise die Spalte Status in der Tabelle „HDROrder“. Sie repräsentiert den Auftragsstatus und kann nur die Werte „1“ bis „4“ annehmen. Hier setzen wir eine Prüfintegri tätsbedingung, die diese benutzerdefinierte Bedingung schützt:



Prüfintegritätsbedingung definieren

Sie sehen, ich grenze unabhängig vom Spalten-Datentyp den gültigen Wertebereich ein.

Sobald ein Anwender versucht, einen anderen Wert einzugeben, wird die Zeile nicht gespeichert werden.



Fehlersituation aufgrund einer ungültigen Benutzereingabe

7.2.4.1.3 Referentielle Datenintegrität

Die referentielle Integrität sichert die definierten Beziehungen zwischen den Tabellen, wenn Daten erfasst, geändert oder gelöscht werden. In der DB2/UDB for iSeries basiert die referentielle Integrität auf Beziehungen zwischen Fremdschlüsseln und Primärschlüsseln oder zwischen Fremdschlüsseln und UNIQUE Constraints. Die referentielle Integrität stellt die tabellenübergreifende Konsistenz von Daten sicher. Eine solche Konsistenz erfordert, dass keine Verweise auf nicht vorhandene Werte bestehen und dass beim Ändern eines Schlüsselwertes alle Verweise auf diesen Wert in der Datenbank konsistent geändert werden.

In nahezu jeder Datenbankanwendung gibt es Tabellen, in denen die Daten einer Tabelle von Daten einer anderen Tabelle abhängig sind. So auch bei meiner Beispieldatenbank. Ein Auftrag kann nur erfasst werden, wenn der entsprechende Kunde existiert. Der Kunde darf wiederum nicht gelöscht werden, solange noch Aufträge vorhanden sind. Um die Integrität von Abhängigkeitsbeziehungen zwischen Tabellen oder zwischen Datensätzen derselben Tabelle sicherzustellen, müssen Fremdschlüssel (Foreign Keys) definiert werden. Deren Attribute müssen mit den entsprechenden Schlüsseln der Mastertabelle übereinstimmen. Durch die referentiellen Integritätsprüfungen wird sichergestellt, dass bei einer Veränderung von Fremdschlüsselspalten geprüft wird, ob dem neuen Foreign-Key-Wert ein entsprechender Wert in der Master-Tabelle gegenübersteht. In der Master-Tabelle können nur dann Werte geändert beziehungsweise Datensätze gelöscht werden, wenn keine abhängigen Datensätze mehr in der Foreign-Key-Tabelle vorhanden sind. Referentielle Datenintegrität ist also ein Mechanismus auf Datenbankebene, der die Vollständigkeit der Daten zwischen relationalen Tabellen sicherstellt. Die DB-Entwickler können Bedingungen für die physischen Dateien beziehungsweise für Tabellen definieren, die sicherstellen, dass jeder

Zeile in der Tabelle A eine Zeile in der Tabelle B entspricht. Kein Programm kann anschließend Daten einfügen, aktualisieren oder löschen, wenn dadurch die entsprechende Regel verletzt würde. Zum besseren Verständnis wollen wir auch hierfür ein Beispiel aus unserer Beispieldatenbank betrachten:

Tabelle „Firma“

FiKurz	PeKurz	Firmen-Zusatz
10	1	GmbH & Co. KG
20	3	
40	5	e. K
50	4	GmbH

Primärschlüssel der Tabelle „FIRMA“ ist die Spalte „FiKurz“. Mit Hilfe dieser Spalte können die Aufträge des Kunden in der Tabelle „HDROrder“ spezifiziert werden. Die Spalte „FiKurz“ wurde allein zu diesem Zweck in der Tabelle „HDROrder“ implementiert. Sie ist der Fremdschlüssel der Tabelle, über den die Relation zwischen den beiden Tabellen abgebildet wird:

Tabelle „HDROrder“

HDKurz	FiKurz	ApKurz	Eingangs-Datum
15	10		01.05.2007
20	20	30	24.08.2007
40	50	60	12.06.2006
50	10		02.08.2007

Bereits in unserem Konzept haben wir hierfür einen Geschäftprozess definiert, der jetzt zum Tragen kommt: Für jede Firmennummer („FiKurz“) in der Tabelle „HDROrder“ muss eine übereinstimmende Firmennummer in einer Zeile der Tabelle „FIRMA“ zugeordnet sein. Das bedeutet, dass für die Firma „10“ zwei Aufträge vorliegen: Auftragsnummer 15 und Auftragsnummer 50. Unsere Geschäftsregel bewirkt folgende Bedingungen, die implementiert werden müssen:

Hinzufügen neuer Datensätze:

- Werden neue Auftragsdaten in der Tabelle „HDROrder“ hinzugefügt, muss es in der Tabelle „Firma“ einen korrespondierenden Satz geben, dessen Wert mit dem neuen Inhalt der Spalte „FiKurz“ übereinstimmt.
- In der Tabelle „Firma“ können neue Zeilen jederzeit und ohne Einschränkungen eingefügt werden.

Änderung bestehender Datensätze:

- Wird eine Firmennummer („FiKurz“) in der Tabelle „HDROrder“ geändert, muss es für den neuen Wert ebenfalls einen entsprechenden Wert in der Tabelle „Firma“ geben.
- Wird eine bestehende Firmennummer in der Tabelle „Firma“ geändert, darf es keinen abhängigen Wert mehr in der Tabelle „HDROrder“ geben.

Löschen bestehender Datensätze:

- Wird eine Firma gelöscht, darf es keine Übereinstimmungen in der Tabelle „HDROrder“ geben.
- In der Auftragstabelle „HDROrder“ hingegen können Zeilen jederzeit ohne Beschränkung gelöscht werden.

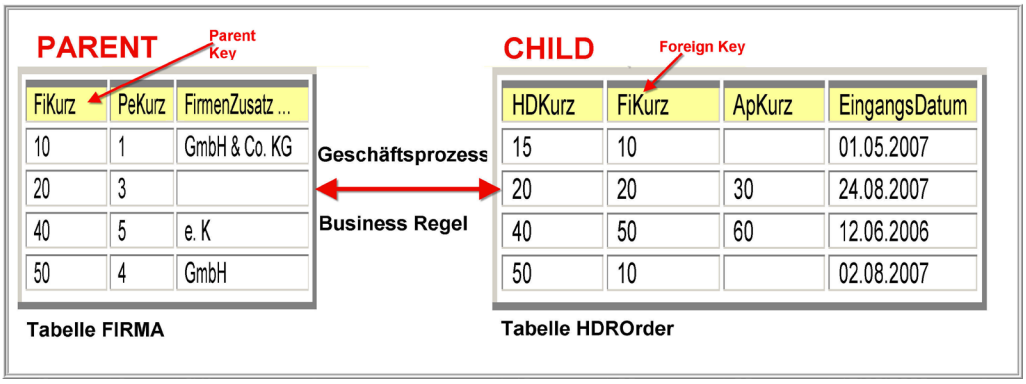
Um diese Regeln einzuhalten, müssen Sie entweder entsprechende Algorithmen in den Anwendungsprogrammen hinterlegen – oder Sie implementieren die Regeln mit Hilfe von referentiellen Integritätsbedingungen auf der Datenbankebene. Letzteres reduziert die Zeilenanzahl in Ihrem Programm und führt zu einer besseren Wartbarkeit der Programme.

Es empfiehlt sich, Integritätsbedingungen immer dann zu nutzen, wenn sichergestellt werden muss, dass Werte einer Spalte den Werten einer anderen Spalte entsprechen müssen. Dabei ist es unerheblich, ob es sich um zwei verschiedene Tabellen oder um eine Tabelle handelt. Diese Regeln können problemlos im Datenbankentwurf und/oder während der Implementierung einer neuen Datenbank integriert werden. Aber auch bestehende Datenbanken und deren Anwendungen können so verändert werden, dass sie von den Datenbankregeln profitieren. Um einen nahtlosen Übergang zu gewährleisten, ist der Planungsaufwand bei bestehenden Datenbanken höher, wenn die Überprüfungen aus den Anwendungsprogrammen entfernt und auf Datenbankebene verlagert werden.

Referentielle Integritätsbedingungen sind bereits Bestandteil des SQL ANSI/ISO-Standards von 1992. IBM hat das Konzept sowohl mit nativen i5/OS-Befehlen umgesetzt als auch im SQL-Standard implementiert.

7.2.4.1.3.1 Definition einer referentiellen Integritätsbedingung

Lassen Sie uns nun, für unsere Beispieltabellen eine solche Regel auf Datenbankebene implementieren: Das Konzept ist nicht schwer zu verstehen, wenn Sie sich die Beziehungen zwischen den Tabellen immer als Eltern-Kind-Beziehungen vorstellen. Schauen Sie sich das nachfolgende Beispiel noch einmal an:



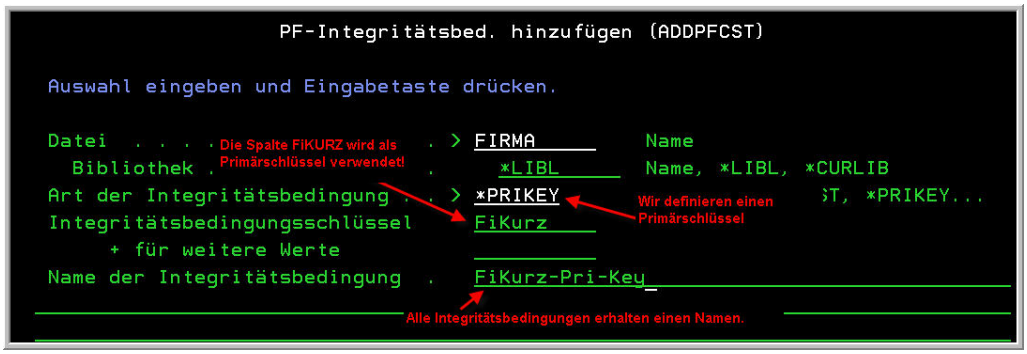
Parent-Child-Relation

Die Tabellen „Firma“ und „HDRorder“ verfügen über eine Fremdschlüsselbeziehung, die auf der Spalte „FiKurz“ basiert. In der Elterndatei „FIRMA“ muss die Firmennummer bereits vorhanden sein, wenn ein zugehöriger Auftrag erfasst werden soll. Daraus ergibt sich die allgemeingültige Aussage:

Jeder Foreign-Key-Wert aus der abhängigen Child-Tabelle, der nicht NULL ist, muss einen übereinstimmenden Parent-Key-Value in der Elterndatei besitzen. Ein Parent-Key kann eine Primär- oder eine UNIQUE-Bedingung in der Elterndatei sein. Beide – UNIQUE-Bedingungen und PRIMARY KEYS – sind potentielle Parent-Key-Kandidaten. Wenn die referentielle Bedingung gesetzt wird, muss der Zugriffspfad für den Foreign Key nicht unbedingt existieren. Allerdings benötigen Sie einen existierenden Parent Key. Das heißt: In der Elterndatei muss bereits eine Primary-Key- oder UNIQUE-Bedingung hinterlegt sein. Sie müssen sicherstellen, dass für unsere Tabelle „Firma“ ein Elternschlüssel existiert!

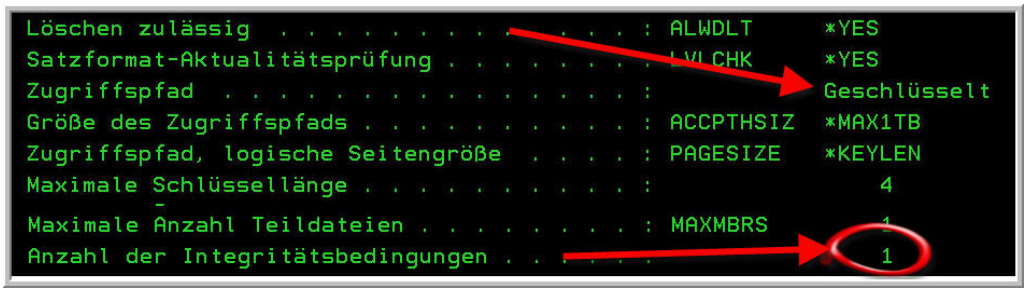
Ich definiere für mein Beispiel zunächst einmal einen Primärschlüssel, der anschließend als Parent Key verwendet werden kann.

7.2.4
Seite 22



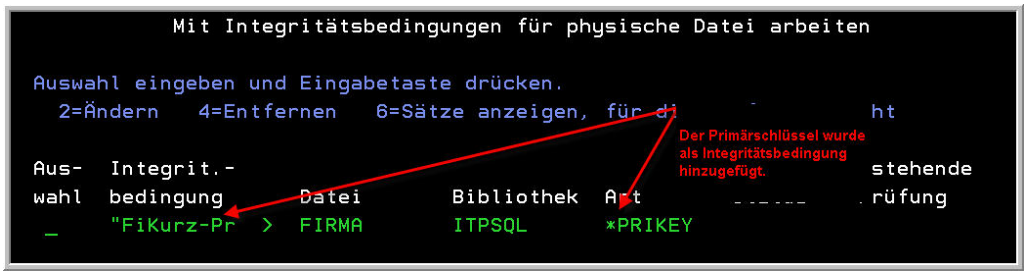
Primärschlüsseldefinition

Das Primärschlüsselfeld in meinem Beispiel ist die Spalte „FiKurz“ in der Tabelle „Firma“. Die Art der Bedingung lautet „*PRIKEY“. Wenn Sie sich anschließend noch mal die Dateibeschreibung anschauen, können Sie erkennen, dass für die Datei jetzt ein geschlüsselter Zugriffspfad erstellt und eine Integritätsbedingung hinzugefügt wurde.



Dateibeschreibung

Besser lassen sich die Bedingungen mit dem Befehl „WRKPFCST“ (Work with Physical File Constraint) überprüfen.

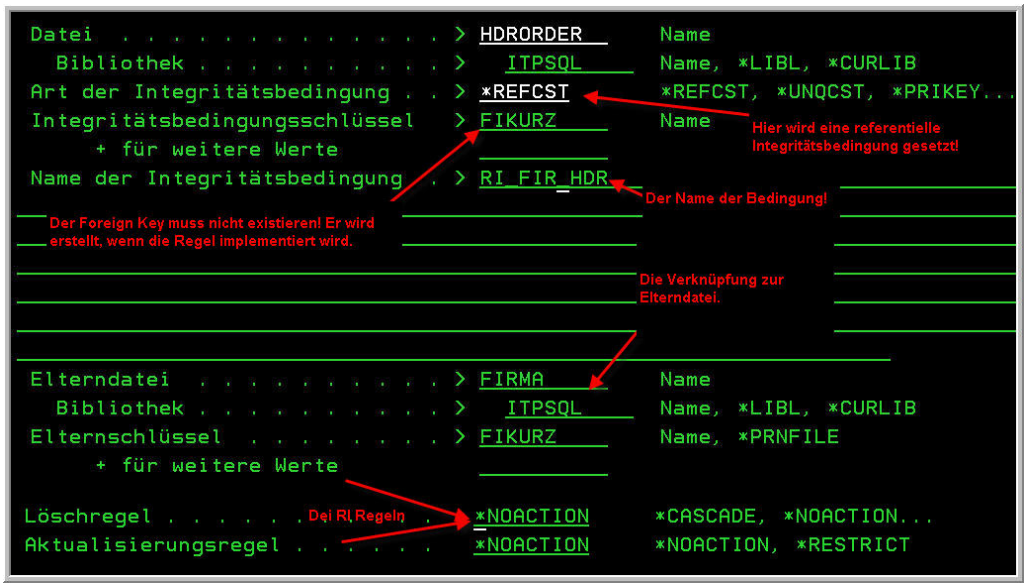


Befehl „WRKPFCST“

Der Befehl ermöglicht das Arbeiten mit Bedingungen einer einzelnen Datei, einer einzelnen Bibliothek oder einer Gruppe von Bibliotheken. Außerdem können Sie wählen, mit welcher Art von Integritätsbedingungen Sie arbeiten möchten. Als Ergebnis wird ein Panel mit den Namen der Constraints und anderer Attribute angezeigt. Gleichzeitig werden Optionen zum Löschen,

Ändern und/oder Verifizieren jedes einzelnen Constraint angeboten. Der Befehl enthält darüber hinaus eine Option zur Darstellung von Datensätzen, die sich in einer Check-Pending-Situation befinden. Doch dazu später mehr.

Kehren wir zum Beispiel zurück: Im nächsten Schritt setze ich die referentielle Bedingung. Hierbei verwende ich wiederum den Befehl ADDPFCST:



Definition der referentiellen Bedingung

Wie bereits erwähnt, wird die Bedingung immer in der Child-Tabelle definiert – also in der Tabelle „HDROrder“. Anschließend wird die Verbindung zur Elterndatei „Firma“ und dem Primärschlüssel „FiKurz“ hergestellt. Wenn eine referentielle Integritätsbedingung eingerichtet wird, muss entweder ein neuer Zugriffspfad für Foreign Key erstellt oder ein vorhandener Zugriffspfad mit passenden Attributen gemeinsam genutzt werden. Für eine physische Datei können Sie bis zu 300 Integritätsbedingungen einrichten, was ausreichen sollte.

Zum Schluss werden die Lösch- und Aktualisierungsregeln definiert. Das heißt: Welche Aktionen sollen ausgeführt werden, wenn eine Zeile geändert oder gelöscht wird. Mit der Aktualisierungs- beziehungsweise Löschregel wird die Auswirkung, die das Aktualisieren oder Löschen eines Satzes in der Elterndatei und in der abhängigen Datei hat, eingeschränkt beziehungsweise definiert. Die Regeln werden festgelegt, wenn der Referential Constraint hinzugefügt wird. Im einfachsten Fall kann die Datenintegrität dadurch bewahrt werden, dass die entsprechende Operation nicht erlaubt wird, wenn sie die Datenintegrität verletzen würde.

7.2.4
Seite 24

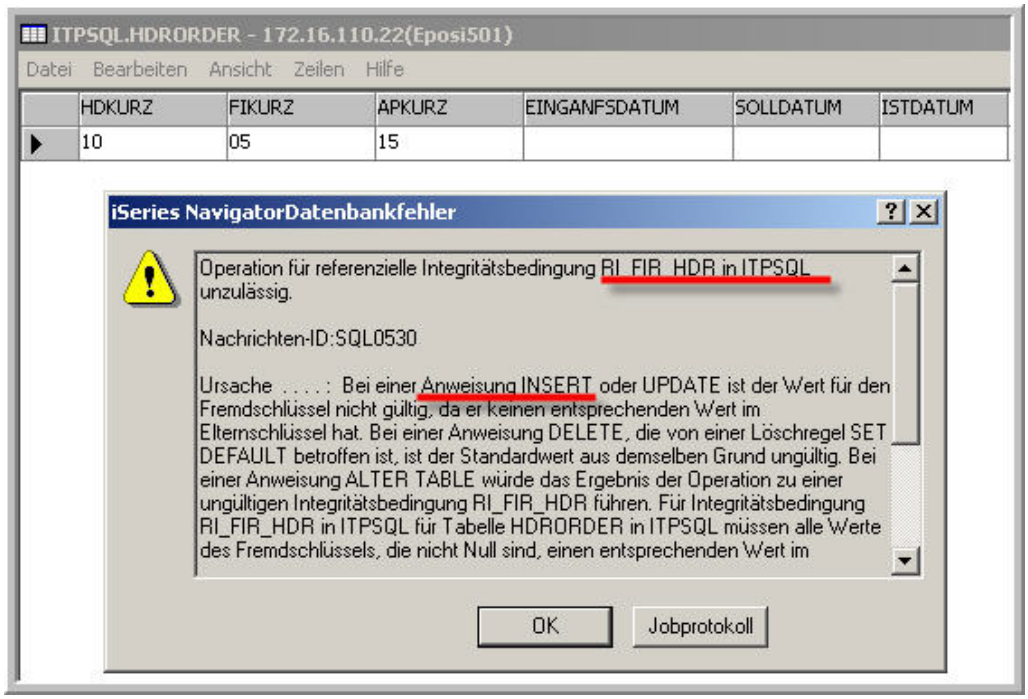
Folgende Regeln stehen Ihnen derzeit zur Verfügung:

INSERT-Regel

NoAction

Das ist die Standardregel. Alternativen hierzu existieren nicht, so dass die Insert-Regel immer implizit durch das System festgelegt ist. Sie müssen die Regel nicht ausdrücklich im Constraint definieren. Aufgrund dieser Regel kann eine Zeile nur dann eingefügt werden, wenn der Foreign Key Value NULL ist oder wenn es einen entsprechenden Parent-Satz in der Elterndatei gibt.

Die Wirkung lässt sich leicht testen. Starten Sie ein beliebiges Dienstprogramm oder nutzen Sie ein beliebiges HLL-Programm. Fügen Sie dann eine Zeile in die Tabelle „HDROrder“ ein, für die es keinen entsprechenden Wert in der Elterndatei gibt. Sie sollten eine ähnliche Fehlermeldung erhalten wie folgt:



Insert-Anweisung schlägt fehl

Um das Problem zu beheben, müssten Sie hier zunächst die Firmennummer „05“ in der Tabelle „FIRMA“ anlegen. Anschließend können Sie in einem zweiten Schritt die Zeile in der Child-Tabelle erfassen.

UPDATE-Regeln

***NOACTION**

Ein Parent Key der Eltern Tabelle kann nicht geändert werden, wenn der Primary Key Value mit einem Foreign Key Value der abhängigen Tabelle übereinstimmt.

Auch ein Foreign Key Value kann nicht geändert werden. Außer es existiert in der Elterntabelle ein Primärschlüssel, der wieder mit dem neuen Foreign Key Value übereinstimmt.

Die Prüfung wird – vom logischen Gesichtspunkt aus – am Ende einer Aktualisierungsanfrage oder einer Transaktionsgruppe durchgeführt. Es ist daher erforderlich, dass die Datenbankdateien aufgezeichnet werden, damit notfalls das DBMS ein Rollback durchführen kann.

***RESTRICT**

Das Aktualisieren eines Satzes in einer Elterndatei ist nur dann zulässig, wenn Daten eines Elternschlüssels, der ungleich NULL ist, nicht mit den Daten eines Fremdschlüssels übereinstimmen.

Auch ein Foreign Key Value kann nicht geändert werden. Außer es existiert in der Elterntabelle ein Primärschlüssel, der wieder mit dem neuen Foreign Key Value übereinstimmt.

Die Aktualisierungsregel RESTRICT wird zu Beginn der Aktualisierungsanfor derung erzwungen. Das heißt: Bevor die Datenbankoperationen ausgeführt werden, findet die Integritätsprüfung statt. Es können somit auch keine anderen Ereignisse vor der Überprüfung der Datenintegrität stattfinden. Eine Journalisierung der Datenbank ist daher nicht zwingend erforderlich.

DELETE-Regel

***NOACTION**

Das Löschen eines Satzes in einer Elterndatei ist nicht eingeschränkt, wenn der Elternschlüssel, der ungleich Null ist, nicht mit den Daten eines Fremdschlüssels übereinstimmt. Die Löschoperation findet hingegen nicht statt. Das heißt: Sie ist eingeschränkt, wenn Daten eines Elternschlüssels, der ungleich Null ist, mit Daten eines Fremdschlüssels übereinstimmen. Der Foreign Kay kann jederzeit gelöscht werden.

Ergänzung 4/2008

7.2.4
Seite 26

***RESTRICT**

Die Löschregel *RESTRICT wird im Gegensatz zur Löschregel *NOACTION wieder zu Beginn der Löschanforderung erzwungen. Dabei ist das Löschen eines Satzes in einer Elterndatei zulässig, wenn Daten eines Elternschlüssels, der ungleich Null ist, nicht mit den Daten eines Fremdschlüssels übereinstimmen. Das Löschen eines Satzes in einer Elterndatei ist eingeschränkt, wenn Daten eines Elternschlüssels, der ungleich Null ist, mit Daten eines Fremdschlüssels übereinstimmen.

Die UPDATE und DELETE Regeln *NOACTION und *RESTRICT unterscheiden sich nur durch den Prüfzeitpunkt. Wurde *RESTRICT vereinbart, wird die Prüfung unmittelbar ausgeführt. Wurde *NOACTION gewählt, sind zunächst andere Ereignisse – zum Beispiel die Ausführung von Datenbank-Triggern – zulässig. Die Prüfung findet erst am Ende der Aktualisierungsanfrage statt. Deshalb erfordert die Regel *NOACTION die Protokollierung der Parent- und der Child-Tabelle. Dies ist zur Sicherstellung der Datenintegrität wichtig. Außerdem erlaubt es die Wiederherstellung einer verschachtelten Transaktionsgruppe, die nicht erfolgreich zu Ende geführt werden konnte.

***SETNULL**

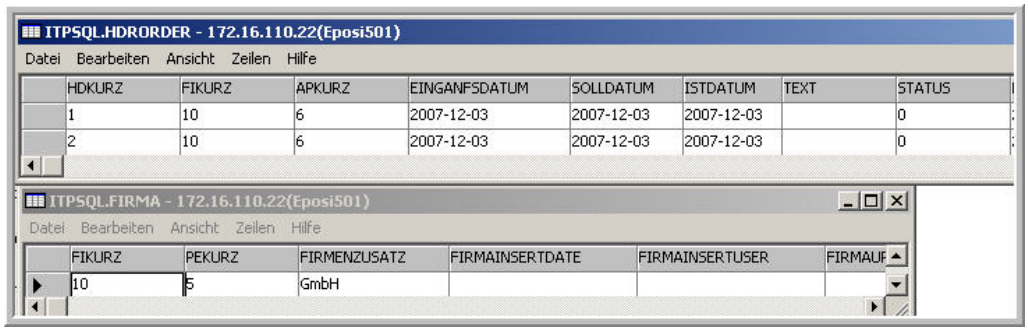
Alternativ steht noch die Löschregel SETNULL zur Verfügung. Durch das Löschen eines Satzes in der Elterndatei werden die entsprechenden Sätze in der abhängigen Datei mit dem Wert NULL aktualisiert. Wenn der Fremdschlüssel Nullwerte enthalten darf, wird der Wert auf NULL gesetzt. Wenn das übereinstimmende Feld des Fremdschlüssels keine Nullwerte enthalten kann, wird das Feld nicht aktualisiert. Voraussetzung für die Verwendung dieser Regel ist, dass mindestens ein Feld im Zugriffspfad des FOREIGN KEY Nullwerte aufweisen kann.

***SETDFT**

Wenn Sie die Regel *SETDFT nutzen, werden die übereinstimmenden Sätze in der abhängigen Child-Tabelle mit den Standardwerten aktualisiert.

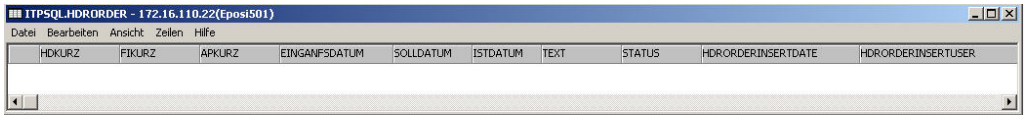
***CASCADE**

Die Löschregel CASCADE bewirkt, dass ein Satz in der Elterndatei gelöscht werden kann – auch wenn es abhängige Child-Sätze gibt. Wenn die Regel zum Tragen kommt, werden alle abhängigen Datensätze der Child-Tabellen automatisch gemeinsam mit dem Parent-Satz gelöscht. Schauen wir uns hierzu ein kleines Beispiel an:



Ausgangssituation

In der Tabelle „FIRMA“ gibt es einen Kunden mit der Firmennummer „10“. Für diesen Kunden wurden zwei Aufträge erfasst. Zwischen den Tabellen gibt es eine Integritätsbedingung, wobei *CASCADE als Löschregel verwendet wurde. Schauen wir uns an, was passiert, wenn wir den Kunden mit der Firmennummer „10“ löschen.



*Ergebnis einer *CASCADE-Bedingung*

Die abhängigen Datensätze wurden gelöscht. Sie erhalten übrigens keine Rückmeldung des Systems, wenn Sie die cascadierende Löschregel verwenden.

Fassen wir zusammen:

	Insert	Update	Delete
No Action	Default	· Datenbank-Journalisierung erforderlich · Default · UPDATE-Anweisungen werden zurückgewiesen, wenn die Datenintegrität verletzt wird	· Datenbank-Journalisierung erforderlich · Default · DELETE-Anweisungen werden zurückgewiesen, wenn die Datenintegrität verletzt wird
Restrict		· KEINE Journalisierung erforderlich · UPDATE-Anweisungen werden zurückgewiesen, wenn die Datenintegrität verletzt wird	· KEINE Datenbank-Journalisierung erforderlich · DELETE-Anweisungen werden zurückgewiesen, wenn die Datenintegrität verletzt wird

	Insert	Update	Delete
CASCADE			· Datenbank-Journalisierung erforderlich · DELETE-Anweisungen werden cascadierend ausgeführt
SET NULL			· Datenbank-Journalisierung erforderlich · Abhängige Foreign Key Values erhalten den Wert NULL
SET DEFAULT			· Datenbank-Journalisierung erforderlich · Abhängige Foreign Key Values erhalten den Standardwert

Abschließend sehen Sie einige Einschränkungen, die meiner Meinung nach selbstverständlich sein sollten:

- Integritätsbedingungen können weder Systemdateien noch programmbeschriebenen Dateien hinzugefügt werden.
- Multi Member-Tabellen werden nicht unterstützt.
- Integritätsbedingungen können keiner Datei hinzugefügt werden, die derzeit für einen Benutzerjob geöffnet ist.
- Referentielle Integritätsbedingungen können sich nicht auf mehrere Zusatzspeicher-Pools (ASPs) erstrecken. RI bleibt auf einen ASP beschränkt.
- Integritätsbedingungen können keiner Datei in der Bibliothek QTEMP hinzugefügt werden.

7.2.4.1.3.2 Überprüfung referentieller Integritätsbedingungen

Bislang haben unsere Integritätsbedingungen fehlerfrei funktioniert. Allerdings kann es auch andere Situationen geben. Schauen Sie sich die folgende Abbildung einmal etwas genauer an:

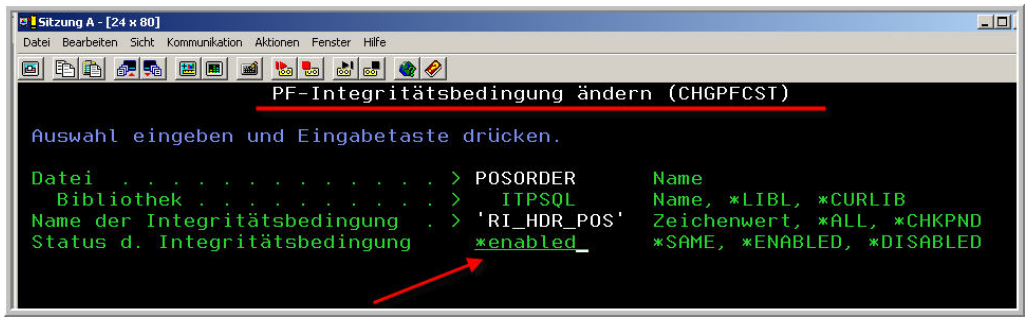
Überblick über die bestehenden Integritätsbedingungen

Ich habe eine weitere Integritätsbedingung für die Auftragspositionsdaten hinzugefügt, denn schließlich dürfen Auftragspositionen nur eingefügt werden, wenn zuvor ein Auftrag generiert wurde. Betrachten Sie die Spalte „STATUS“ in der Abbildung. Im Gegensatz zu unserer ersten Bedingung, die den Status „EST/ENB“ aufweist, finden wir beim zweiten Beispiel den Status „EST/DSB“.

Der erste Teil des Statusfeldes legt fest, ob die Datenbankbedingung einsetzbar ist: „DEFINED“ bedeutet, dass die Bedingung nicht benutzt werden kann; „Established“ (EST) bedeutet in diesem Zusammenhang, die Regel ist nutzbar. Der zweite Teil des Statusfeldes – „ENABLED“ (ENB) oder „DISABLED“ (DSB) – bestimmt, ob die Regel aktiviert oder deaktiviert ist. Ein voll funktionierender Constraint muss den Status „EST/ENB“ aufweisen, wie in unserem ersten Beispiel gezeigt wurde. Andere Stati weisen darauf hin, dass die Constraints aus verschiedenen Gründen nicht funktionieren.

Ein Grund, der die Funktion des Constraint beeinflusst, sind fehlerhafte Datensätze in der Parent- oder Child-Datei. Wenn Sie eine Bedingung hinzufügen und Ihre Child-Datei dabei leer ist oder nur ungültige Datensätze enthält, die nicht der Regel entsprechen, ist der Constraint „DEFINED“, nicht aber „ESTABLISHED“ – wie in meinem zweiten Beispiel. Auch wenn Sie jetzt Zeilen in der Tabelle „POSOrder“ einfügen, die der Bedingung entsprechen, wird die Regel nicht automatisch aktiviert. Vielmehr müssen Sie nachträglich mit den Befehlen „CHGPFCST“ (Change Physical File Constraint) oder mit dem Befehl „WRKPFCST“ (Work with Physical File Constraint) die Regel manuell aktivieren:

Ergänzung 4/2008



Manuelle Aktivierung einer Integritätsregel

Sobald Sie eine Bedingung definieren oder auch nachträglich aktivieren, wird die aktuelle Datenlage der Tabellen verifiziert. Nur wenn keine Fehler in der aktuellen Datenbasis gefunden wurden, ist der Constraint-Status „EST/ENB". Die Regel ist etabliert und aktiviert. Wenn in der Datenbasis hingegen Fehler gefunden wurden oder wenn die Integritätsbedingung – wie in meinem Beispiel – mit i5/OS Befehlen erstellt wurde, wird der Status auch auf „EST/ENB" gestellt – allerdings mit der zusätzlichen Markierung als CHECK Pending.

Check-Pending-Situation

Solange die CHECK-Pendng-Situation nicht aufgelöst ist, können Sie die Tabelle nicht nutzen. Außerdem ist es nicht möglich, neue Datensätze hinzuzufügen, zu löschen oder zu aktualisieren. Solange der Constraint als CHECK Pending gilt, ist für die CHILD-Tabelle kein Datenzugriff möglich; für die Parent-Tabelle sind nur Lese- oder Schreibzugriffe (INSERT) erlaubt. Die Fehler in der Datenbasis müssen zunächst behoben werden, um die Situation aufzulösen. Meiner Meinung nach ist das einer der wichtigsten Gründe, warum viele DB-Entwickler es noch nicht geschafft haben, nachträglich die RI-Bedingungen zu implementieren.

Gehen Sie beim Auflösen der Situation wie folgt vor:

1. Deaktivieren Sie zunächst die Integritätsbedingung, falls dies noch nicht geschehen ist, mit folgendem Befehl:

```
CHGPFCST FILE(ITPSQL/POSORDER) CST(RI_HDR_POS) STATE(*DISABLED)
```

2. Lassen Sie sich anschließend die fehlerhaften Daten mit dem Befehl „Check Pending Constraint“ (DSPCPCST) anzeigen.

```
DSPCPCST FILE(ITPSQL/POSORDER) CST(RI_HDR_POS)
```

Alternativ können Sie auch die Option „6“ des Befehls „WRKPFCST“ verwenden. Jeder der Befehle wird die abhängigen Dateien und Parent-Daten abfragen. Danach werden alle Zeilen der CHILD-Tabellen angezeigt, deren Foreign Key keinem Parent Key entsprechen.

```
Zeile    ....+....1....+....2....+....3....+....4....+....5.
         Positionsnummer  Auftragsnummer  Rechnungsnummer

         ---------------  --------------  ---------------
000001                10              25                 -
000002                20              25                 -
****** * * * * *  E N D E  D E R  D A T E N  * * * * *
```

Ergebnis des Befehls DSPCPCST

3. Lösen Sie jetzt die Datenprobleme, indem Sie DFU, SQL, Query Manager oder ein Anwendungsprogramm nutzen, so dass die Foreign Keys einen entsprechenden Elternschlüssel erhalten. Vielleicht müssen Sie die fehlerhaften Daten in der CHILD-Tabelle auch löschen. Es könnte eventuell auch nützlich sein, der Parent-Tabelle einen Datensatz mit passendem Primärschlüssel hinzuzufügen, um die Konflikte schnell zu lösen.

4. Wenn alle Fehler korrigiert sind, aktivieren Sie die Integritätsbedingung wieder mit dem Befehl „CHGPFCST“ – oder nutzen Sie hierfür die Option „2“ des „WRKPFCST“-Befehls.

```
CHGPFCST FILE(ITPSQL/POSORDER) CST(RI_HDR_POS) STATE(*ENABLED)
```

DBMS führt jetzt eine erneute Prüfung durch. Werden keine Integritätsverletzungen mehr festgestellt, wird die Integritätsbedingung mit dem Status „EST/ENB“ korrekt aktiviert.

Wenn Sie Ihre Datenbankbedingungen nicht mit nativen Befehlen implementieren und stattdessen SQL-Anweisungen nutzen, erhalten Sie keine Check-Pending-Situationen. Vielmehr wird die Bedingung mit einer Fehlermeldung zurückgewiesen. Die Bedingung ist nicht definiert und auch nicht etabliert. Weitere Informationen entnehmen Sie bitte dem Kapitel 7.3.3.4ff.

7.2.4.1.3.3 Allgemeine Überlegungen zur Implementierung

Wenn die Prüfbedingungen derzeit auf Anwendungsebene im Programmcode hinterlegt sind, muss eine sorgfältige Planung erfolgen, um die Implementierung auf die Datenbankebene zu verlagern.

Der wichtigste Punkt ist der aktuelle Status der Datenbank zum Zeitpunkt der Implementierung der RI-Regeln. Wenn die Datenbasis korrupt ist, also wenn es zu einigen Child-Sätzen aktuell keine Parent-Sätze gibt, kann es schwierig werden RI-Bedingungen zu implementieren. Als Entwickler sollten Sie unbedingt vor der Implementierung SQL, den Query Manager oder andere Dienstprogramme einsetzen, um Datenfehler zu beheben. IBM hat hiefür schon vor einiger Zeit ein sehr nützliches JOIN-Statement implementiert: EXCEPTION JOIN. Mit diesem JOIN Statement lassen sich Datenintegritätsprobleme einfach und schnell lokalisieren:

LEFT EXCEPTION JOIN

Das SQL-Statement in der oberen Abbildung verbindet die beiden Tabellen „POSOrder“ und „HDROrder“ mit LEFT EXCEPTION JOIN über das entsprechende Schlüsselfeld. So erhalten Sie alle Sätze der links vom JOIN stehenden Tabelle. Das sind Sätze der Tabelle „PosOrder“, zu denen keine Übereinstimmungen in der zweiten Tabelle („HDROrder“) gefunden worden sind. Die Abfrage ermöglicht es, sich schnell und mit minimalem Aufwand einen Überblick über die momentane Datenlage zu verschaffen. Erst wenn Sie sicherstellen können, dass die Datenintegrität gewährleistet ist, können Sie die RI-Regeln implementieren. Wenn Sie feststellen, dass die Integrität Ihrer Datenbank an vielen Stellen verletzt ist, könnte dies ein Anlass sein, einmal grundlegend über die bisherige Strategie nachzudenken und vielleicht ein Grund mehr, die Integritätsprüfungen auf die Datenbankebene zu verlagern!

Außerdem müssen Sie natürlich die Auswirkungen von referentiellen Integritätsbedingungen auf die allgemeinen Grundsätze der Dateiverwaltung berücksichtigen. Das heißt: Die Regeln legen fest, wie Daten in abhängigen Dateien manipuliert werden dürfen. Sie müssen also Ihre Programme dahingehend prüfen, ob Sie in den Programmen in der richtigen Reihenfolge

Datensätze löschen, aktualisieren oder einfügen. Hier müssen die Programmcodes gegebenenfalls angepasst werden. Falls es nicht möglich ist, die entsprechenden Programme anzupassen, weil Massen-Updates oder -Inserts nachts laufen und weil Sie darauf keinen Einfluss haben, können Sie die Regeln auch vorübergehend deaktivieren. Sie führen Datenmanipulationen durch und aktivieren abschließend wieder die RI-Bedingungen – vorausgesetzt die Datenintegrität wird durch die Aktionen nicht gefährdet!

Vermutlich ist es das Beste, die Algorithmen in den Programmen sukzessive zu entfernen. Gegebenenfalls können Sie in den bestehenden Anwendungen auch gänzlich auf das Entfernen der Überprüfungen verzichten, wenn die Programme nicht grundsätzlich gegen die RI-Regeln verstoßen. Dadurch könnten Sie zusätzlichen Zeitaufwand vermeiden der entsteht, wenn Sie alle Anwendungen ändern, kompilieren und testen. Der Nachteil dieser Methode wird dadurch verursacht, dass das System die Integritätsbedingungen doppelt prüft – sowohl auf Anwendungs- als auch auf Datenbankebene.

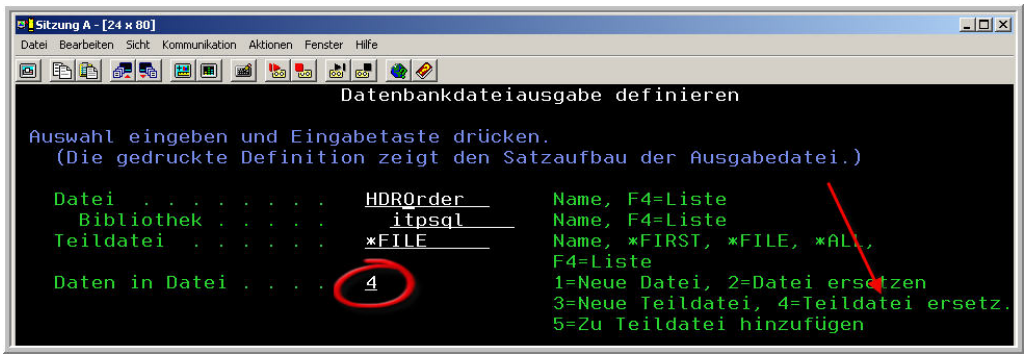
7.2.4.1.3.4 Überlegungen zur Performance

Referentielle Datenintegrität ist eine komplexe und hochentwickelte Funktion in der DB2/UDB for iSeries. Sie kann minimale Auswirkungen auf die Leistung haben, kann aber auch erhebliche Auswirkungen mit sich bringen, wenn zum Beispiel das Löschen eines Datensatzes in einer Parent-Tabelle ein cascadierendes Delete mit einer großen Anzahl übereinstimmender Sätze in der oder den abhängigen Dateien auslöst. Die Auswirkung von RI-Bedingungen sollte minimal sein, um die auf Anwendungsebene implementierten Funktionen zu ersetzten. Wenn hingegen zusätzliche Funktionen auf die Datenbankebene verlagert werden, erfordert dies möglicherweise zusätzliche Systemressourcen; es könnte sich auch durchaus negativ auf die Systemleistung auswirken. Delete Cascade, Delete Set NULL und Delete Set Default können diese negativen Auswirkungen auf die Leistung haben – abhängig davon, wie tief die cascadierenden Operationen greifen und wie viele abhängige Child-Sätze betroffen sind.

Bedenken Sie, dass auch beim Hinzufügen eines UNIQUE Keys, PRIMARY Keys oder referentiellen Constraints ein Zugriffspfadschlüssel für die entsprechende Tabelle erstellt wird. Wenn die Tabelle viele Zeilen enthält, kann die Erstellung und Überprüfung ebenfalls sehr viel Zeit in Anspruch nehmen. Normalerweise sind dies einmalige Vorgänge. Doch ich sehe immer wieder Techniken, bei denen nachts mithilfe von Query physische Dateien neu erstellt werden, um dann in einem zweiten Schritt die abhängigen logischen Dateien zu erstellen.

Ergänzung 4/2008

7.2.4
Seite 34



Physische Datei mit Query erstellen

Mal abgesehen davon, dass ich kein Query-Freund bin, geht es bei der oben beschriebenen Problematik vorrangig darum, zunächst alle bestehenden Sätze der physischen Datei zu löschen, um dann die neuen Daten hinzuzufügen. Häufig wird zur Lösung des Problems die Option „2=Datei ersetzen“ verwendet. Das gleiche Ergebnis erzielen Sie, wenn Sie an Stelle der Option „2“ die Option „4=Teildatei ersetzen“ wählen. Dadurch wird nicht die gesamte Datei gelöscht, sondern nur die Teildatei. Alle logischen Dateien können bestehen bleiben. Letztendlich kann aber auch dieser Prozess viel Zeit in Anspruch nehmen, da die abhängigen Zugriffspfade auch jetzt gepflegt werden müssen. Vielleicht finden Sie eine Möglichkeit, nicht alle, sondern nur die neuen relevanten Sätze zu übernehmen und/oder zu aktualisieren.

Eine Kombination von RI-Bedingungen mit Triggern, die wir uns im nächsten Beitrag genauer anschauen werden, sollten Sie auch sehr vorsichtig nutzen. Es ist wichtig zunächst die Auswirkungen jeder einzelnen Komponente auf die Systemleistung zu kennen, um lange Schleifen oder gar Endlosschleifen zu vermeiden. Erst dann lassen sich referentielle Integritätsbedingungen sinnvoll mit Datenbank-Triggern kombinieren.

Eine gute Planung der Constraints auf Grundlage Ihrer Datenbank ist somit Voraussetzung dafür, dass die Systemressourcen intelligent genutzt werden können. Erst wenn Sie die Funktionsweise der RIs genau kennen, lassen sich die Auswirkungen auf die Systemleistung abschätzen!

7.2.4.2 Datenbankrelationen analysieren

Wir können eine Vielzahl verschiedener Datenbankobjekte erzeugen: Tabellen, Indizes, Views aber auch Integritätsbedingungen sind Bestandteil einer modernen Datenbank. Meine Datenbank ist verhältnismäßig klein und es sollte mir gelingen, den Überblick zu behalten. Diese Vermutung äußern wahrscheinlich viele Datenbankadministratoren zu Beginn ihrer Arbeit. Doch dann wird die Datenbank erweitert und modifiziert. Sie wächst und verändert sich und während ich noch darüber nachdenke, bin auch ich mir nicht mehr sicher, ob ich nach vielen Monaten oder gar in einigen Jahren noch alle Satzformate und Relationen auswendig kennen werde. Hinzu kommt, dass es Situationen geben kann, in denen Sie sich mit fremden Datenbanken auseinandersetzen müssen. Sie müssen ad hoc die Abhängigkeiten verschiedener Tabellen ermitteln, wollen aber kein neues Tool kaufen. Keine Sorge! Das Betriebssystem bietet einige Möglichkeiten, um Datenbankrelationen zu analysieren und natürlich bieten auch die modernen, grafischen Front-End-Tools verschiedene Möglichkeiten. Es ist, glaube ich, an der Zeit, sich mit den entsprechenden CL-Befehlen und natürlich mit der grafischen Darstellung zu befassen.

Beginnen wir zunächst ganz klassisch, indem wir einen i5/OS-Befehl nutzen, und schauen uns dann in einem zweiten Schritt eine Funktion des System i Navigators an, die wenig genutzt wird, obwohl sie gute Möglichkeiten bietet, bestehende Datenbanken zu analysieren.

Ergänzung 12/2010

7.2.4.2.1 Der Befehl „Display database relation (DSPDBR)“

Sie können den Befehl „Display database relation (DSPDBR)“ verwenden, um die Beziehungen zwischen mehreren Dateien Ihres System zu ermitteln. Die Informationen können angezeigt, gedruckt oder in eine Datei ausgegeben werden, die dann natürlich anschließend auch mittels SQL ausgewertet werden kann.

Schauen wir uns hierzu ein einfaches Beispiel an: Ich möchte zunächst alle abhängigen Objekte der SQL-Tabelle „Master“ im Schema „ITPSQL“ ermitteln:

Der Befehl DSPDBR

Der Befehl bietet kaum Möglichkeiten. Sie können – wie ich es getan habe – eine physische Datei oder SQL-Tabelle eintragen, drücken die „Datenfreigabe“ und die Daten werden anschließend auf dem Bildschirm angezeigt.

Alternativ können Sie auch die Selektion einschränken, indem Sie nur die Objekte anzeigen lassen, die dasselbe Satzformat verwenden oder die eine Abhängigkeit für eine bestimmte Teildatei aufweisen. Meine Tabelle „Master“ ist eine SQL-Tabelle und hat weder mehrere Member noch wurde ein Satzformatname vereinbart, so dass die Möglichkeiten für mein Beispiel irrelevant sind. Das Ergebnis sieht anschließend folgendermaßen aus:

```
  9.11.09              Datenbankrelationen anzeigen
DSPDBR Befehlseingabe
  Datei . . . . . . . . . . . . . . . . . . . : FILE       MASTER
    Bibliothek  . . . . . . . . . . . . . . . :            ITPSQL
  Teildatei . . . . . . . . . . . . . . . . . : MBR        *NONE
  Satzformat  . . . . . . . . . . . . . . . . : RCDFMT     *NONE
  Ausgabe . . . . . . . . . . . . . . . . . . : OUTPUT     *
Spezifikationen
  Dateiart  . . . . . . . . . . . . . . . . . :            Physisch
  Datei . . . . . . . . . . . . . . . . . . . :            MASTER
    Bibliothek  . . . . . . . . . . . . . . . :            ITPSQL
    Teildatei . . . . . . . . . . . . . . . . :            *NONE
    Satzformat  . . . . . . . . . . . . . . . :            *NONE
    Anzahl abhängiger Dateien . . . . . . . . :               3
Von angegebener Datei abhängige Dateien
  Abhängige Datei        Bibliothek    Abhängigkeit JREF    Integritätsbed.
     I_MASTER            ITPSQL        Daten
     D_MASTER            ITPSQL        Daten
     DETAIL              ITPSQL        Integritätsb.        FK_DETAIL_MASTER_MAKURZ
```

Wirkungsweise des Befehls DSPDBR

Zunächst werden eine Reihe allgemeiner Informationen über die Tabelle „Master“ angezeigt. Dann folgen die abhängigen Objekte: Einerseits existieren logische Dateien für die Tabelle „Master“, andererseits wurde eine referentielle Integritätsbedingung definiert. Schauen Sie sich die Abbildung nochmals genau an: Viele Informationen fehlen. So können wir nicht erkennen, ob und welcher Zugriffspfad über die logische Datei „I_Master“ definiert wird. Wir können nicht einmal erkennen, ob es sich um einen SQL-Index, eine SQL-View oder eine logische DDS-Datei handelt. Auch für die referentielle Integritätsbedingung werden keine nützlichen Informationen angezeigt. Wenn ich nicht in den referentiellen Bedingungen „sprechende“ Namen verwenden würde, könnten wir weder die CHILD-Tabelle identifizieren noch wüssten wir welche Spalte für die Bedingung verwendet wurde. Die Anzeige enthält wenige Informationen und ist in der heutigen Zeit nicht mehr besonders nützlich.

7.2.4
Seite 38

Dennoch möchte ich den Befehl ein zweites Mal verwenden und stelle die Informationen dieses Mal in eine Datei:

```
                    Datenbankrelationen anzeigen (DSPDBR)

Auswahl eingeben und Eingabetaste drücken.

Datei  . . . . . . . . . . . . . . > MASTER      Name, generisch*, *ALL
  Bibliothek . . . . . . . . . . .   ITPSQL      Name, *LIBL, *CURLIB...
Ausgabe  . . . . . . . . . . . . . > *OUTFILE    *, *PRINT, *OUTFILE
Datei für Ausgabe  . . . . . . . .   master_DBR   Name
  Bibliothek . . . . . . . . . . .   itpsql       Name, *LIBL, *CURLIB
Auswahl für Ausgabeteildatei:
  Teildatei für Ausgabe  . . . .     *FIRST       Name, *FIRST
  Sätze ersetzen od. hinzufügen      *REPLACE     *REPLACE, *ADD
```

*Der Befehl DSPDBR –Ausgabe *Outfile*

Nun wollen wir den Inhalt der Datei analysieren; vielleicht erhalten wir auf diesem Weg detaillierte Informationen über die abhängigen Objekte:

```
                    Physische Teildatei anzeigen
Datei  . . . . . :   MASTER_DBR          Bibliothek . . . :   ITPSQL
Teildatei  . . . :   MASTER_DBR          Satz . . . . . :   1
Steuerung  . . . :                       Spalte . . . . . :   1
Suchen . . . . . :
*...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+....0....+....1....+....2....+....3
PMASTER    ITPSQL    *NONE     *NONE     000031091109224913I_MASTER  ITPSQL          D0000EPSSRV
PMASTER    ITPSQL    *NONE     *NONE     000031091109224913D_MASTER  ITPSQL          D0000EPSSRV
PMASTER    ITPSQL    *NONE     *NONE     000031091109224913DETAIL    ITPSQL          C0000EPSSRV  ITPSQL      FK_DETAIL_MASTER
           ****** DATENENDE ******
```

Inhalt der Datei MASTER_DBR

Das Ergebnis ist auf den ersten Blick enttäuschend. Einige der Daten sind nicht lesbar, weil Sie in gepacktem oder binärem Format gespeichert wurden, und auch die Zahlenkolonnen sind wenig sprechend, wenn man deren Bedeutung nicht kennt.

Versuchen wir also unser Glück mit SQL:

SQL- Ergebnis für das Outfile „Master_DBR“

Wirklich befriedigend sind auch diese Informationen nicht. Wir erhalten jetzt zwar Spaltenüberschriften, nur die Bedeutung der Spalten bleibt ein Rätsel. Um das Rätsel zu lösen, benötigen Sie zunächst einmal die Datenbeschreibung des *OUTFILES. Am einfachsten erfahren wir die Bedeutung der Spalten, indem wir den CL-Befehl „Display file field description (DSPFFD)“ nutzen:

```
WHDTM      CHAR          13      13        47        Beides   Abrufdatum-
                                                               und Uhrzeit
  Feldtext . . . . . . . . . . . . . . . . :  Abrufdatum: Jahrhundert/Datum/Zeit
  ID des codierten Zeichensatzes . . . . . :    273
WHREFI     CHAR          10      10        60        Beides   Abhängige Datei
  Feldtext . . . . . . . . . . . . . . . . :  Abhängige Datei
  ID des codierten Zeichensatzes . . . . . :    273
WHRELI     CHAR          10      10        70        Beides   Abhängige Bibliothek
  Feldtext . . . . . . . . . . . . . . . . :  Abhängige Bibliothek
  ID des codierten Zeichensatzes . . . . . :    273
WHREMB     CHAR          10      10        80        Beides   Abhängige Teildatei
  Feldtext . . . . . . . . . . . . . . . . :  Abhängige Teildatei
  ID des codierten Zeichensatzes . . . . . :    273
WHTYPE     CHAR           1       1        90        Beides   Art der Abhängigkeit
  Feldtext . . . . . . . . . . . . . . . . :  D=Daten,I=Zugr.pfad,O=Zugr.pfadeigner,V=SQL VIEW,C
  ID des codierten Zeichensatzes . . . . . :    273
```

Ergebnis des Befehls „Display file field description (DSPFFD)“

Nun kennen wir die Bedeutung der einzelnen Spalten und es werden sogar die verwendeten Codes näher erläutert. Allerdings sollten wir uns nochmals unsere Ausgangsfrage in Erinnerung rufen: Wir wollten lediglich detaillierte Informationen über die abhängigen Objekte der SQL-Tabelle „Master“ erhalten. Viel Aufwand, wenn Sie mich fragen, und die Informationen sind nach wie vor unzureichend. Diese Befehle haben lange Zeit gute Arbeit geleistet, wenn logische DDS-Dateien ermittelt werden mussten, um beispielsweise physische Dateien zu löschen. Heute benötigen wir Tools, die mehr Selektionen erlauben und detaillierte Informationen liefern.

7.2.4.2.2 Database Navigator-Masken einsetzen

Database Navigator-Masken ermöglichen die grafische Darstellung der Beziehungen und Abhängigkeiten zwischen Datenbankobjekten auf einem System i Server. Dabei handelt es sich im Wesentlichen um eine „Momentaufnahme“ der Datenbank mit den verschiedenen Beziehungen und Abhängigkeiten, die zwischen allen in der Maske enthaltenen Objekten bestehen. So können Sie mit Database Navigator die komplexen Beziehungen Ihrer Datenbankobjekte problemlos anhand einer grafischen Darstellung untersuchen. Die Grafik zeigt Ihnen:

- die verschiedenen Tabellen in der Datenbank,
- die Beziehungen der Tabellen zueinander,
- die den Tabellen zugeordneten Indizes

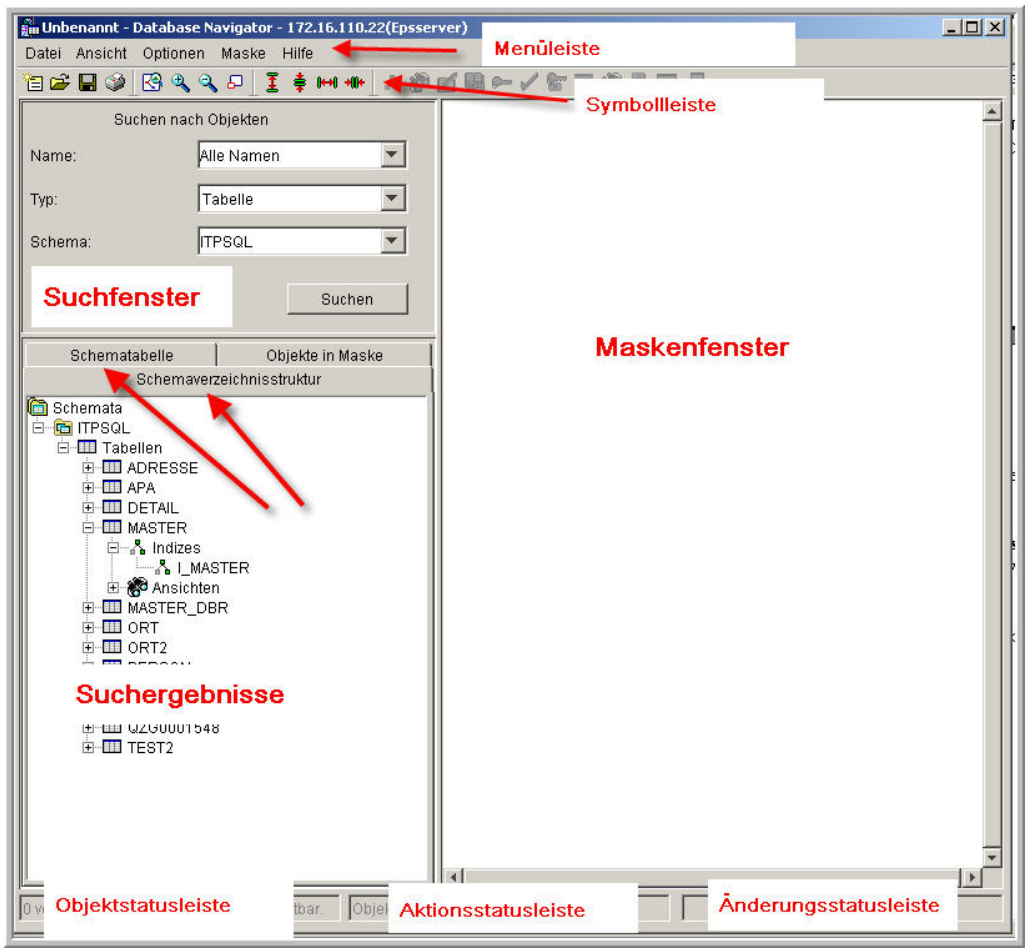
und die Einschränkungen (Integritätsbedingungen).

Sie können den Database Navigator aber nicht nur für die Analyse Ihrer Datenbank verwenden, sondern können die Maske auch manipulieren, um bestimmte Elemente daraus anzuzeigen, ohne dass dabei Änderungen an Objekten im System vorgenommen werden. Außerdem können Sie für alle angezeigten Objekte den entsprechenden SQL-Code generieren und natürlich können Sie die Masken auch drucken und speichern, um später entsprechende Vergleiche durchführen zu können.

Lassen Sie uns nun eine Database Maske für die Tabelle „Master“ erstellen, um die Unterschiede zum Befehl „Display database relation (DSPDBR)“ zu verdeutlichen. Öffnen Sie zu diesem Zweck zunächst die Verbindung zu Ihrem System i Server. Anschließend öffnen Sie die Datenbank und markieren mit der rechten Maustaste den Menüpunkt „Database Navigator-Masken“. Wählen Sie – wie in der folgenden Abbildung dargestellt – anschließend den Eintrag „Neu ➜ Maske“.

Eine neue Datenbankmaske erstellen

Jetzt sollte sich der folgende Dialog öffnen:



Elemente der Database Navigator-Maske

Schauen wir uns als erstes die verschiedenen Elemente dieses Dialoges an:

Der primäre Arbeitsbereich des Database Navigator ist ein in mehrere Hauptbereiche unterteiltes Fenster. Das Suchfenster steht links und dient zum Lokalisieren der Objekte, die Sie in die neue Maske aufnehmen möchten. Dabei stellt der obere Bildschirmbereich eine Suchfunktion zur Verfügung, damit Sie die in die Maske aufzunehmenden Objekte schnell finden. Die Ergebnisse der Suchoperation werden im rechten Bildschirmfenster unterhalb der Indexzungen „Schemaverzeichnisstruktur“ und „Schematabelle“ angezeigt.

7.2.4
Seite 42

Bevor wir unsere Datei „Master“ auswählen, wollen wir zunächst die weiteren Elemente des Fensters kurz untersuchen. Den rechten Bildschirmteil im Database Navigator-Fenster nennt man Maskenfenster. Das Maskenfenster enthält später die grafische Darstellung der Datenbankobjekte, ihre Beziehungen und die Abhängigkeiten. Für die Bearbeitung stehen verschiedene Funktionen zur Verfügung:

- Es können jederzeit weitere Tabellen und Ansichten in die Maske hinzugefügt werden.
- Natürlich können Objekte auch aus der Maske entfernt werden.
- Die Positionen von Objekten und deren dargestellte Größe kann angepasst werden.
- Es können Änderungen an den Objekten in der Maske vorgenommen werden.
- Für alle Objekte in der Maske kann der entsprechende SQL-Code generiert werden, der zur Erstellung der Objekte benötigt wird.

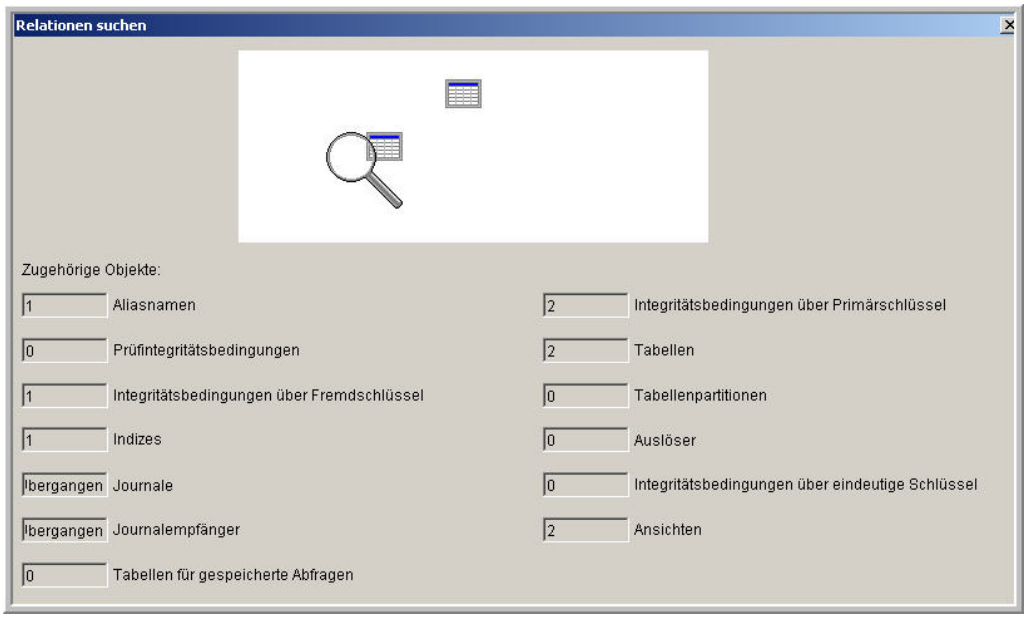
Die Objektstatusleiste unten links im Database Navigator-Fenster zeigt die Anzahl sichtbarer und auswählbarer Objekte in der Maske.

Die Aktionsstatusleiste unten in der Mitte des Database Navigator-Fensters bietet eine eindeutige Beschreibung der Vorgänge in der Maske sowie Angaben darüber, ob Änderungen anstehen.

Die Änderungsstatusleiste gibt Auskunft darüber, ob eine Änderung vorgenommen wurde oder ansteht.

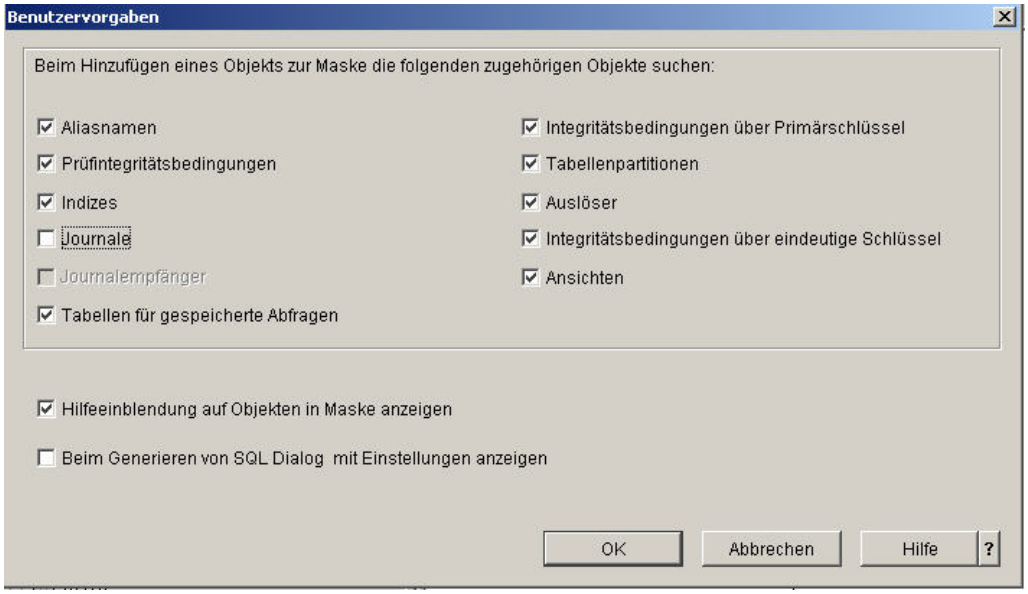
Die Menü- und Symbolleiste stellt weitere Funktionen für die Darstellung der Maske und die Objektbearbeitung zur Verfügung.

Dies sollte als kurzer Überblick ausreichend sein. Ich wähle nun die Tabelle „Master“ für die Datenbankmaske aus. Dies ist erst möglich, wenn Suchergebnisse im Suchfenster angezeigt werden. Sie können die Objekte zur Maske hinzufügen, indem Sie entweder mit der rechten Maustaste (Kontextmenü) auf ein Objekt klicken und „Zu Maske hinzufügen“ auswählen oder indem Sie auf den Objektnamen doppelt klicken. Sobald Sie ein Objekt ausgewählt haben, erscheint eine Maske, die Informationen darüber enthält, welche Objekttypen vom ausgewählten Objekt abhängig sind.



Abhängige Objekte suchen

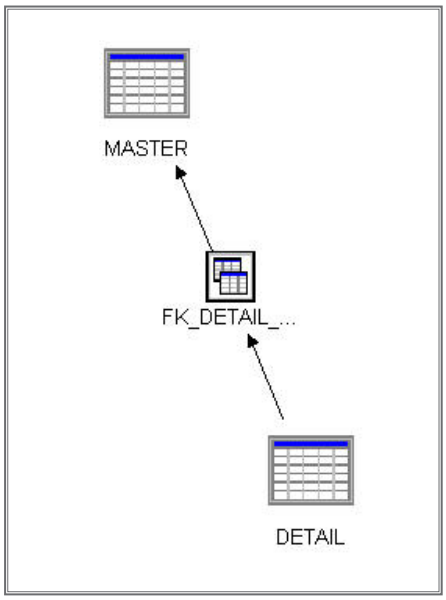
Sie sehen, es wird nicht nur nach Integritätsbedingungen und logischen Dateien gesucht, sondern auch ALIAS-Namen, Journale, MQTs, Trigger und viele weitere Objekte werden berücksichtigt. Ist die Zahl der abhängigen Objekte groß, dauert der Suchvorgang entsprechend lange. In der Menüleiste des Database Navigator-Fensters finden Sie den Menüpunkt „Optionen ➜ Benutzervorgaben“.



Benutzervorgaben definieren den Suchvorgang

Über die Benutzervorgaben entscheiden Sie, welche abhängigen Objekte in das Maskenfenster übernommen werden. Vielleicht ist es sinnvoll, einige Objekte von der Suche auszuschließen, weil Sie für die Analyse irrelevant sind und den Suchvorgang nur unnötig in die Länge ziehen, dann können Sie in diesem Fenster die Objekte „abwählen“, indem Sie den Haken vor dem entsprechenden Objekttypen entfernen.

Mein Suchergebnis wird mittlerweile im Maskenfenster angezeigt und ist recht überschaubar, wie Sie sehen können:



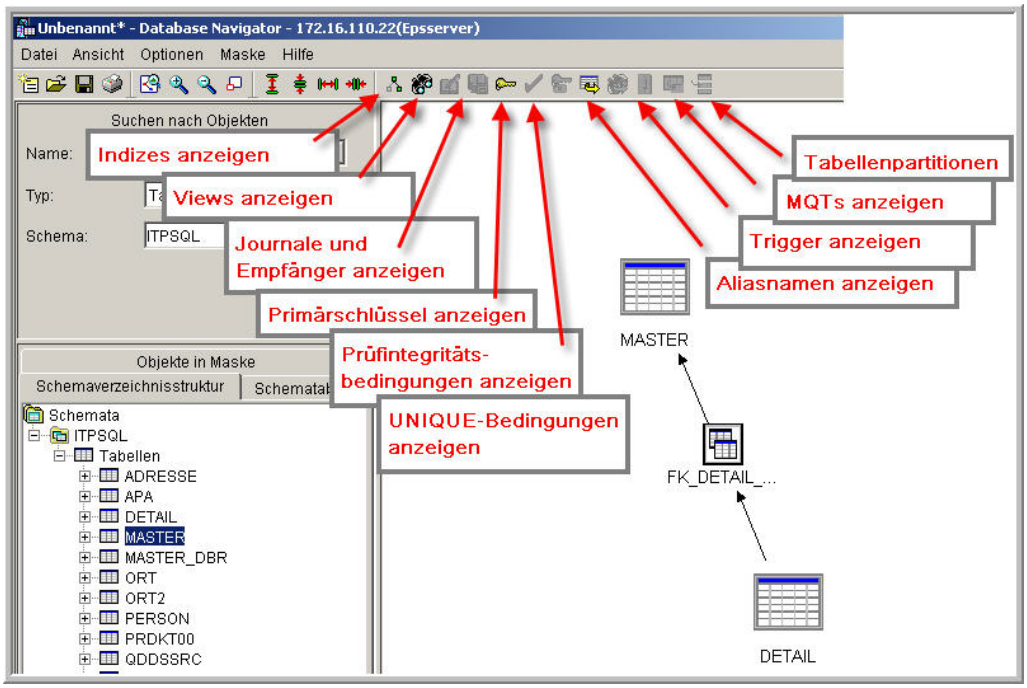
Die grafische Darstellung

Für jedes abhängige Datenbankobjekt existiert ein Icon. Sobald Sie den Cursor auf eines der Icon positionieren, erhalten Sie detaillierte Informationen zum entsprechenden Objekt. Zudem stehen viele Funktionen über die Kontexttaste zur Verfügung, die Sie für die Bearbeitung des jeweiligen Objektes nutzen können.

Vielfältige Informationen und zahlreiche Funktionen stehen zur Verfügung

Erinnern Sie sich noch an den CL-Befehl „Display database relation (DSPDBR)“? Wir hatten keine Informationen über die in Beziehung stehenden Tabellen. Hier erhalten wir alle benötigten Informationen und eine Reihe von Funktionen, die es uns erlauben, die Bedingung zu deaktivieren, zu löschen oder einfach aus der Maske zu entfernen. Für Tabellen und Indizes stehen entsprechend weitere Funktionen zur Verfügung.

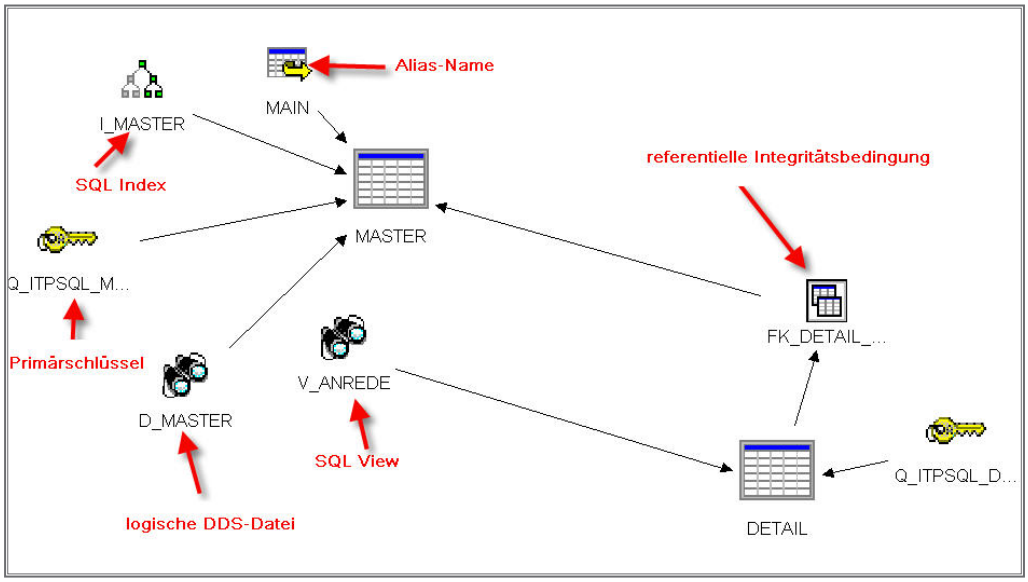
Außerdem können wir die grafische Anzeige filtern. Schauen Sie sich einmal in Ruhe die Aktionsleiste an. Sie erlaubt es, Datenbankobjekte in der Grafik anzuzeigen oder auszublenden, wenn die Informationen zu unübersichtlich werden:



Objekte für die Anzeige auswählen

7.2.4
Seite 46

Ich blende jetzt weitere Objekte ein. Meine Grafik wird dadurch natürlich komplexer:



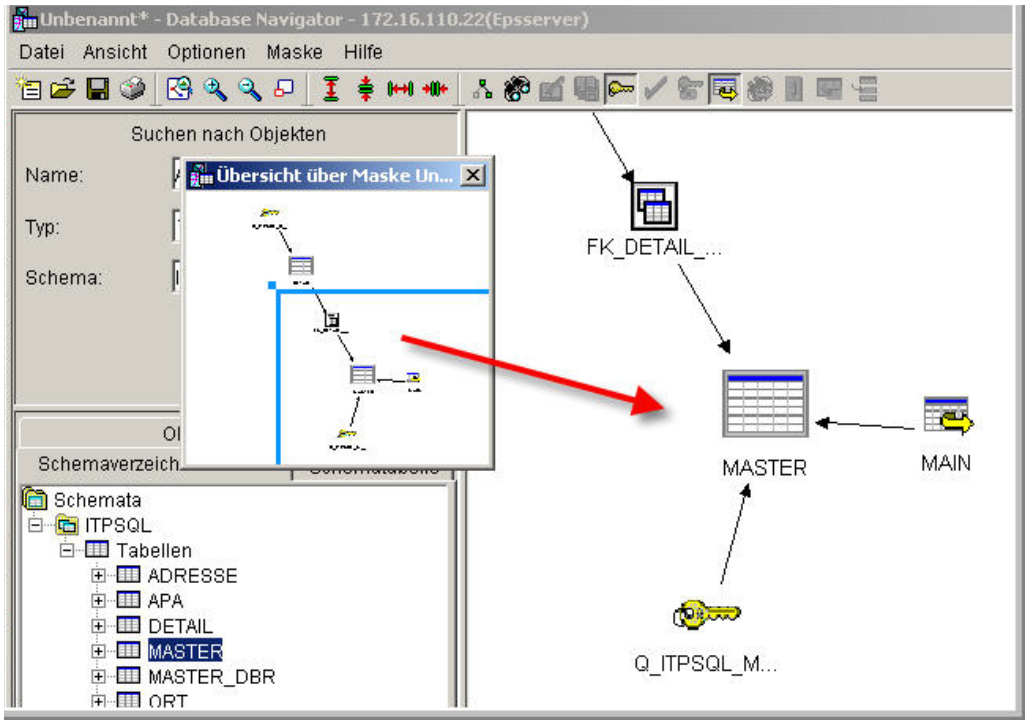
Alle abhängigen Datenbankobjekte anzeigen

Sie kennen die dargestellten Objekttypen bereits aus den vorangegangenen Kapiteln und wissen daher auch, dass unsere alten logischen DDS-Dateien als SQL-View dargestellt werden – auch wenn sie einen Zugriffspfad besitzen.

Je nach Anzahl der darzustellenden Objekte kann die grafische Darstellung schnell etwas klein ausfallen, wenn alle abhängigen Objekte angezeigt werden. Sie können aber mit Hilfe eines Übersichtsfensters leicht einzelne Bereiche der Grafik ansteuern. Dazu öffnen Sie das Menü „Ansicht“ und klicken auf die Option „Übersichtsfenster anzeigen“. In einer separaten Anzeige erscheint dann folgendes Fenster:

Das Übersichtsfenster

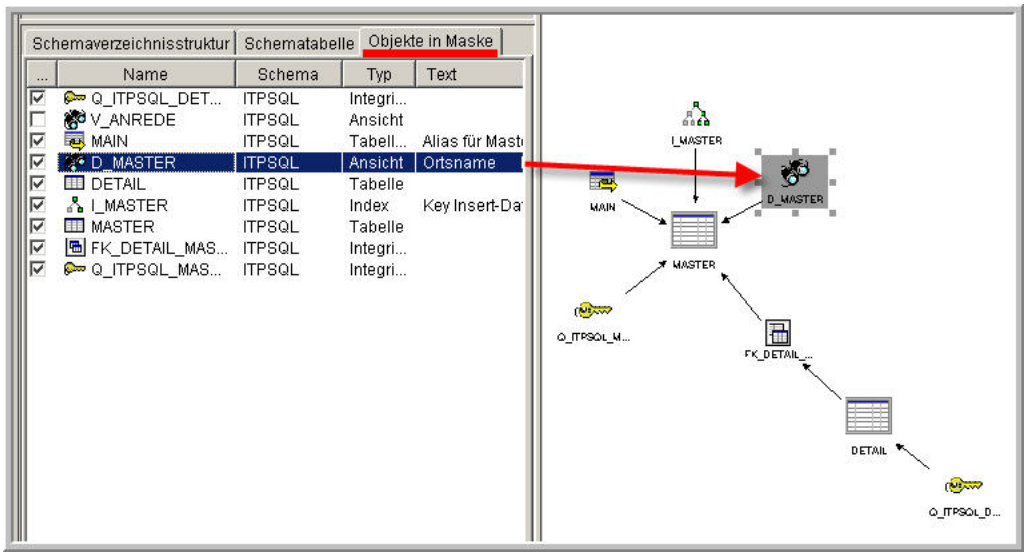
Bewegen Sie nun die Maus im Bereich dieses Übersichtsfensters. Der Mauszeiger wird sich in eine kleine Lupe verwandeln; damit kann ein rechteckiger Bereich in der Übersicht aufgezogen werden. Die Inhalte des Bereichs werden im rechten Teil synchronisiert in die Datenbank-Maske eingepasst. So sind Sie in der Lage, recht komfortabel in den Diagrammen und an interessanten Positionen zu navigieren.



Navigation und Ansicht mit der Lupenfunktion

Wenn man Objekte in einem sehr umfangreichen Diagramm auffinden möchte, kann das mit dem Übersichtsfenster oder mit dem reinen Suchen in der Grafik sehr aufwändig und zeitraubend sein. Aber es gibt eine andere Möglichkeit, ein bestimmtes Objekt innerhalb der Grafik ausfindig zu machen. Schauen Sie sich noch einmal das Suchfenster an. Dort finden Sie die Indexzunge „Objekte in Maske“ bzw. die Indexzunge „Schemaverzeichnisstruktur“. Beide Anzeigen sind hilfreich bei der Suche nach bestimmten Objekten.

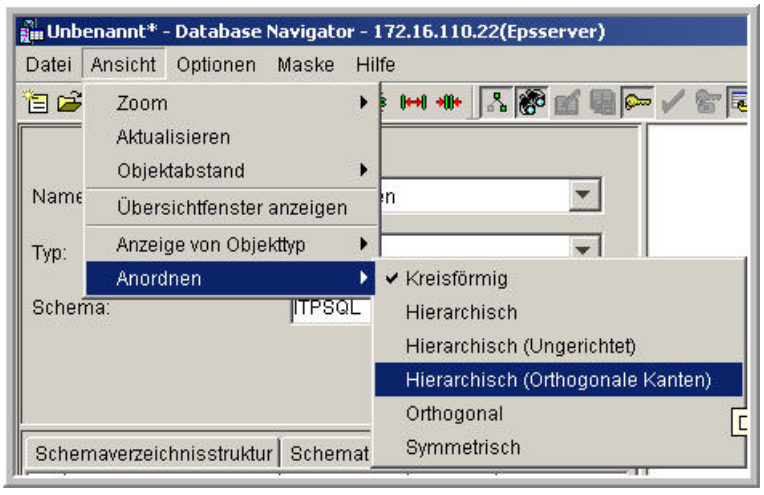
7.2.4
Seite 48



Objekte in der Grafik suchen

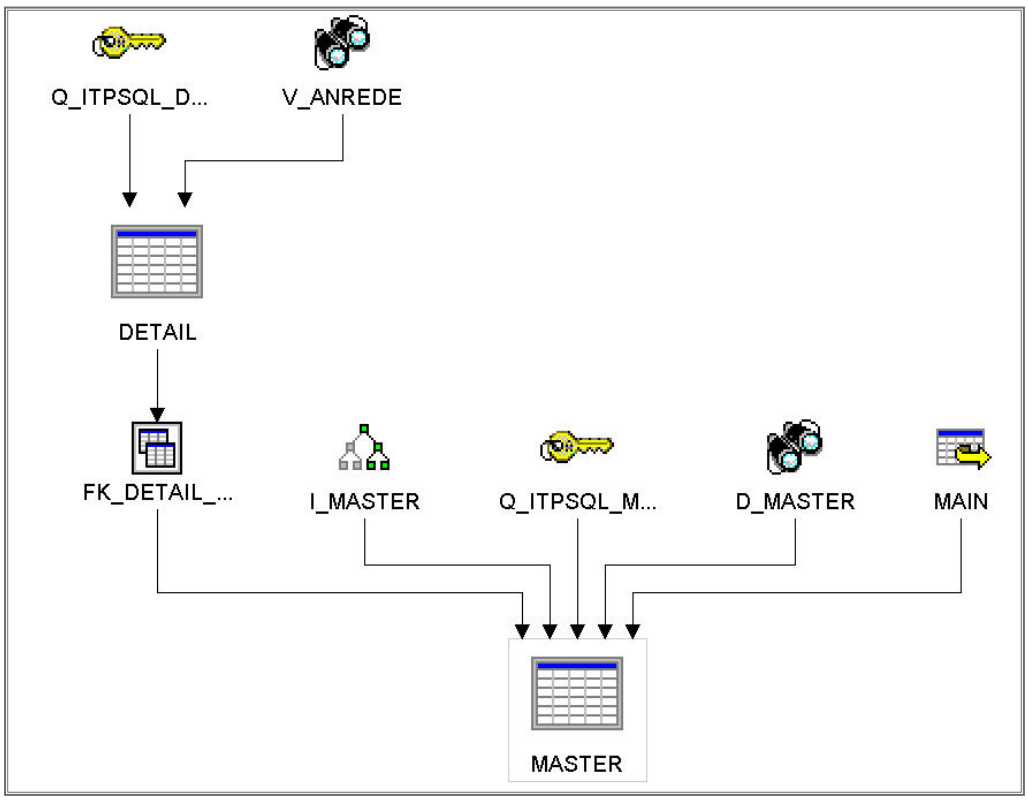
Wenn Sie in der Indexzunge eines der Objekte anklicken, wird das betroffene Objekt sofort im Diagramm selektiert – auch im größten Diagramm. Zum Sortieren der Objekte in der Anzeige „Objekte in Maske“ klicken Sie einfach auf den entsprechenden Spaltenkopf. Außerdem können Sie über diese Anzeige einzelne Objekte im Diagramm ausblenden, wenn Sie mit den kleinen Auswahlboxen in der linken Spalte arbeiten.

Die Anordnung der Objekte im Fenster wirkt zum Teil ein wenig unmotiviert. Dem können Sie abhelfen, indem Sie eine andere Darstellung wählen. Hierzu öffnen Sie den Menüpunkt „Ansicht ➜ Anordnen“ wie in der folgenden Grafik gezeigt:



Objektanordnung wählen

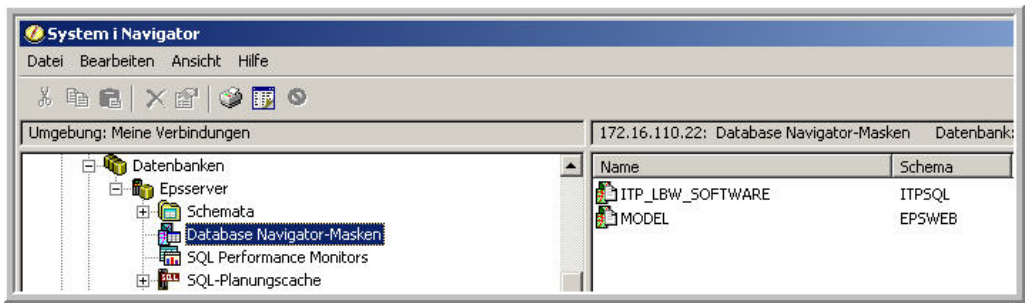
Es stehen verschiedene Auswahlmöglichkeiten zur Verfügung. Probieren Sie einfach aus, welche Darstellung Ihren Anforderungen genügt. Ich verwende gern die Anordnung „Hierarchisch (Orthogonale Kanten)“. Die Anzeige verändert sich daraufhin in der folgenden Weise:



Hierarchische Anordnung – Orthogonale Kanten

Sie sehen, das grafische Tool des System i Navigators bietet eine Vielzahl an Optionen, um die Anzeige Ihren Bedürfnissen anzupassen. Und natürlich können Sie die erstellten Datenbankmasken auch speichern und zu einem späteren Zeitpunkt erneut öffnen.

Zum Speichern wählen Sie den Menüpunkt „Datei ➜ Datei speichern“. Die Database Navigator-Masken finden Sie anschließend im System i Navigator im Menüpunkt „Database Navigator-Masken“ wieder.



Die gespeicherten Database Navigator-Masken

7.2.4
Seite 50

Sie können die gespeicherten Modelle mit einem Doppelklick jederzeit wieder öffnen. Bedenken Sie hierbei aber bitte, dass es sich bei den Masken lediglich um Snapshots handelt, die die Datenbankabhängigkeiten zum Zeitpunkt der Erstellung repräsentieren. Falls sich in der Zwischenzeit Ihre Datenbankobjekte verändert haben, müssen Sie die Anzeige aktualisieren, indem Sie im geöffneten Maskenfenster den Menüpunkt „Ansicht“ öffnen:

Database Navigator-Maske aktualisieren

7.3 Data Definition Language (DDL)

Im vorangegangenen Kapitel haben wir unser Datenbankkonzept entworfen. Jetzt müssen wir diesen Entwurf in entsprechende Datenbankobjekte umsetzen. Wir möchten unser Konzept in das interne Schema übersetzen. Wenn Sie SQL noch nicht oder nur wenig kennen, ist die erste Hürde, die Sie nehmen müssen, die Überwindung traditioneller Namenskonventionen. Die zweite Hürde besteht darin, dass SQL sowohl eine **Datendefinitionssprache** (**DDL**) als auch eine **Datenmanipulationssprache**(**DML**) ist.

Eine Definitionssprache ermöglicht die Definition von Tabellen, Indizes und Views – vergleichbar mit DDS. Eine Datenmanipulationssprache ermöglicht die Abfrage und die Modifizierung von Daten. Solch eine Sprache ist uns im traditionellen Umfeld eher unbekannt, da der Datenzugriff vorwiegend über HLL-Programme erfolgt.

Es geht somit im folgenden Kapitel nicht nur um das Erstellen einer neuen Datenbank, sondern auch um das Kennenlernen des Unterschieds zu traditionellen Methoden.

Allerdings möchte ich Eines an dieser Stelle vorweg schicken: Um SQL zu verwenden, müssen Sie Ihre Datenbank nicht neu erstellen. Alle Vorteile, die SQL mit sich bringt, können Sie auch mit den bestehenden Datenbanken nutzen; auch Ihre RPG- oder COBOL-Programme können Sie weiter verwenden.

7.3
Seite 2

7.3.1 Relationale iSeries-Datenbanken

Der Begriff „Relationale Datenbank“ ist auf einem iSeries-System irreführend, da das gesamte System eine einzige Datenbank mit separaten Bibliotheken darstellt. Die meisten Menschen, die von einer iSeries-Datenbank sprechen, meinen damit jedoch einzelne Bibliotheken, die physische und logische Dateien enthalten.

Die Verwirrung wird noch verstärkt durch die SQL-Anweisung CREATE DATABASE. Tatsächlich erstellt die Anweisung keine Datenbank, sondern ebenfalls nur eine einzelne Bibliothek. Das gleiche Ergebnis erzielen Sie aber auch mit der SQL-Anweisung CREATE COLLECTION oder dem Befehl CREATE SCHEMA. Welcher Befehl ist denn nun der richtige? Und wie versteht Ihre iSeries den Begriff der relationalen Datenbank?

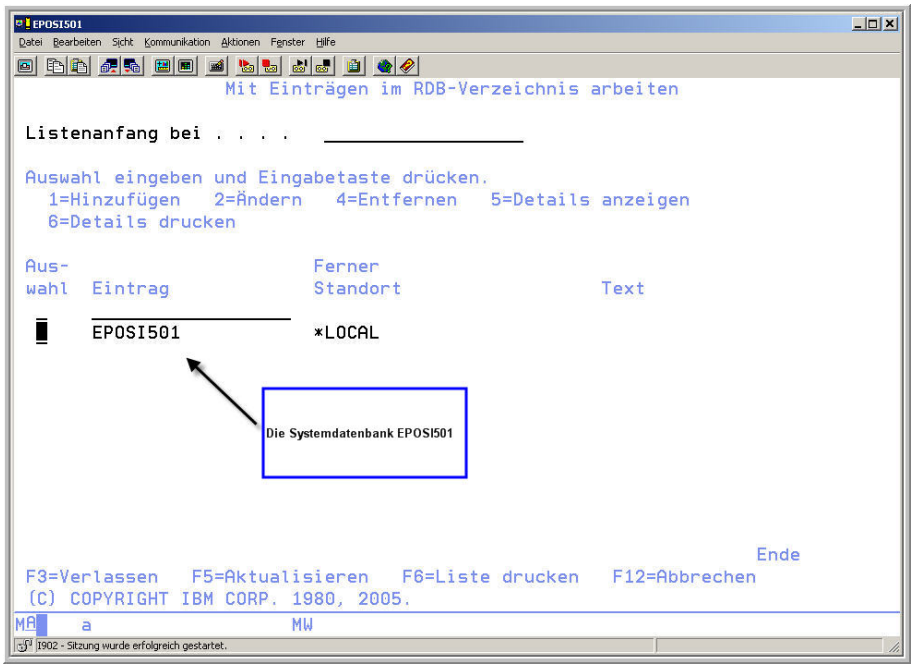
Um dies zu beantworten, müssen wir erneut die Frage aufgreifen: Was ist eine relationale Datenbank? Eine relationale Datenbank ist lediglich eine Sammlung verschiedener Tabellen und anderer Objekte, deren Daten miteinander logisch in Beziehung stehen. Nirgendwo steht geschrieben, dass die Daten sich in einer einzigen Bibliothek befinden müssen!

IBM unterscheidet auf einem iSeries-System drei verschiedene relationale Datenbanktypen:

System Relational Database (Systemdatenbank)

Auf jedem iSeries-System finden Sie eine Default-Datenbank. Diese relationale Systemdatenbank ist stets identisch mit Ihrem lokalen System. Sie enthält alle Datenbankobjekte, die auf Ihrer Festplatte existieren. Der Name der Systemdatenbank ist häufig identisch mit dem Systemnamen Ihrer iSeries. Er kann aber auch abweichen. Den Namen Ihrer Systemdatenbank finden Sie im Datenbankverzeichnis:

7.3.1
Seite 2



Datenbankverzeichnis

Prüfen Sie einfach mal mit dem Befehl WRKRDBDIRE (Work with Relational Database Directory Entry), wie Ihre Datenbank heißt.

User Relational Database

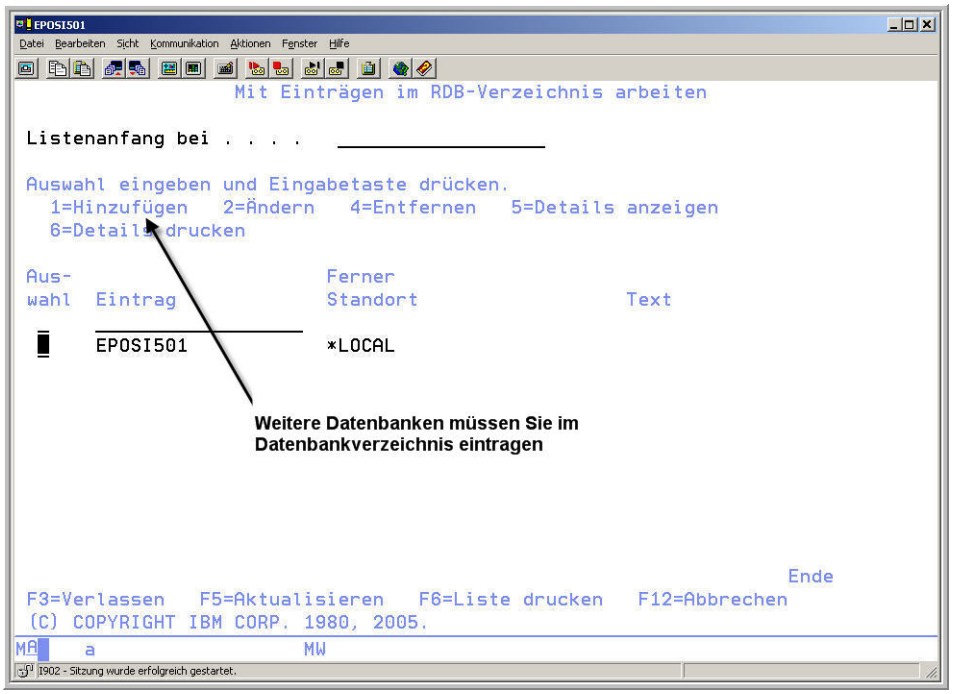
Sie erstellen Benutzerdatenbanken, wenn Sie mit unabhängigen ASPs arbeiten. Jeder ASP wird als eigenständige relationale Benutzerdatenbank verstanden. Der Name der benutzerdefinierten Datenbank entspricht dem Namen des ASP, es sei denn, Sie geben der Datenbank einen eigenständigen Namen. Eine solche benutzerdefinierte Datenbank enthält alle Datenbankobjekte, die sich im entsprechenden ASP befinden. Zusätzlich besteht eine logische Verbindung zu abhängigen Objekten der Systemdatenbank. Doch trotz dieser logischen Verbindung zwischen System- und Benutzerdatenbank gibt es ein paar Restriktionen, die Sie beachten müssen, wenn Sie ASPs einsetzen:

- Erstellen Sie Views und Indizes stets in dem Schema (=Bibliothek), in dem sich die Referenzobjekte befinden.
- Datenbank-Trigger und Datenbankbedingungen (Constraint) müssen Sie in dem ASP erstellen, in dem sich Ihre Tabellen befinden, wobei allerdings unterschiedliche Bibliotheken verwendet werden können.
- Falls Sie mit benutzerdefinierten Datentypen arbeiten (Distinct Types), müssen die erstellten Datentypen und die Tabellen, die diese Datentypen verwenden, im selben ASP liegen.
- Bei referentiellen Integritätsbedingungen müssen die Parent- und Child-Tabellen im gleichen ASP gespeichert sein.

Alle anderen Querverbindungen zwischen Systemdatenbank und Benutzerdatenbank sind erlaubt. So kann eine Prozedur, die in der Systemdatenbank liegt, durchaus Objekte der Benutzerdatenbank verwenden und umgekehrt.

Remote Relational Database

Mit relationalen Datenbanken anderer iSeries-Systeme oder fremder Systeme können Remote-Verbindungen hergestellt werden. Dafür muss die Remote-Datenbank mit dem Befehl ADDRDBDIRE (Add Relational Database Directory Entry) im Datenbankverzeichnis registriert sein. Dieses Verzeichnis enthält im Übrigen alle Datenbanken, mit denen Sie arbeiten können oder wollen.



Befehl WRKRDBDIRE

7.3.1
Seite 4

7.3.2 Schemata

Ein Schema dient innerhalb einer Datenbank (egal, ob es sich um System- oder Benutzerdatenbank handelt) nur der Objektorganisation – das heißt: Man verwendet eine unterschiedliche Anzahl Schemata, um die Datenbankobjekte logisch zu strukturieren. Ein Schema wird auch Collection oder Library genannt, wobei Ihnen vermutlich der Begriff der Library (=Bibliothek) am geläufigsten ist. Außerdem wird das Schema zur Qualifizierung der Objekte verwendet. Mittels des Schema-Namens können Sie die Datenbankobjekte im SQL-Statement adressieren. Auch die unqualifizierte Objektadressierung ist möglich. Sie sollte Ihnen bekannt sein! Im Hintergrund der unqualifizierten Adressierung sucht das System über die Bibliotheksliste die entsprechenden Referenzobjekte. Aber Vorsicht! Die Bibliothekssuchliste ist iSeries-spezifisch und in dieser Form anderen Systemen nicht bekannt. SQL kann zwar auch mithilfe der Suchliste auf die Tabellen zugreifen, doch hängt die Verwendung der Suchliste von unterschiedlichen Faktoren ab: Statisches SQL und dynamisches SQL qualifizieren unterschiedlich. Auch die verwendete Namenskonvention *SYS oder *SQL spielt eine entscheidende Rolle – und nicht zu letzt die verwendeten Front-End-Tools beziehungsweise die Konfiguration der Schnittstellen. Wenn Sie nicht Gefahr laufen wollen, versehentlich falsche Tabellen zu manipulieren, rate ich Ihnen, in den SQL-Statements Ihre Objekte zu qualifizieren. Andernfalls können schnell Probleme auftreten, mit denen Sie sich vielleicht sehr unbeliebt machen.

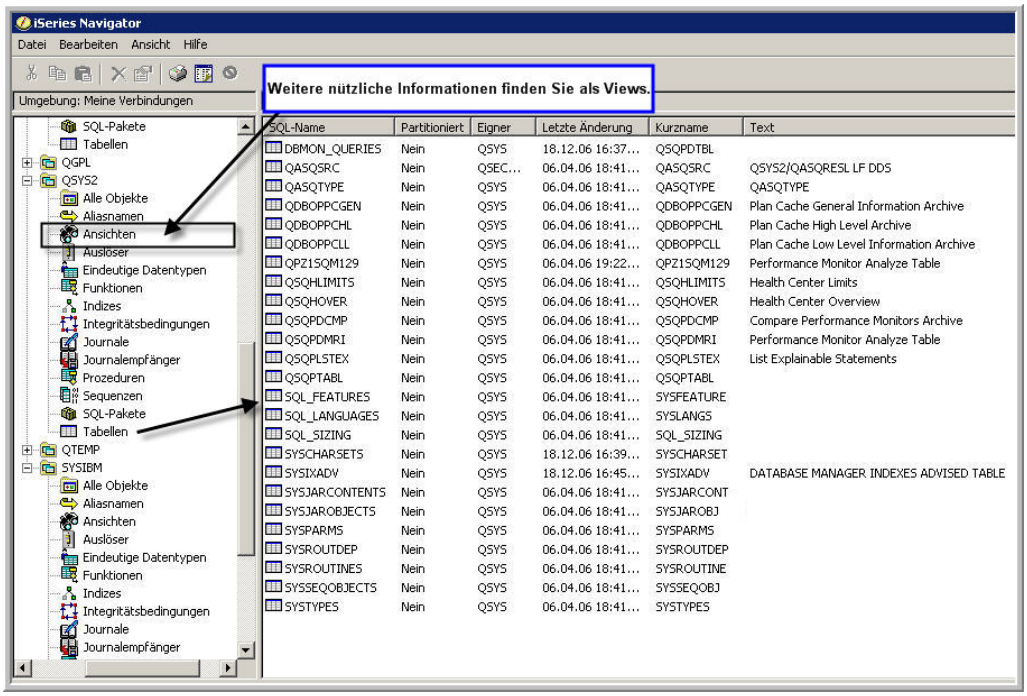
Damit kommen wir zu einem weiteren Punkt: die Namenskonvention. Gemäß der Systemnamenskonvention dient ein Slash (/) als Trennzeichen zwischen zwei Bezeichnern. Die iSeries unterstützt aber im SQL-Umfeld auch den Punkt als Trennzeichen. Sie müssen sich für eine der beiden Namenskonventionen entscheiden: entweder *SYS oder *SQL. Ich ziehe den Punkt vor und verwende daher in allen Beispielen die Namenskonvention *SQL. Die Konventionen können Sie individuell in den Prozeduren oder in den entsprechenden Front-End-Tools festlegen. Sie sollten jedoch auf keinen Fall mal mit der *SYS- und mal mit der *SQL-Konvention arbeiten. Anderenfalls müssten Sie die Syntax Ihres SQL-Codes ständig anpassen. Entscheiden Sie sich für eine Methodik und bleiben Sie anschließend dabei!

7.3.2
Seite 2

7.3.2.1 Systemschemata

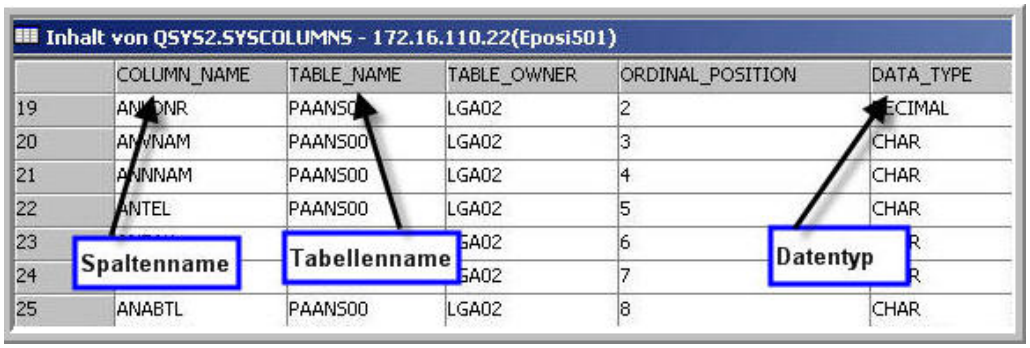
Jede relationale Datenbank benötigt eine bestimmte Anzahl an Systemschemata. Das Schema SESSION ist ein solches; es wird stets für die Deklaration temporärer SQL-Tabellen verwendet. Zusätzlich verwendet Ihr System eine Reihe weiterer Systemschemata, die alle mit dem Buchstaben „Q“ oder der Zeichenfolge „SYS“ beginnen. Sie selbst sollten also besser eine andere Namenskonvention verwenden.

Systemschemata enthalten Tabellen, die nützliche Informationen liefern. Schauen Sie sich bei Gelegenheit einmal das Schema QSYS2 an. Es enthält viele Systemtabellen und Katalogsichten, die wertvolle Informationen liefern.



Tabellen und Katalogsichten der Systembibliothek QSYS2

Ich weiß nicht mehr, wann IBM diese Systemkataloge eingeführt hat, was aber auch keine Rolle spielt. Viel wichtiger ist es, zu wissen, dass diese Informationen sich nicht nur auf SQL-Tabellen beziehen. Alle physischen und logischen Dateien, die mittels DDS erzeugt wurden, sind hier verzeichnet. So gibt es beispielsweise die Tabelle SYSCOLUMNS. Sie enthält alle Datenbankfelder. Und auch hier spielt es überhaupt keine Rolle, wie die Tabelle erzeugt wurde. Mussten Sie auch schon einmal herausfinden, in welchen Tabellen (=Dateien) eine bestimmte Spalte (Feld) verwendet wird?



	COLUMN_NAME	TABLE_NAME	TABLE_OWNER	ORDINAL_POSITION	DATA_TYPE
19	ANKDNR	PAANS00	LGA02	2	DECIMAL
20	ANVNAM	PAANS00	LGA02	3	CHAR
21	ANNNAM	PAANS00	LGA02	4	CHAR
22	ANTEL	PAANS00	LGA02	5	CHAR
23			GA02	6	R
24			GA02	7	R
25	ANABTL	PAANS00	LGA02	8	CHAR

Inhalt der Tabelle QSYS2.SYSCOLUMNS

Die Grafik zeigt nur einen kleinen Ausschnitt der verfügbaren Informationen. Schauen Sie sich einfach mal bei Gelegenheit Ihre Systemtabellen an. Sie werden staunen, was man dort an Informationen findet. Ist es nicht einfacher, die Systemtabelle abzufragen anstatt alle DDS-Quellen zu durchsuchen?

```
SELECT *
FROM QSYS2.SYSCOLUMNS
WHERE COLUMN_NAME like 'KDKDNR'
```

In der Bibliothek QSYS2 existiert auch die Tabelle SQL_SIZING. Sie enthält die je Release gültigen Datenbank-Limits:

SIZING_NAME	SUPPORTED_VALUE	COMMENTS
MAXIMUM CATALOG NAME LENGTH	18	Maximale Länge des Namens einer relationalen Datenbank
MAXIMUM COLUMN NAME LENGTH	128	Maximale Länge eines Spaltennamens
MAXIMUM COLUMNS IN GROUP BY	120	Maximale Anzahl Ausdrücke in einer Klausel GROUP BY
MAXIMUM COLUMNS IN ORDER BY	32766	Maximale Anzahl Ausdrücke in einer Klausel ORDER BY
MAXIMUM COLUMNS IN SELECT	8000	Maximale Anzahl Ausdrücke in einer Auswahlliste
MAXIMUM COLUMNS IN TABLE	8000	Maximale Anzahl Spalten in einer Tabelle oder Sicht
MAXIMUM CONCURRENT ACTIVITIES	0	Maximale Anzahl gegenwärtig aktiver SQL-Anweisungen
MAXIMUM CURSOR NAME LENGTH	18	Maximale Länge eines Cursornamens
MAXIMUM DRIVER CONNECTIONS	0	Maximale Anzahl gegenwärtig aufgebauter SQL-Verbindungen
MAXIMUM IDENTIFIER LENGTH	10	Maximale Länge einer Kennung
MAXIMUM SCHEMA NAME LENGTH	10	Maximale Länge eines Schemanamens
MAXIMUM STATEMENT OCTETS	2097151	Maximale Länge einer SQL-Anweisung
MAXIMUM STATEMENT OCTETS DATA	2097151	Maximale Länge einer SQL-Datenbearbeitungssprachenanweisung
MAXIMUM STATEMENT OCTETS SC...	2097151	Maximale Länge einer SQL-Datendefinitionssprachenanweisung
MAXIMUM TABLE NAME LENGTH	128	Maximale Länge eines Tabellen- oder Sichtnamens
MAXIMUM TABLES IN SELECT	1000	Maximale Anzahl Tabellen oder Sichten in einer Klausel FROM

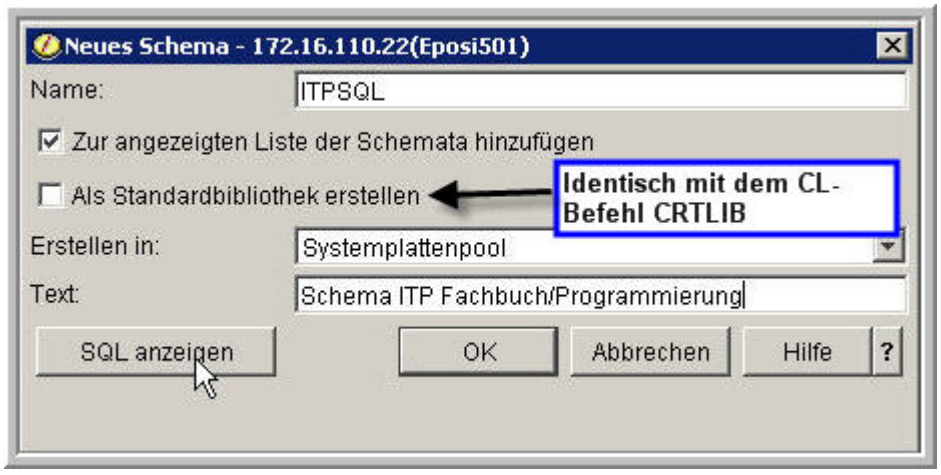
Inhalt der Tabelle QSYS2.SQL_SIZING

Sie sehen, die Crossreferenz-Tabellen der IBM enthalten nützliche Informationen, die ich im weiteren Verlauf sicherlich noch das eine oder andere Mal verwende.

7.3.2.2 Benutzerschemata erstellen

Kommen wir zurück zu unserer Beispieldatenbank, die wir ja schließlich realisieren wollen. Als Erstes benötigen wir ein eigenes Schema. Ich möchte meine Beispieldatenbank im Schema ITPSQL erstellen. Ich setze voraus, dass Sie sich ein klein wenig mit dem iSeries Navigator auskennen. Es ist ein komfortables Tool, das die Arbeit mit Datenbanken erleichtert. Ich arbeite derzeit mit dem Navigator V5R4. Falls Ihr Navigator andere Oberflächen zeigt, könnte es an unterschiedlichen Release-Ständen liegen. Das Navigator Release muss im Übrigen nicht automatisch Ihrem i5/OS Release entsprechen – Sie müssen die Oberfläche manuell aktualisieren. Auch muss der Release-Stand des Navigators nicht zwingend identisch mit dem i5/OS Release sein, um damit arbeiten zu können. Es ist durchaus möglich, mit einem Navigator V5R4 auf einen Server mit Release V5R3 zu arbeiten.

Um ein neues Datenbankschema im iSeries Navigator zu erzeugen, stellen Sie zunächst die Verbindung zum System her und erweitern den Eintrag „Datenbanken“. Sie sollten danach den Begriff der Schemata sehen. Markieren Sie diesen Eintrag. Wenn Sie jetzt im Kontext den Eintrag „NEU“ auswählen, erscheint folgender Dialog:



Ein neues Schema erstellen

Ich finde, der Dialog ist selbsterklärend. Sie geben den Schema-Namen an und bestimmen den zu verwendenden ASP; außerdem können Sie einen erklärenden Text hinzufügen. Das Gute an allen iSeries Navigator-Komponenten im Datenbankumfeld ist, dass Sie sich immer das zugehörige SQL-Statement anzeigen lassen können. So werden Sie feststellen, dass Sie anstelle der grafischen Oberfläche auch den folgenden SQL-Befehl verwenden könnten:

```
CREATE  SCHEMA  ITPSQL IN ASP1
```

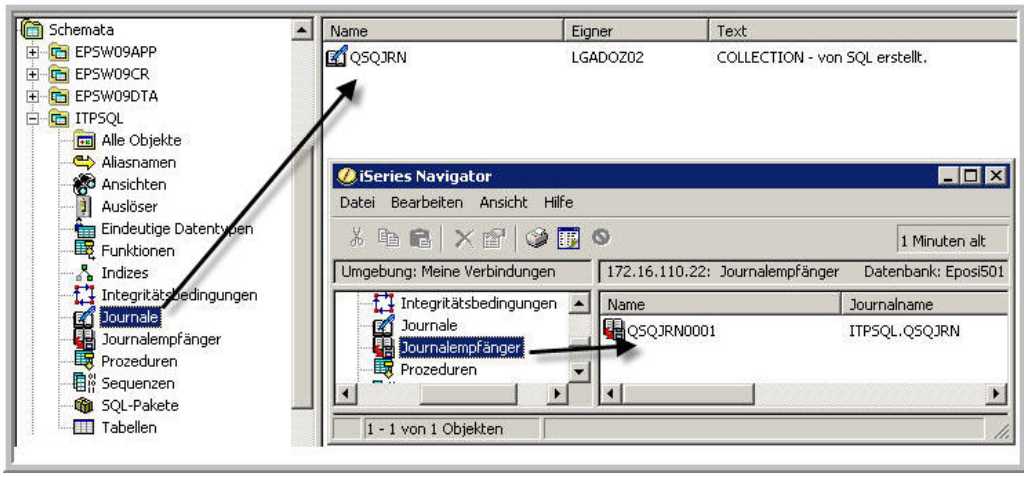
Mein Schema ITPSQL wird im ASP 1 erstellt und ist somit Teil der Systemdatenbank. Dies ist die Default-Einstellung. Alternativ könnten Sie auch im SQL-Statement anstelle des Wortes SCHEMA den Begriff COLLECTION oder DATABASE verwenden. COLLECTION ist ebenso ein Synonym für den Begriff Schema wie das Wort DATABASE. Allerdings entsprechen die Schlüsselworte COLLECTION und DATABASE nicht dem ANSI-Standard, sondern stammen aus früheren Zeiten.

Schauen wir uns jetzt das Ergebnis an: Wir haben ein *LIB-Objekt erstellt. Nichts weist darauf hin, dass die Bibliothek mit einem SQL-Statement erzeugt wurde.

```
                          Mit Objekten arbeiten

Auswahl eingeben und Eingabetaste drücken.
  2=Berechtigung editieren   3=Kopieren   4=Löschen   5=Berechtigung anzeigen
  7=Umbenennen   8=Beschreibung anzeigen   13=Beschreibung ändern

Opt  Objekt      Art       Bibliothek  Attribut    Text
__   ITPSQL      *LIB      QSYS        PROD        Schema für Fachbuch/Program
```

**LIB-Objekt ITPSQL*

Allerdings gibt es Unterschiede, wenn Sie sich den Inhalt des Schemas anschauen. Die Bibliothek ist nicht leer, wie Sie es gewohnt sind, sondern sie enthält Objekte.



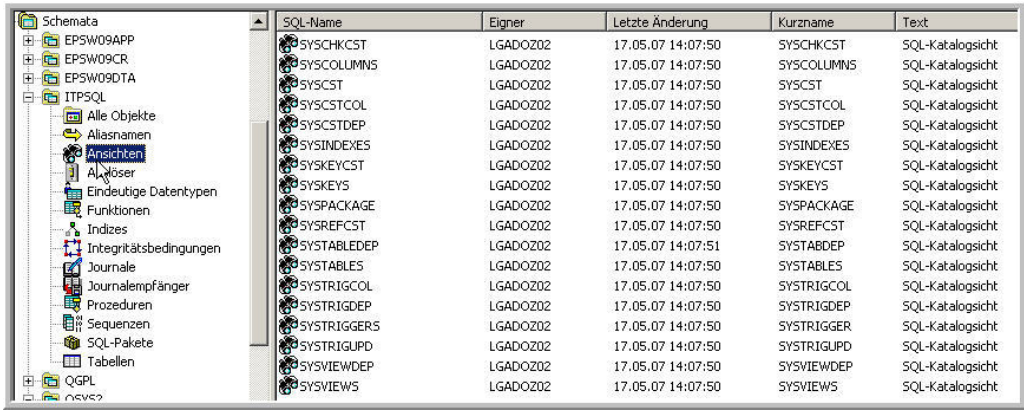
Jedes SQL-Schema enthält ein Journal

Im Schema ITPSQL existiert das Journal QSQJRN sowie der dazugehörigen Empfänger QSQJRN001. Alle Tabellen, die wir später im Schema erstellen, werden automatisch aufgezeichnet. Dabei ist es egal, ob Sie die Tabellen mit DDS oder SQL erstellen. Die Empfänger werden bei jeder Datensicherung automatisch gewechselt und gesichert. Sie müssen dann nur in regelmäßigen Abständen alte Empfänger bereinigen. Falls Sie keine Aufzeichnungen Ihrer Tabellen wünschen, müssen Sie die Journalisierung manuell stoppen oder eine Standardbibliothek erstellen, die kein Journal enthält. Doch bedenken Sie, dass viele

7.3.2
Seite 6

SQL Features die Journalisierung voraussetzen. Beachten Sie, dass Sie zukünftig mit einem SQL-Statement eine Vielzahl an Zeilen manipulieren. Häufig lassen sich fehlerhafte Manipulationen nur korrigieren, wenn die Tabellen aufgezeichnet wurden.

Schauen Sie sich ein weiteres Mal den Inhalt des Schemas ITPSQL an. Wählen Sie hierfür den Eintrag „Ansichten“.



Katalogsichten des SQL-Schemas

Ansichten sind logische Sichten, die im i5/OS als logische Dateien existieren. Man bezeichnet diese Objekte als Kataloge. Sie beziehen sich nur auf das jeweilige Schema, zeigen Ihnen somit einen Ausschnitt der bereits besprochenen Systemkataloge bzw. einen Ausschnitt der Systemdatenbank.

Zusätzlich könnten wir in unserem Schema ein interaktives Data Dictionary erstellen, wenn wir folgenden Zusatz verwenden:

```
CREATE  SCHEMA  ITPSQL  WITH  DATA  DICTIONARY
```

Aufgrund der Klausel werden IDDU-Data-Dictionary-Tabellen in das Schema eingefügt. Seitens DBMS wird sichergestellt, dass diese bei Änderungen und/oder Erweiterungen der Datenbankobjekte automatisch aktualisiert werden. Falls Sie mit einem Schema arbeiten, dass IDDU-Data-Dictionary-Tabellen enthält, dürfen Ihre Tabellen keine LOB- und/oder Datalink-Spalten enthalten. Daher wird der Zusatz seitens IBM zwar weiter unterstützt, jedoch nicht empfohlen.

Fassen wir abschließend zusammen:

SQL		i5/OS
	Synonym	
SCHEMA	COLLECTION oder DATABASE	Bibliothek
CREATE SCHEMA (Es wird ein Objekt *LIB mit Journal und Katalogsichten erstellt.)	CREATE COLLECTION (Es wird ein Objekt *LIB mit Journal und Katalogsichten erstellt.)	CRTLIB (Es wird ein leeres Objekt *LIB erstellt.)

7.3.2
Seite 8

7.3.3 SQL-Tabellen

Die Tabelle ist die Grundlage des relationalen Datenbanksystems. Auf einer iSeries werden Tabellen normalerweise als physische Dateien bezeichnet, die auf unterschiedliche Arten erstellt werden können. Sie sehen vermutlich eher physische Dateien, die mit DDS erzeugt wurden. DDS ist eine mächtige Sprache, die sowohl für die Beschreibung physischer als auch logischer Dateien verwendet wird. Nur entwickelt IBM diese Sprache nicht weiter!

SQL verfügt über das Statement CREATE TABLE. Diese Anweisung erlaubt es uns, Tabellen innerhalb der System- oder einer Benutzerdatenbank zu erzeugen. Das SQL-Statement wird zudem seitens IBM kontinuierlich erweitert und verbessert.

Die Tabelle selbst ist eine Sammlung von Spalten, die Sie normalerweise als Felder bezeichnen. Inhalt der Tabelle sind viele Zeilen, die Sie als Datensätze kennen. Die Tabelle kann keine oder mehrere Zeilen enthalten; die einzelne Zeile kann aus einer oder mehrerer Spalten bestehen. Die Begriffe „Datensatz“ und „Zeile“ beziehungsweise „Spalte“ und „Feld“ sind beliebig austauschbar. Ich nutze allerdings im Folgenden stets die SQL-Begriffe „Zeile“ und „Spalte“.

Wie verhält es sich nun aber mit den Begriffen „Physische Datei“ und „Tabelle“? Sind diese Begriffe ebenfalls beliebig austauschbar? Aus Sicht des Betriebssystems i5/OS können wir die Frage mit „Ja“ beantworten. Sie werden keine Unterschiede in der Handhabung der Objekte feststellen, denn alle SQL-Tabellen sind *FILE-Objekte. Auch Ihre HLL-Programme arbeiten problemlos mit SQL-Tabellen.

Ob ein Dateiobjekt mittels DDS oder SQL erzeugt wurde, lässt sich lediglich anhand der Dateibeschreibung feststellen:

```
DSPFD-Befehlseingabe
  Datei . . . . . : FILE       ORT
    Bibliothek  . . . . . :    ITPSQL
  Art der Information . . . . . : TYPE       *ALL
  Dateiattribute  . . . . . : FILEATR    *ALL
  System  . . . . . : SYSTEM     *LCL
Kopfzeile der Dateibeschreibung
  Datei . . . . . : FILE       ORT
  Bibliothek  . . . . . :      ITPSQL
  Dateiart  . . . . . :        Physisch
  Dateiart  . . . . . : FILETYPE   *DATA
  Zusatzspeicherpool-ID . . . . . :  00001
Datenbankdateiattribute
  Extern beschriebene Datei . . . . . :  Ja
  SQL-Dateiart  . . . . . :  ------->    TABLE
```

Dateibeschreibung der Tabelle ORT

7.3.3
Seite 2

Alle SQL-Tabellen erhalten das Attribut SQL, Art TABLE. Auch der Navigator unterscheidet in diesem Punkt:

Unterscheidung zwischen physischer Datei und SQL-Tabelle

Falls Sie einmal ermitteln müssen, welche Ihrer Tabellen mit SQL und welche mit DDS erzeugt wurden, können Sie hierfür die Systemtabelle SYSTABLES im Schema QSYS2 nutzen. Sie enthält alle Tabellen Ihres Systems. Dabei unterscheidet der Eintrag in der Spalte TABLE_TYPE zwischen logischen DDS-Dateien (L), physischen DDS-Dateien (P) und SQL-Tabellen (T). Das folgende SQL-Statement zeigt beispielsweise alle Tabellen, die mit dem Befehl CREATE TABLE erzeugt wurden.

```
SELECT TABLE_NAME, TABLE_TYPE
FROM QSYS2.SYSTABLES
WHERE TABLE_TYPE = 'T'
ORDER BY TABLE_NAME;
```

Fassen wir zusammen:

SQL		i5/OS
Deutsch	**Englisch**	
Zeile	Row	Datensatz
Spalte	Column	Feld
Tabelle	Table	Physische Datei

7.3.3.1 Der Befehl CREATE TABLE

CREATE TABLE ist eine SQL92-Anweisung. Sie wurde von allen relationalen Datenbankherstellern implementiert und kann als Kern der Datendefinitionssprache gesehen werden, denn mit ihr kann man fast das gesamte ER-Modell realisieren.

Das folgende SQL-Skript erstellt zunächst unsere „Adresstabelle“:

```
CREATE TABLE ITPSQL.ADRESSE
(
ADKURZ                     INTEGER NOT NULL
                          GENERATED ALWAYS AS IDENTITY
                                    (
                                    START WITH 10 INCREMENT BY 10
                                    MINVALUE 10 MAXVALUE 2147483647
                                    NO CYCLE ORDER
                                    CACHE 20
                                    ),

AKEY                                INTEGER           NOT NULL,
ADRESSART     FOR COLUMN ADRART     INTEGER           NOT NULL ,
OKURZ                               INTEGER           NOT NULL ,
STRASSE                             VARCHAR(150) ALLOCATE(50) ,

HAUSNR                              CHAR(10) ,
ADRESSINSERTDATE     FOR COLUMN ADRINSDAT     TIMESTAMP ,
ADRESSINSERTUSER     FOR COLUMN ADRINSUSR     CHAR(10) ,
ADRESSUPDATEDATE     FOR COLUMN ADRUPDDAT     TIMESTAMP ,
ADRESSUPDATEUSER     FOR COLUMN ADRUPDUSR     CHAR(10)

)
RCDFMT ADRFMT;
```

Man sieht, der Befehl beginnt mit den Schlüsselwörtern CREATE TABLE – gefolgt vom Tabellennamen und einer geöffneten runden Klammer. Danach folgt eine durch Komma getrennte Reihe von Spaltendefinitionen. Nach der schließenden Klammer können weitere Klauseln folgen. Erwähnenswert ist hier die seit V5R4 vorhandene Klausel RCDFMT. Jetzt ist es möglich, den SQL-

7.3.3
Seite 4

Tabellen abweichende Satzformatnamen zuzuweisen. Darüber dürften sich alle RPG-Programmierer freuen, die bislang stets im Programm den Satzformatnamen umbenennen mussten, wenn SQL-Tabellen verwendet wurden. Wieder ein Grund weniger, auf die Verwendung der SQL-Tabellen zu verzichten!

Jede Spalte hat einen SQL-Namen und – falls erforderlich – einen verkürzten i5/OS Spaltennamen. Außerdem werden Datentyp und optional eine Default-Einstellung für die Spalte bestimmt. Am Ende des Befehls stehen die abschließende runde Klammer und ein Semikolon. Optional könnten noch Spaltenbedingungen hinzugefügt werden.

Betrachten wir obiges Skript noch einmal: Der Name der Tabelle ist Adresse, was leicht zu erkennen ist. Der Name der ersten Spalte ist AdKurz mit dem Datentyp INTEGER as IDENTITY. Die SQL-Datentypen werden wir uns im weiteren Verlauf noch genauer anschauen. Man sieht nach der Datentyp-Definition in derselben Reihe die Spalteneinschränkung NOT NULL. NOT NULL steht dafür, dass die Spalte AdKurz niemals den Wert NULL annehmen darf. Bei vielen Spalten der Tabelle Adresse fehlt die Einschränkung NOT NULL. Das heißt, dass sie jeweils auch den Wert NULL annehmen dürfen. Die Spalte Strasse ist vom Typ VARCHAR(150). Dies ist die Maximallänge der Spalte. Außerdem werden für die Spalte Strasse 50 Zeichen durch das Schlüsselwort ALLOCATE im festen Zeilenbereich reserviert, um zu verhindern, dass die Spalteninhalte im Overflow-Bereich gespeichert werden. Auch auf die Definition variabler Spalten gehe ich zum späteren Zeitpunkt ein. Schließlich finden wir auch noch die Klausel FOR COLUMN. Damit lege ich einen verkürzten i5/OS-Spaltennamen fest.

In unserem Konzept haben wir zudem beschreibende Texte für die Tabellen und deren Spalten festgelegt. Diese Labels werden in einem separaten Statement zugewiesen. Um die Tabellenbeschreibung zu hinterlegen, können Sie das folgende Statement verwenden:

```
LABEL ON TABLE ITPSQL.ADRESSE
     IS ‚Adressen der Kunden(FIRMA) und Ansprechpartner‘ ;
```

Für die Beschreibung der einzelnen Spalten, nutzen Sie die Anweisung:

LABEL ON COLUMN ITPSQL.ADRESSE

```
(
     ADKURZ           TEXT IS      'PrimärKey',
     AKEY               TEXT IS      'Fremdschlüsselfeld' ,
     ADRESSART          TEXT IS      'Art der Adresse LIAdr, ReAdr' ,
     OKURZ           TEXT IS      'Ortsbezeichnung' ,
     STRASSE           TEXT IS      'Strassenname' ,
     HAUSNR           TEXT IS      'Hausnummer' ,
     ADRESSINSERTDATE      TEXT IS      'Datum Insert' ,

     ADRESSINSERTUSER      TEXT IS      'User für Insert' ,

     ADRESSUPDATEDATE      TEXT IS      'Datum des Update' ,

     ADRESSUPDATEUSER      TEXT IS      'Benutzer des Update'
) ;
```

7.3.3.2 Tabellen- und Spaltennamen

Sie haben gesehen, dass Sie den Tabellennamen oder besser gesagt die Tabellennamen festlegen müssen, wenn die Tabelle mit SQL erstellt wird. Außerdem bestimmen Sie, welche Spalten die Tabelle enthalten soll, indem Sie den Spaltennamen bestimmen, den Datentyp deklarieren und mögliche Tabellen- und Schlüsselbedingungen definieren.

Ein erster Unterschied zwischen DDS und SQL ist die Länge der Tabellen- und Spaltennamen. SQL erlaubt eine Länge von 128 Zeichen für Tabellennamen und Spalten. i5/OS hingegen verwendet höchstens 10 Zeichen. Was geschieht also, wenn Sie lange SQL-Namen verwenden?

Die Anweisung

```
          CREATE TABLE  ITPSQL.KOMMUNIKATION
                              :
                        ( KOMMUNIKATIONINSERTDATE  ……,
                          KOMMUNIKATIONINSERTUSER          .)
```

erstellt die Tabelle KOMMUNIKATION im Schema ITPSQL. Diese Tabelle enthält eine Spalte KOMMUNIKATIONINSERTDATE. Da sowohl der Tabellenname als auch der Spaltenname zu lang ist, übernimmt die iSeries die ersten fünf Zeichen jedes Namens und hängt eine fortlaufende Nummer an das Ende des Namens. Sie erstellt also die Tabelle KOMMU00001. Anschließend wird der Systemkatalog durchsucht, um festzustellen, ob eine derartige Tabelle im Schema ITPSQL bereits existiert. Wenn ja, inkrementiert das System die Nummer, bis es eine nicht belegte Nummer findet. Dasselbe geschieht mit den Spal-

tennamen KOMMUNIKATIONINSERTDATE und KOMMUNIKATIONINSERTUSER. Das System würde die Spaltennamen KOMMU00001 und KOMMU00002 generieren. Später können Sie in den SQL-Zugriffen sowohl die generierten Systemnamen als auch die SQL-Namen verwenden. In Nicht-SQL-Programmierumgebungen müssen Sie jedoch stets den Systemnamen für die Bezeichnung der Tabellen und Spalten verwenden. Eine wenig verlockende Aussicht.

Eine Methode, um die automatische Erstellung von Systemnamen zu verhindern besteht darin, nur Namen mit 10 oder weniger Zeichen zu verwenden. Diese Methode versuche ich – wenn möglich – für Tabellennamen anzuwenden. Daher habe ich den Tabellennamen KOMMUNIKATION im physischen Entwurf umbenannt und verwende stattdessen den Namen KONTAKT. Ich finde es allerdings schade, wenn ich aufgrund dieser Konvention sprechende Spaltennamen derart verkürze, dass ich sie zusätzlich dokumentieren muss. Daher nutze ich im CREATE TABLE-Statement für Spalten, die die Längenkonventionen verletzen, die Möglichkeit, eigene Systemnamen anzugeben und verhindere dadurch die Generierung der hässlichen Spaltennamen mit fortlaufender Nummer.

```
CREATE TABLE  ITPSQL.KONTAKT
(      :
     KONTAKTINSERTDATE  FOR COLUMN KONINSDAT
        KONTAKTINSERTUSER  FOR COLUMN KONINSUSR …….)
```

Die Klausel FOR COLUMN legt für die Spalten KontaktInsertDate und KontaktInsertUser die Systemnamen KonInsDat und KonInsUSR fest.

Auch wenn Sie unerlaubte Zeichen oder Leerzeichen in den Tabellen- oder Spaltennamen verwenden, werden iSeries-Systemnamen generiert. Nur, warum sollten Sie solche Zeichen verwenden und sich unnötig das Leben schwer machen? Verwenden Sie für Tabellen- und Spaltennamen ausschließlich die Buchstaben A bis Z, die Ziffern 0 bis 9 und die Zeichen @, #, $ und _. Alle weiteren Zeichen sollten Sie vermeiden!

7.3.3.3 SQL-Datentypen

Bevor wir uns die Datentypen genauer ansehen, sollten wir kurz über den Wert NULL sprechen. NULL ist ein spezifischer SQL-Wert. Programmierer, die mit der DB2/UDB for iSeries arbeiten, haben sich selten mit NULL-Daten beschäftigt; sie wissen wenig darüber. NULL-Werte werden immer häufiger in iSeries-Datenbanken verwendet, deshalb sollten sich alle Programmierer mit ihnen vertraut machen. Außerdem gibt es durchaus Situationen, in denen NULL-Werte nützlich sind. Stellen Sie sich folgendes Szenario vor: Sie berechnen in regelmäßigen Abständen Positionswerte. Solange die Positionswerte nicht berechnet sind, ist die entsprechende Spalte mit dem Wert 0 belegt. Jetzt gibt es aber auch Positionen, die ohne Berechnung laufen und ebenfalls den Wert 0 haben. Eine Unterscheidung zwischen kostenlosen Positionen und noch nicht berechneten Positionswerten ist anhand des Inhalts nicht möglich. Oder anders ausgedrückt: Wenn Sie mit NULL arbeiten, dann hätten nicht berechnete Werte den Wert NULL; sie könnten problemlos identifiziert werden. Wir sollten uns also etwas genauer mit dem Wert NULL beschäftigen.

Bei Null-Werten handelt es sich nicht um Leerstellen, sondern um unbekannte Werte. Sie werden auf anderen DB-Plattformen umfassend eingesetzt und haben einige wertvolle Funktionen. Bevor man Null-Daten in eine Spalte laden kann, muss die Spalte als NULL zulässig definiert werden.

Nehmen wir an, Sie erfassen einen neuen Kunden. Die Zeile darf nur mit einer eindeutigen Kundennummer gespeichert werden. Der Primärschlüssel – in diesem Fall die Kundennummer – benötigt einen Wert. Damit SQL versteht, dass die Spalte nicht leer bleiben darf, benötigen Sie die Klausel **NOT NULL**.

Jetzt gibt es aber auch Spalten, die durchaus keine Benutzereingaben erzwingen. Ein neuer Kunde wird zunächst keine Umsätze vorzuweisen haben. Das heißt: Die Spalte UMSATZ könnte den Wert 0 haben. Wenn Sie es gewohnt sind, dass leere Felder automatisch mit ihren Default-Werten initialisiert werden, werden Sie feststellen, dass SQL sich in diesem Fall anders verhält. Die Spalte UMSATZ würde den Wert NULL erhalten, was soviel heißt wie „unbekannter Wert“. Sie müssen SQL mitteilen, dass Sie die Default-Werte für die Initialisierung erwarten. Dafür verwenden Sie die Klausel **NOT NULL WITH DEFAULT**. Dies entspricht dem Verhalten, das Sie kennen.

Betrachten wir nun die SQL-Datentypen. Grundsätzlich lassen sich drei Typen unterscheiden: **temporale Daten, Strings** und **Ziffern**. Temporale Daten sind Daten mit Zeitbezug – wie zum Beispiel Datumsangaben, Zeitstempel und Zeitangaben. String-Daten basieren auf Zeichen, enthalten aber auch Mechanismen für die Speicherung binärer Daten. Ziffern sind nichts anderes als Felder für die Speicherung von Zahlen in unterschiedlicher Präzision und Byte-Länge. Wenn Sie sich die SQL-Datentypen detailliert anschauen, werden Sie sehr

schnell feststellen, dass SQL alle Datentypen kennt, die Sie auch im DDS verwenden können. Darüber hinaus gibt es bereits seit Version V4 das Konzept der benutzerdefinierten Datentypen sowie des neuen Datentyps DATALINK. DATALINKs sind letztendlich nur Hyperlinks zu Dateien, die außerhalb der DB2/UDB gespeichert sind. Später kamen dann Identity-Spalten hinzu. Dazu zählt der Datentyp ROWID, die Klausel GENERATED ALWAYS AS IDENTITY und auch das Objekt SEQUENZ.

7.3.3.3.1 Zeichen speichern

Zeichenspalten können Buchstaben und/oder Ziffern speichern. Die nachfolgende Tabelle zeigt verschiedene Character-Datentypen und deren Attribute:

Character Strings	SBCS-Datentypen
CHAR(x) CHARACTER(x)	Zeichenstring. x bezeichnet die feste Länge und kann maximal den Wert 32.766 annehmen.
VARCHAR(x) CHAR VARYING(x) CHARACTER VARYING(x)	Variabler Zeichenstring mit unterschiedlicher Länge. x bezeichnet hierbei die maximale Länge und muss einen Wert zwischen 1 und 32.740 enthalten.
CLOB(x) CHARACTER LARGE OBJECT(x) CHAR LARGE OBJECT(x)	Zeichenstring mit der Maximallänge x. Maximal können zirka 15 MB gespeichert werden.

Der Eintrag CHAR(x) entspricht dem DDS-Datentyp A. Dieser Datentyp ist der wichtigste String-Datentyp. Jedes Zeichen beansprucht genau ein Byte. Optional können Sie angeben, ob die Spalte Single-Byte-Character String- (SBCS-) oder Bit-Daten enthalten soll. Hierfür verwenden Sie die Klausel **FOR SBCS DATA** oder **FOR BIT DATA**. Außerdem ist es möglich, eine CCSID anzugeben. Falls die verwendete CCSID Double-Byte-Character Data-Daten (DBCS) darstellt, beansprucht jedes gespeicherte Zeichen zwei Byte. Die gleiche Wirkung erzielen Sie, wenn Sie GRAPHIC deklarieren.

```
CREATE TABLE  ITPSQL.TEST
               (
                    Col1  CHAR(10) FOR SBCS DATA,
                    Col2  CHAR(10) FOR BIT DATA,
                    Col3  CHAR(8) CCSID 367
               )
```

Sie sehen, dass ich im Beispiel die Klausel **FOR BIT DATA** verwende. Dadurch wird CCSID dieser Spalte auf 65535 gesetzt; sie kann in keine andere CCSID übersetzt werden, wenn die Rückkehr zum Client erfolgt. Die Klausel FOR BIT DATA kann hilfreich sein, wenn Sie binäre Informationen auf dem Server speichern wollen und wenn Sie sicherstellen müssen, dass keine Übersetzung zwischen den unterschiedlichen Zeichensätzen stattfindet.

Der Datentyp VARCHAR(x) ähnelt dem Datentyp CHAR. Mit VARCHAR-Spalten ist man in der Lage, Zeichenspalten zu deklarieren, die eine große Maximallänge haben, aber jeder Zeile nur den Platz zuteilen, der für die Speicherung der Daten erforderlich ist. In DDS wird dieser Sachverhalt durch das Schlüsselwort VARLEN repräsentiert. VARCHAR-Spalten können maximal 32.739 Byte lang sein.

Der Datentyp CLOB(x) steht für Character Large Object. Diese Felder können bis zu 15.768.640 Byte speichern. Der Wert X ist ein Integer Wert, der die maximale Anzahl der Bytes repräsentiert, die im Feld gespeichert werden sollen. X kann in Kilobyte oder Megabyte angegeben werden, indem ein K oder M nach dem Integer-Wert verwendet wird. CLOB (10 K) spezifiziert beispielsweise ein Zeichenfeld mit maximal 10 Kilobyte Daten.

Graphic Strings	DBCS-Datentypen
GRAPHIC(X)	Ein DBCS-String mit fester Länge. Das Längenattribut muss zwischen 1 und 16.383 liegen.
VARGRAPHIC(x) GRAPHIC VARYING(x)	Ein variabler DBCS String mit der maximalen Länge x. x repräsentiert einen Wert zwischen 1 und 13.370.
DBCLOB	DBCS-Variante des CLOB-Typen mit einer maximalen Länge von 7 MB.

Der Datentyp GRAPHIC verhält sich wie ein Zeichenfeld. Allerdings werden die Daten grundsätzlich als DBCS- oder Universal-Coded-Character-Set-Daten (UCS2) gespeichert. Wenn Sie den Datentyp GRAPHIC nutzen, sollten Sie CCSID des DBCS-Zeichensatzes angeben, den sie verwenden wollen. Verzichten Sie auf die Angabe von CCSID wird 65535 angenommen; es findet keine automatische Umsetzung zwischen Host- und Client-Zeichensatz statt.

Binary Strings	
BLOB(x) BINARY LARGE OBJECT(x)	BLOB-Felder enthalten Binärdaten. Sie haben CCSID 65535 und werden zwischen den einzelnen Zeichensätzen nicht übersetzt; maximal 15 MB.
VARBINARY(x) BINARY VARYING(x)	Variable Binärfelder, deren Maximallänge 32.740 Byte beträgt.

Ergänzung 2/2007

7.3.3
Seite 10

Binärspalten verhalten sich wie CLOB-Spalten, speichern aber grundsätzlich binäre Informationen. Dies können Bilder, Audiodateien oder Dokumentenabbilder sein. Die maximale Größe einer BLOB-Spalte beträgt 15.768.640 Byte.

Variable Spalten und große Objekte sind nützlich. Doch gibt es einige Restriktionen, die Sie beim Einsatz beachten sollten:

	CLOB, DBCLOB oder BLOB
GROUP By Klausel	Nicht erlaubt.
ORDER By Klausel	Nicht erlaubt.
CREATE INDEX Statement	Nicht erlaubt.
SELECT DISTINCT	Nicht erlaubt.
SubSelect mit UNION, EXCEPT oder INTERSECT ohne ALL-Schlüsselwort	Nicht erlaubt.
Definition eines Primär-, UNIQUE oder Fremdschlüssels	Nicht erlaubt.

Noch etwas sollten Sie beim Einsatz variabler Spalten berücksichtigen – dies gilt gleichermaßen für den Einsatz der LOB-Spalten. Um die Arbeitsweise von Feldern mit variabler Länge oder LOBs zu verstehen, muss man zuerst begreifen, wie Datensätze gehandhabt werden.

Üblicherweise implementiert die DB2/UDB auf der iSeries Datensatzformate mit fester Länge – das heißt: Jede Zeile hat exakt die gleiche Länge, basierend auf dem maximalen Speicher, der für jede Spalte in der Tabelle benötigt wird. Wenn eine Spalte als alphanumerisches Feld mit variabler Länge definiert ist, wird der variable Teil des Datensatzes in einem separaten Bereich auf der Festplatte gespeichert. Im festen Längenteil des Datensatzes wird ein Pointer implementiert, der auf den Overflow-Bereich der Tabelle verweist. Da die variablen Daten aber im separaten Bereich stehen, erfordert das Lesen dieser Zeilen eine zusätzliche I/O-Operation, um die Daten vom sekundären Speicherplatz zu holen. Durch diese zusätzliche Operation wird die Zeit erhöht, die für das Lesen des Datensatzes erforderlich ist. Die Performance der Anwendung sinkt.

Stellen Sie sich vor, Sie deklarieren die Tabelle ORT, die die Spalte ORTSNAME enthält.

```
CREATE TABLE  ITPSQL.ORT
               (
                    …,
                    OrtsName VARCHAR(150), …
               )
```

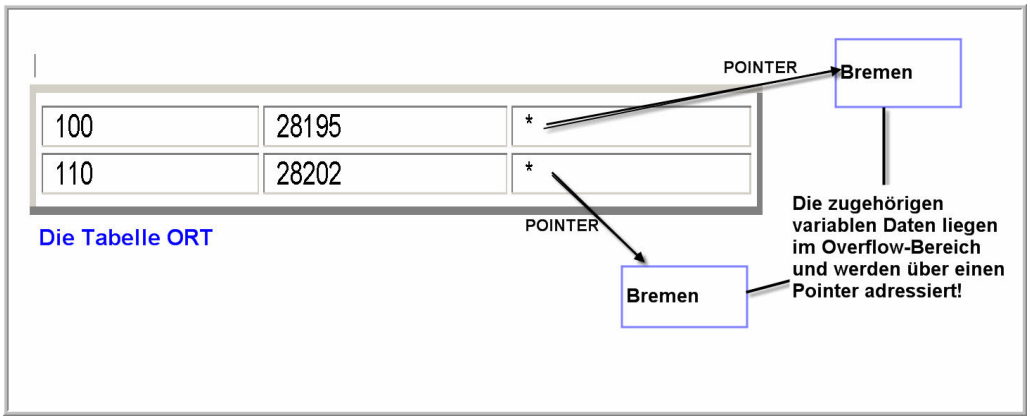


Tabelle ORT mit variabler Spalte ohne ALLOCATE

Jeder Ortsname landet im Overflow-Bereich und ist nicht zusammenhängend mit den restlichen Informationen gespeichert.

Sie wissen allerdings aufgrund Ihrer Erfahrungen, dass 90 Prozent aller Ortsnamen nur 50 Zeichen lang sind. Die übrigen 10 Prozent benötigen bis zu 150 Zeichen. In diesem Fall ist es sinnvoll, die Spalte ORTSNAME als VARCHAR-Spalte mit der maximalen Länge 150 zu definieren und mit Hilfe des Schlüsselworts ALLOCATE 50 Byte als feste Länge zuzuordnen:

```
CREATE TABLE  ITPSQL.ORT
               (
                    …,
                    OrtsName VARCHAR(150) ALLOCATE(50), …
               )
```

Bei korrekter Verwendung des Schlüsselworts ALLOCATE, werden weniger Daten in den Overflow-Bereich geschrieben. Dadurch verhindern Sie die zusätzlichen I/O-Operationen, da die meisten Daten als zusammenhängende Zeile gespeichert werden.

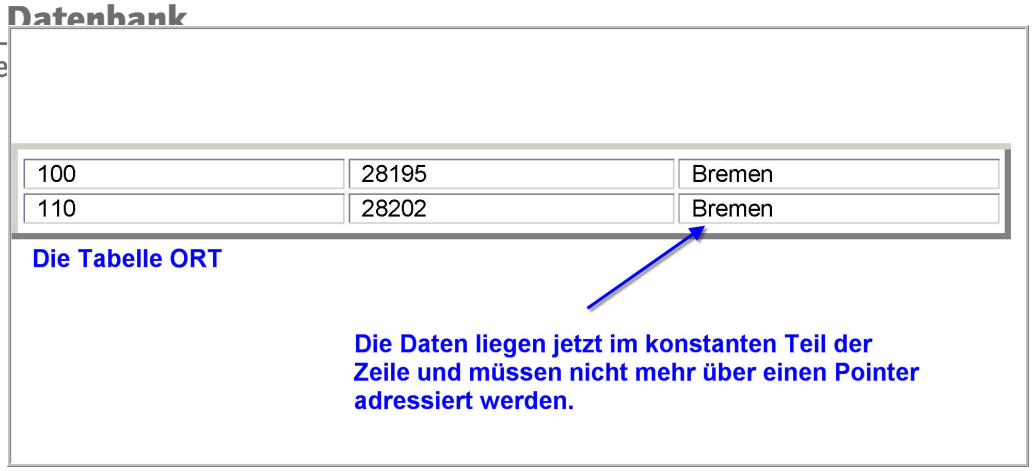
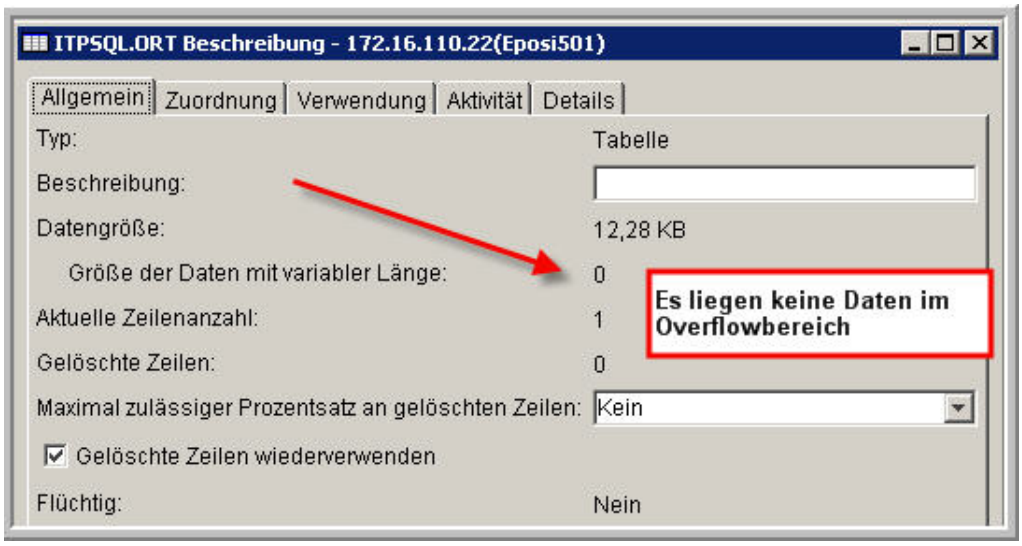


Tabelle ORT mit variabler Spalte mit ALLOCATE

Was bedeutet das für die Entwicklung Ihrer Datenbank? Erstens müssen Sie eine ausreichende Menge an Speicherplatz über das Schlüsselwort ALLOCATE reservieren. Prüfen Sie also stets vorher, wie lang die Mehrzahl der Spalteninhalte ist. Diese Größe reservieren Sie mit dem ALLOCATE-Schlüsselwort, so dass nur wenige Zeilen im Overflow-Bereich landen. Vielleicht ist es sogar möglich, vollständig auf die variable Spalte zu verzichten? Ich habe nie verstanden, warum plötzlich – nach Einführung der VARCHAR-Typen – alle Zeichenspalten variabel lang sein müssen. Falls Sie bereits Tabellen oder physische DDS-Dateien nutzen, die Datentyp VARCHAR verwenden, sollten Sie mit dem iSeries Navigator die Größe des Overflow-Bereichs prüfen. Dazu wählen Sie im Kontextmenü der entsprechenden Tabelle den Eintrag „Beschreibung“. Daraufhin erscheint folgendes Fenster:



Tabellenbeschreibung/Overflow prüfen

Wenn Sie BLOB- oder CLOB-Daten planen, ist es immer besser, eine separate Tabelle zu erstellen, die diese Spalte enthält. Stellen Sie sich vor, Sie entwickeln eine Anwendung, bei der für jeden Artikel ein entsprechendes Bild gespeichert wird. Zusätzlich gehören zum Artikel natürlich weitere Attribute: Preis, Lieferzeit, Lieferantennummer usw. Anstatt eine einzige Tabelle zu erstellen, die alle

Attribute enthält, sollten Sie zwei Tabellen nutzen. Die erste Tabelle enthält die grundsätzlichen Attribute sowie einen Fremdschlüssel auf das zugehörige Bild in der zweiten Tabelle. Auf diese Weise müssen die Bilddaten nicht abgerufen werden, wenn Sie nur die Preis- und/oder Lieferanteninformationen benötigen. So können mehr Artikelzeilen in den Cache-Bereich geladen werden; Sie verhindern unnötige I/O-Zugriffe.

7.3.3.3.2 Temporale Daten speichern

Temporale Datentypen	
DATE	Enthält gültigen Datumswert. Die Formate *JUL, *DMY, *YMD, *MDY können nur die Jahre 1940 bis 2039 enthalten. Das Datumsfeld kann 6, 8 oder 10 Bytes lang sein; intern werden lediglich 4 Byte für die Speicherung benötigt.
TIME	Enthält eine gültige Uhrzeit bestehend aus STD: MIN: SEK. Intern werden 3 Byte gespeichert; jedes Byte enthält eine zwei Byte gepackte Zahl.
TIMESTAMP	Ein Feld, das in 10 Byte einen Datums-Zeit-Wert speichert. Es enthält die Informationen: JJMMTTStdMinSecMSec

Datums- und Zeitfelder können Informationen zum Tagesdatum und zur Tageszeit speichern. Zeitstempel sind die Kombination aus beidem. Allerdings verfügt ein Zeitstempelfeld über eine Präzision bis auf Mikrosekundenebene. Intern bildet die iSeries das Zeitstempelfeld als 10-Byte-String ab. Die ersten vier Byte werden für das Datum, die nächsten drei Byte für die Zeit und die letzten drei Byte für die Mikrosekunden benötigt. Wenn Sie die Klausel WITH DEFAULT für eine temporale Spalte verwenden, wird als Default-Wert stets das aktuelle Datum und die aktuelle Uhrzeit genutzt.

7.3.3
Seite 14

7.3.3.3.3 Ziffern speichern

Die DB2/UDB erlaubt zwei Arten von Ziffern: exakte und approximierte. Gleitkommazahlen sind approximierte Zahlen; sie stehen mit einfacher oder doppelter Präzision zur Verfügung. Alle anderen numerischen Datentypen bieten eine präzise Darstellung der Nachkommastellen. Die folgende Tabelle zeigt Ihnen die numerischen Datentypen und deren Eigenschaften:

Numerische Datentypen mit Präzision	
SMALLINT	Wertebereich -32.768 bis +32.768, keine Nachkommastellen. Intern werden 2 Byte für die Darstellung der Zahl benötigt.
INTEGER INT	Wertebereich -2.147.483.648 bis +2.147.483.648, keine Nachkommastellen. Intern werden 4 Byte für die Darstellung der Zahl benötigt.
BIGINT	Wertebereich: -9.223.372.036.854.775.808 bis -9.223.372.036.854.775.808, keine Nachkommastellen. Intern werden 8 Byte für die Darstellung der Zahl benötigt.
DECIMAL(X;Y) DEC(X;Y)	Gepackte Dezimalzahl, x entspricht der Länge, y repräsentiert die Anzahl Nachkommastellen. Ohne Angabe der Nachkommastellen ist der Standardwert 0.
NUMERIC(X;Y)	Gezonte Dezimalzahl, x entspricht der Länge, y repräsentiert die Anzahl Nachkommastellen. Ohne Angabe der Nachkommastellen ist der Standardwert 0.
Numerische Datentypen ohne Präzision	
REAL	8-Byte Gleitkommazahl, Wertebereich von $3.4 \_ 10^{38}$ bis $1.18 \_ 10^{-38}$
DOUBLE	16-Byte Gleitkommazahl, Wertebereich von $1.79 \_ 10^{308}$ bis $2.23 \_ 10^{-308}$

Alle numerischen Datentypen kennen Sie bereits durch DDS. So können Sie die Integer-Typen mit Typ B abbilden – das heißt: die Deklaration 2B 0 entspricht SMALLINT, 4B 0 entspricht dem Datentyp INT. Und 8B 0 deklarieren Sie im DDS für einen BIGINT-Typ.

Die Datentypen DECIMAL und NUMERIC sind Ihnen wahrscheinlich am geläufigsten. Sie werden im DDS durch S und P deklariert. Die Datentypen REAL und DOUBLE sind das Äquivalent zum DDS-Datentyp F. Um einen doppelten Präzisionswert in DDS zu deklarieren, benötigen Sie das Schlüsselwort FLTPCN.

7.3.3.3.4 DATALINK-Spalten

DATALINKs sind URLs (Hyperlinks), die auf die Position eines spezifischen Objekts verweisen. Die URLs können sowohl eine Web-Seite lokalisieren oder sich auf eine Datei beziehungsweise ein Dokument beziehen, das im IFS – irgendwo im Netzwerk – oder im Internet abgelegt ist. Standardmäßig wird bei einer DATALINK-Spalte als einzige Validitätsprüfung die Basis-Syntax geprüft. Es stehen jedoch einige andere Optionen zur Verfügung, die zwingender sind:

LINKTYPE URL NO LINK CONTROL:
Syntax-Prüfung erfolgt, aber es wird kein Link zur Datei hergestellt.

LINKTYPE URL FILE LINK CONTROL:
Syntax-Prüfung erfolgt; ein Link wird zur Datei hergestellt. Die Datei muss existieren; sie kann nur mit einer Spalte in der Datenbank verlinkt werden.

```
CREATE TABLE ITPSQL.FIRMA     (
                              ......
                              PAGE  DATALINK(256) LINKTYPE URL
                              NO LINK CONTROL
                              NOT NULL WITH DEFAULT
                          )
```

In diesem Beispiel wird die Spalte PAGE als DATALINK-Spalte mit einer maximalen Länge von 256 Zeichen definiert. Diese Länge entspricht der Zeichenanzahl in einer URL. Aufgrund der Spezifikation von **NO LINK CONTROL** wird die Datenbank nicht versuchen, den Zugriff auf die verlinkten Inhalte dieser Spalte zu kontrollieren. Hätte man FILE LINK CONTROL deklariert, dann müsste der *DLFM TCP/IP-Server gestartet werden. Der Server ist Voraussetzung für die Verifizierung der DATALINK-Spalten. Die optionalen Steuerungsattribute, die Sie mit der Klausel **FILE LINK CONTROL** verwenden können, entnehmen Sie bitte dem SQL-Referenzhandbuch. Bedenken Sie auch, dass die Klausel FILE LINK CONTROL die Performance Ihrer Datenbank beeinflusst. Daher sollten Sie versuchen, auf diese Klausel zu verzichten.

7.3.3
Seite 16

7.3.3.3.5 Benutzerdefinierte Datentypen

Bereits in Version V4 wurden benutzerdefinierte Datentypen (User-defined Distinct Type) implementiert. Ein benutzerdefinierter, einzigartiger Datentyp (UDT) basiert immer auf einen Datentyp, der vom System bereitgestellt wird. Allerdings wird er – gemäß spezifischer Anforderungen – angepasst. UDTs können Sie anschließend innerhalb der SQL-Datenbank als eigenständige Datentypen verwenden.

Ein Beispiel soll den Einsatz verdeutlichen:

Für Audiodateien, Videosequenzen oder Bilder wird in der Regel immer der Datentyp BLOB verwendet, obwohl die Struktur und auch die Verarbeitung dieser Daten tatsächlich sehr unterschiedlich ist. Wir könnten jetzt einen eigenen Datentyp definieren, der für alle Spalten eingesetzt wird, die Audiodaten aufnehmen.

```
CREATE DISTINCT TYPE AUDIO AS BLOB (1M)
```

Unser Datentyp AUDIO basiert zwar auf dem SQL-Datentyp BLOB; er ist aber nur 1 MB groß und wird vom System fortan als eigenständiger Datentyp verstanden.

Nach der Erstellung kann ich meinen Datentyp in jedem CREATE TABLE-Statement verwenden:

```
CREATE TABLE  ITPSQL.AUDIOFILE      (
                                ID INTEGER NOT NULL,
                                SPEAK  ITPSQL.AUDIO
                                )
```

Aber Vorsicht! Wer jetzt glaubt, für die unterschiedlichsten Sachverhalte eigene Datentypen zu erstellen, der wird später sehr viel Ärger bekommen. Benutzerdefinierte Datentypen sind eigenständig und erben nicht automatisch die Eigenschaften des zu Grunde liegenden Datentyps. Eindeutige, benutzerdefinierte Datentypen (UDT) leiten sich zwar von vorhandenen Datentypen ab, allerdings erben UDTs nicht deren Funktionalität. Es können nur Funktionen und Operatoren angewendet werden, die für den benutzerdefinierten Typ definiert sind. Was heißt das? Schauen wir uns auch hierzu ein Beispiel an:

```
CREATE DISTINCT TYPE EURO AS DEC(11, 2)
```

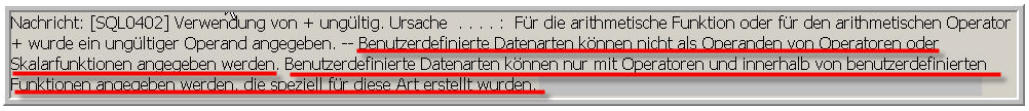
Wir definieren den Datentyp BETRAG. Diesen Typ verwenden wir in einer Tabelle PERSONAL.

```
CREATE TABLE ITPSQL.WERT     (…
                          BETRAG  ITPSQL.EURO
                          )
```

Normalerweise wäre folgendes SELECT-Statement kein Problem:

```
SELECT BETRAG, BETRAG + 100.00
FROM ITPSQL.WERT
```

Jetzt aber erhalten Sie folgende Fehlermeldung:



Fehlermeldung für das SQL Statement

Der Operator „+“ kann zwar auf Spalten vom Typ DEC angewendet werden, jedoch nicht auf unseren Datentyp BETRAG. Wir müssten den benutzerdefinierten Datentyp GEHALT innerhalb der Abfrage konvertieren oder uns eine eigene Funktion „+“ schreiben. Ziemlich viel Aufwand, wenn Sie mich fragen!

Benutzerdefinierte Datentypen sollten Sie also wirklich nur dann einsetzen, wenn Sie exakt diese Wirkung beabsichtigen – das heißt: Sie wollen, dass alle Spalten, die den benutzerdefinierten Datentyp verwenden, auch nur mit benutzerdefinierten Funktionen bearbeitet werden. So stellen Sie sicher, dass keine integrierten SQL-Funktionen verwendet werden können, um Daten zu manipulieren. Diese Situation kann vorkommen – allerdings nicht sehr häufig.

7.3.3.3.6 Identity-Spalten

SQL bietet die einmalige Möglichkeit, autogenerierte Schlüsselfelder mit Identity Columns (Identifikationsspalten) oder mit anderen Objekten zu definieren. Anstelle eines Programms, das eindeutige Artikelnummern oder andere Schlüsselfelder zur Verfügung stellt, liefert jetzt die SQL-Engine eindeutige Schlüsselwerte. Mittlerweile kennt die DB2/UDB viele Möglichkeiten, eindeutige Werte zu erzeugen. Sie können den Datentyp ROWID verwenden, die Klausel GENERATED ALWAYS AS IDENTITY einsetzen oder ein SEQUENZ-Objekt nutzen. Nachfolgend ein kurzer Überblick über die Möglichkeiten und deren Unterschiede.

7.3.3
Seite 18

7.3.3.3.6.1 ROWID-Spalten

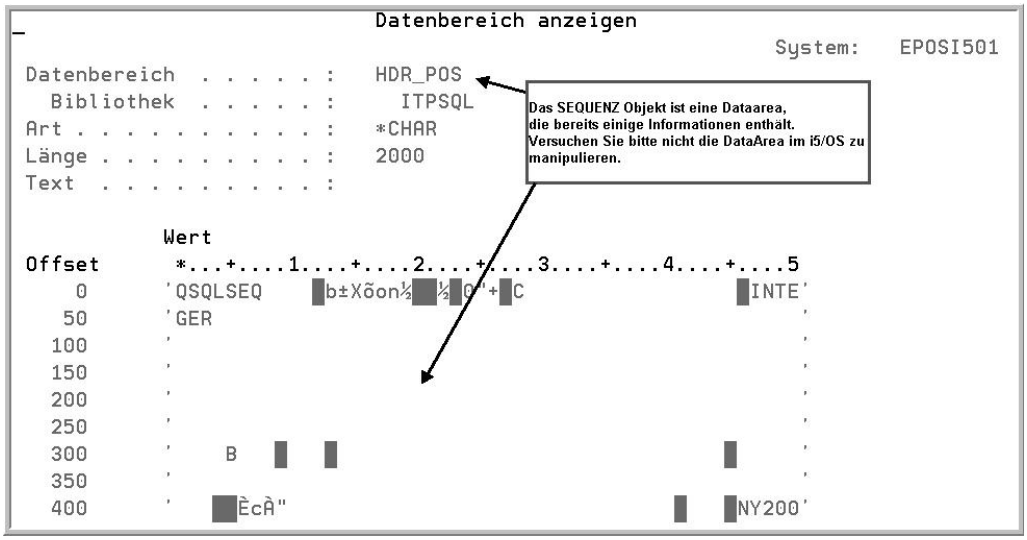
Zeilen-IDs werden verwendet, um eindeutige Schlüsselwerte für jede Zeile in einer Tabelle zur Verfügung zu stellen. Sobald eine neue Zeile in die Tabelle eingefügt wird, erstellt das DBMS einen gültigen Wert, der innerhalb der Tabelle eindeutig ist.

Die Verwendung des Datentyps ist einfach:

```
CREATE TABLE ORT     (
                     OKURZ ROWID GENERATED ALWAYS,
                     PLZ CHAR (10) ,
                     ORTSNAME VARCHAR(150) ALLOCATE(50)
                )
```

Sie sehen, dass ich die Klausel **GENERATED ALWAYS** für die ROWID-Spalte OKURZ verwende. GENERATED ALWAYS ist nicht ausschließlich für ROWID-Spalten zu nutzen, sondern wird generell für IDENTITY-Spalten benutzt, um der SQL-Engine mitzuteilen, dass die Werte dieser Spalte ausschließlich von der Datenbank generiert werden. Das bedeutet im Umkehrschluss, dass Sie auch nicht mit Hilfe eines Programms die Werte der Spalte OKURZ manipulieren können.

Anstelle des SQL-Statements könnte ich auch den iSeries Navigator verwenden:



Datentyp ROWID

ROWID-Spalten sind Binärdaten und werden daher auch nicht automatisch in eine andere CCSID umgesetzt. Es besteht keine Möglichkeit, Einfluss auf deren Wertebereiche zu nehmen.

7.3.3.3.6.2 Die Klausel AS IDENTITY

Die Implementierung der ROWID-Spalten ist gegenüber der Klausel **GENERATED ALWAYS AS IDENTITY** etwas leichter. Allerdings ist – meiner Meinung nach – ROWID jedoch nicht so nützlich wie die Schlüsselspalte. Die Schlüsselspalte erscheint als deutlich lesbarer und als einfach generierter Wert, wohingegen die Zeilen-ID ein 40-Byte großes Bit-Muster ist, was die Abfrage deutlich erschwert.

Wenn wir uns die Deklaration einer Schlüsselspalte anschauen, so stellen wir fest, dass wir wesentlich mehr Angaben machen müssen:

```
CREATE TABLE ORT     (
                    OKURZ INT NOT  NULL
                    GENERATED  ALWAYS  AS  IDENTITY
                    (START WITH 10 INCREMENT BY 10  CYCLE),
                    PLZ CHAR (10) ,
                    ORTSNAME VARCHAR(150) ALLOCATE(50)
                )
```

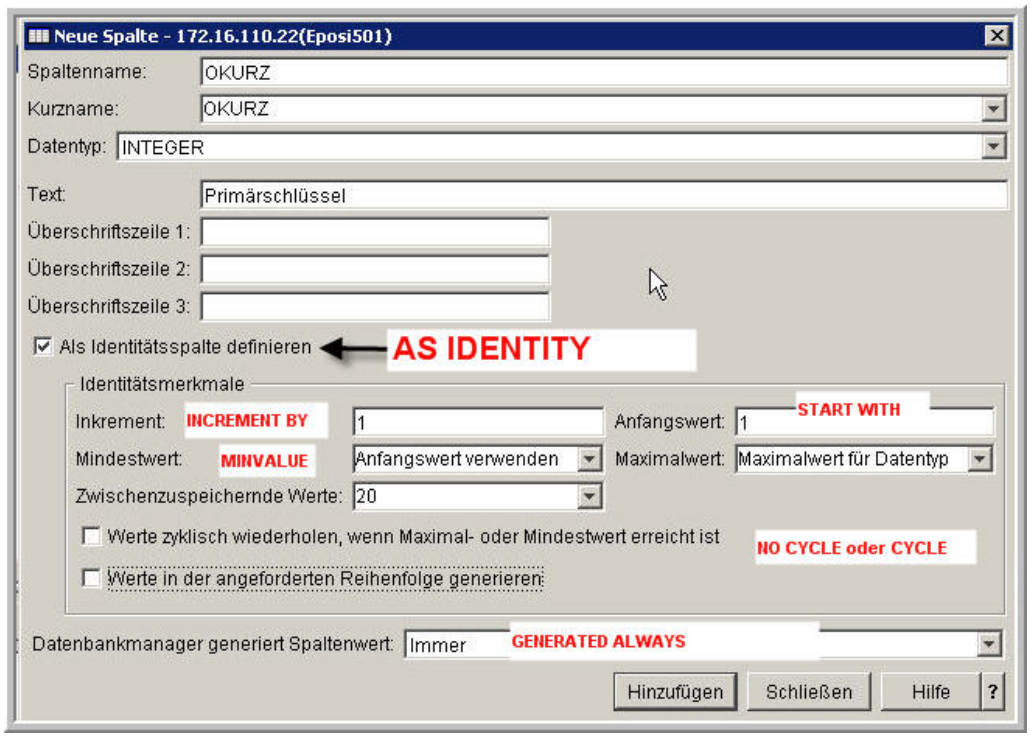
In diesem Beispiel verwende ich für die Definition der Spalte OKURZ die Klausel **GENERATED ALWAYS AS IDENTITY**. Die Klausel ist nur für SMALLINT-, INTEGER-, BIGINT-, DECIMAL- oder NUMERIC-Spalten erlaubt. Insgesamt darf nur eine Spalte innerhalb der Tabelle als IDENTITY-Spalte deklariert werden. Für bestehende Spalten können Sie das Attribut nicht nachträglich implementieren, da bereits Werte erfasst wurden. Der Vorteil der IDENTITY-Spalten besteht darin, dass Sie sowohl den Startwert als auch das Inkrement festlegen können. In meinem Beispiel wird die Spalte OKURZ den Wertebereich 10 bis 2.147.483.640 verwenden, da der Maximumwert in diesem Fall durch den Spaltentypen INTEGER bestimmt wird. Aufgrund des Schlüsselworts CYCLE würde der Startwert 10 wieder verwendet werden, sobald der Maximalwert erreicht ist – unabhängig davon, ob der Wert bereits in der Tabelle existiert. Sie können auch den exakten Wertebereich der IDENTITY-Spalte festlegen:

```
CREATE TABLE ITPSQL.ORT (
     OKURZ INT GENERATED ALWAYS AS IDENTITY
     (START WITH 10      INCREMENT BY 10
     NO MINVALUE   MAXVALUE 100000
     NO CYCLE ),
     PLZ CHAR(10) NOT NULL DEFAULT '' ,
     ORTBEZ VARCHAR(150) ALLOCATE(50) NOT NULL DEFAULT '' ,
```

Hier würde die Spalte OKURZ den Wertebereich 1 bis 100.000 erhalten, wobei die Werte sich jeweils um den Wert 10 erhöhen. Sobald der Grenzwert 100.000 erreicht ist, können keine weiteren Werte erfasst werden, da die zyklische Vergabe der Werte durch das Schlüsselwort NO CYCLE unterbunden ist.

7.3.3
Seite 20

Im iSeries Navigator sieht die Deklaration folgendermaßen aus:



IDENTITY-Spalten im iSeries Navigator deklarieren

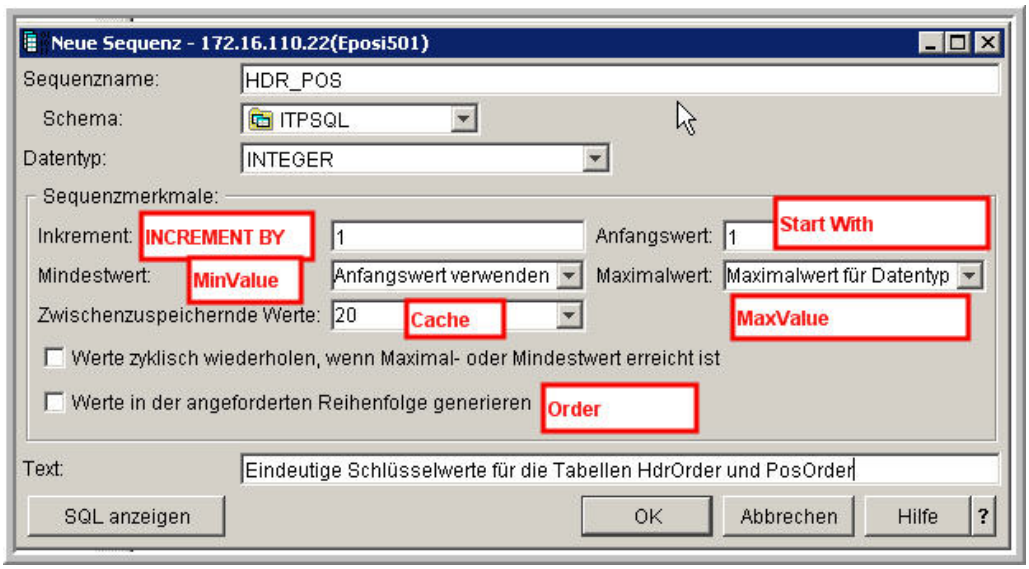
Vorteilhaft ist es auch, dass die Attribute der Identitätsspalten nachträglich geändert oder entfernt werden können.

```
ALTER TABLE ITPSQL.ORT
ALTER COLUMN OKURZ RESTART WITH 1

ALTER TABLE ITPSQL.ORT
ALTER COLUMN OKURZ DROP IDENTITY
```

Solche Schlüsselspalten sind nicht zwingend notwendig, aber sie können die Arbeit ab Release V5R2 deutlich erleichtern.

7.3.3.3.6.3 SEQUENZ-Objekte

Als Letztes schauen wir uns ein SEQUENZ-Objekt an.

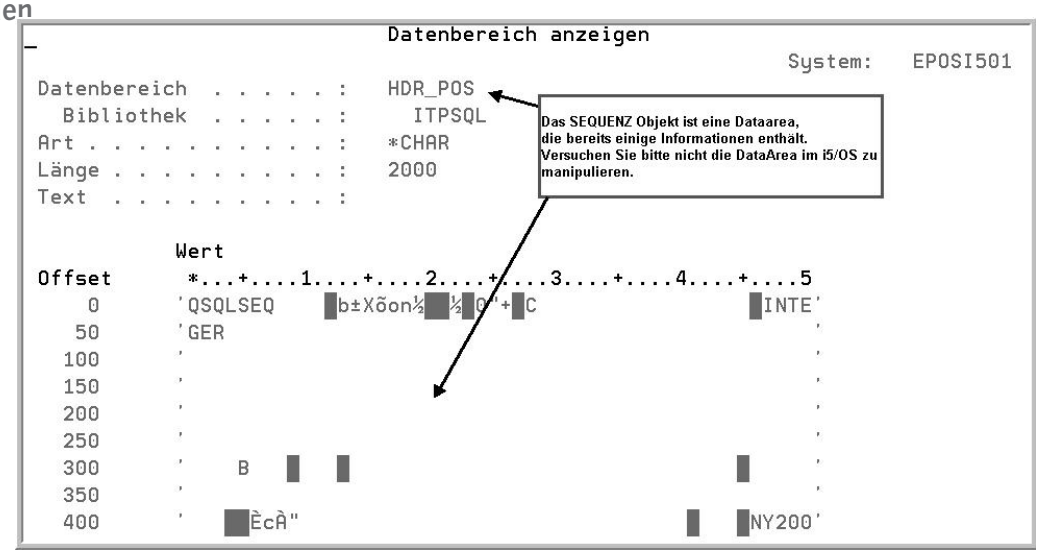


SEQUENZ-Objekt HDR_POS

Die Deklaration erinnert an IDENTITY-Spalten. Auch in einem Sequenz-Objekt können Sie nur numerische Datentypen ohne Nachkommastellen verwenden; sie legen Start- und Endwert sowie das Inkrement fest. Alternativ zum iSeries Navigator steht das entsprechende SQL-Statement zur Verfügung:

```
CREATE SEQUENCE   ITPSQL.HDR_POS
START WITH 1      INCREMENT BY 1
NO MAXVALUE
NO CYCLE
CACHE 20
ORDER
```

In beiden Fällen – egal ob Sie SQL-Statement oder iSeries Navigator verwenden – erhalten Sie ein eigenständiges Objekt – genauer gesagt: Sie erzeugen *DTAARA.



DataArea HDR_POS

Versuchen Sie bitte nicht, DataArea mit dem Befehl CHGDTAARA (Change DataArea) zu verändern. Als Resultat könnten fehlerhafte Schlüsseldaten auftreten, wenn Sie das SEQUENZ-Objekt nutzen.

Bislang habe ich es versäumt, den Parameter CACHE (=Zwischenzuspeichernde Werte) zu erläutern. Das möchte ich nun nachholen. IDENTITY-Spalten und SEQUENZ-Objekte können bereits generierte Schlüsselwerte im Arbeitsspeicher vorhalten. Dadurch können neue Zeilen schneller eingefügt werden, da der SQL-Engine dadurch bereits Schlüsselwerte im Cache zur Verfügung stehen. Der Standardwert ist 20 – das heißt: 20 Schlüsselwerte werden generiert und im Cache bereitgehalten. Wenn alle Werte verbraucht sind oder Sie mit der Klausel NO CACHE arbeiten, ist ein UPDATE von DataArea erforderlich, sobald weitere Schlüssel angefordert werden. Und das kostet halt Zeit!

Im Unterschied zur Klausel AS IDENTITY ist das SEQUENZ-Objekt nicht Bestandteil der Tabelle. Es kann daher mehrere Tabellen parallel mit eindeutigen Schlüsselwerten versorgen. Dabei gibt es keine Verbindung zwischen den jeweiligen Spalten, die das Objekt nutzen, sowie dem Objekt selbst. Vielmehr müssen Sie das Objekt verwenden, wenn Daten in die Tabelle eingefügt werden. Schauen Sie sich das folgende Beispiel an:

```
CREATE TABLE ITPSQL.POSORDER
          (
          POKURZ INT NOT NULL,
          ORKURZ INTEGER,
          ISTDaTUM DATE,
          TEXT VARCHAR(150) ALLOCATE(50) NOT NULL DEFAULT ,‘,
          ReNR SMALLINT,
          ReDat DATE,
          POSORDERINSERTDATE For Column POSINSDAT TimeStamp,
          POSORDERINSERTUSER For Column POSINSUSR CHAR(10)
          NOT NULL DEFAULT ,‘,
          POSORDERUPDATEDATE For Column POSUPDDAT TimeStamp,
          POSORDERUPDATEUSER For Column POSUPDUSR CHAR(10)
          NOT NULL DEFAULT ,‘
          )
;
```

Die Tabelle POSORDER enthält eine Spalte POKURZ. Für diese Spalte wollen wir die Schlüsselwerte durch das SEQUENZ-Objekt generieren lassen. In der Tabellendeklaration können wir keine Verbindung zum Objekt herstellen. Erst wenn wir Zeilen in die Tabelle einfügen, ist es möglich, unseren Schlüsselgenerator zu nutzen:

```
INSERT INTO ITPSQL.PosOrder     (
                    POKURZ,ORKURZ, ISTDATUM, TEXT, ReNr, ReDat,
                    PosOrderInsertDate, PosOrderInsertUser
                    )

VALUES              (
                    NEXT VALUE FOR ITPSQL.HDR_POS, 100, ‘2007-04-10’,
                    ‘Testbedingungen schaffen’, NULL, NULL, Now(), User
                    )
```

Jetzt wird für die erste Spalte der nächste Wert des Objekts HDR_POS eingefügt.

POKURZ	ORKURZ	ISTDATUM
1	100	10.04.2007

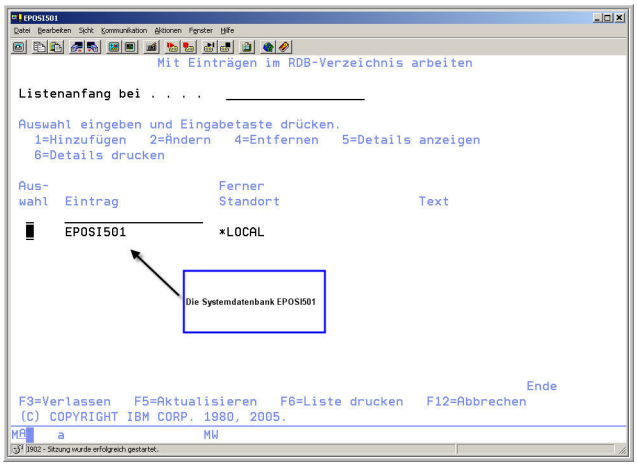
Ergebnis

7.3.3
Seite 24

Es ist auch möglich, den vorangegangenen Sequenz-Wert einzufügen, indem Sie die Klausel PREVIOUS VALUE verwenden:

```
INSERT INTO ITPSQL.POSOrder (
                            POKURZ, .....
                                 :
                            )

VALUES                        (
                            PREVIOUS VALUE FOR ITPSQL.HDR_POS,
                            :
                            )
;
```

Das Objekt SEQUENZ ist sehr vielseitig. Sie können diese Objekte auch nutzen, um beispielsweise konstante Werte zu generieren.



Konstante Werte erzeugen

Außerdem steht mit dem Statement **ALTE SEQUENZ** eine Anweisung zur Verfügung, um

- Startwert,
- Inkrement,
- Minimum- und Maximumwerte,
- Cache-Werte
- und das zyklische Verhalten

der Sequenzen nachträglich zu verändern.

Das war's. Sie haben viele SQL-Datentypen sowie auch einige Klauseln des Statements CREATE TABLE kennengelernt. Wir haben einige Parallelen zum DDS gefunden, aber auch Unterschiede – oder können Sie mit CL-Befehlen SEQUENZ-Objekte erstellen?

Doch fast scheint es so, als hätten wir vor lauter Theorie unsere Beispieldatenbank vergessen. Nein, dem ist nicht so! IBM hat viele Oberflächen implementiert, um mit SQL zu arbeiten. Es ist an der Zeit, diese in den folgenden Kapiteln kennenzulernen.

7.3.3
Seite 26

7.3.3.4 ALTER TABLE – Änderung Ihrer Tabellen

Mit dem SQL-Befehl „ALTER TABLE“ können Sie die Struktur einer bestehenden Tabelle ändern. Spalten können beispielsweise hinzugefügt oder gelöscht, Indizes erzeugt oder gelöscht werden. Sie ändern mit der Anweisung ALTER TABLE den Datentyp bestehender Spalten oder deren Namen. Jede Tabelle einer Datenbank kann mit einem ALTER-Statement geändert werden unabhängig davon, ob Sie die Tabelle mit DDS oder mit SQL erstellt haben. Allerdings benötigen Sie hierfür die entsprechenden Zugriffsrechte auf das Objekt. Das Statement ALTER TABLE können Sie für folgende Aufgaben verwenden:

- Ein Feld hinzufügen oder löschen.
- Die Attribute eines Feldes ändern.
- Integritätsbedingungen hinzufügen oder entfernen.
- Tabellenpartitionen nachträglich hinzufügen, ändern oder entfernen.
- Materialized Query Table (MQT) nachträglich erstellen, ändern oder entfernen.

Schauen wir uns ein kleines praktisches Beispiel an, um zu verstehen, welche Möglichkeiten die ALTER TABLE-Anweisung bietet:

Ich möchte die Spalte „LKZ“ in der Tabelle „ADRESSE“ löschen und in die Tabelle „ORT“ einfügen. Zusätzlich möchte ich den bestehenden Datentyp der Spalte „PLZ“ ändern. Aus einem Zeichenfeld soll ein SMALLINTEGER-Feld werden. Zurzeit wird die Spalte „LKZ“ in der Tabelle „ADRESSE“ als Fremdschlüssel für eine referentielle Integritätsbedingung genutzt. Die referentielle Integritätsbedingung soll anschließend in die Tabelle „ORT“ integriert werden. Zudem bilden die Spalten „PLZ“ und „ORTSNAME“ einen UNIQUE Key. Nachdem die neue Spalte „LKZ“ eingefügt ist, muss der UNIQUE Key um diese Spalte erweitert werden.

Bevor ich beginne die Tabellen zu verändern, möchte ich mir die derzeitige Aktualitäts-ID der Tabelle „ORT“ anschauen. Hierfür verwende ich folgenden i5/OS-Befehl:

Der Befehl DSPFD

Die Aktualitäts-ID lautet:

Aktualitäts-ID der Tabelle ITPSQL.ORT

Als erstes werde ich in der Tabelle „ADRESSE“ die Spalte „LKZ“ entfernen. Um eine Spalte zu löschen, können Sie das folgende Statement verwenden:

```
ALTER TABLE tabellenname DROP COLUMN spaltenname
[RESTRICT oder CASCADE]
```

Beim Löschen eines Feldes aus einer Tabelle können allerdings schnell Fehler auftreten, wenn Indizes oder Sichten von dieser Spalte abhängen. Um dieses Problem zu lösen, stehen zwei verschiedene Klauseln zur Verfügung: RESTRICT und CASCADE.

Die Klausel RESTRICT verhindert die Ausführung der Tabellenänderungen, wenn abhängige Objekte gefunden werden. Die Klausel CASCADE hingegen löscht neben der genannten Spalte auch alle abhängigen Objekte. Für mein Beispiel verwende ich das folgende Statement:

```
ALTER TABLE ITPSQL.ADRESSE DROP COLUMN LKZ    CASCADE;
```

Aufgrund der Klausel CASCADE wird die bestehende referentielle Integritätsbedingung gemeinsam mit der Spalte „LKZ“ aus der Tabelle entfernt.

In einem weiteren Schritt ändere ich den Datentyp der Spalte „PLZ“. Die Syntax für die Änderung einer Spalten-Definition lautet:

```
ALTER TABLE tabellenname
ALTER COLUMN spaltenname
SET DATA TYPE neuer Datentyp NOT NULL  WITH DEFAULT 1.00
```

Die Klausel ALTER COLUMN hat zusätzlich noch zwei Erweiterungen:

Variante 1

```
ALTER TABLE tabellenname ALTER COLUMN spaltenname
DROP DEFAULT
```

Der Standardwert für die spezifizierte Spalte wird auf Null gesetzt.

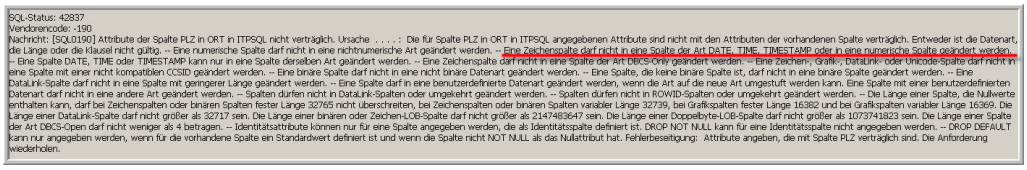
Variante 2

```
ALTER TABLE tabellenname ALTER COLUMN spaltenname
DROP NOT NULL
```

Für mein Beispiel müsste das SQL-Statement demnach folgendermaßen lauten:

```
Alter Table ITPSQL.ORT
Alter column PLZ set data type smallint;
```

Leider wird das Statement nicht fehlerfrei ausgeführt. Vielmehr erhalten wir die folgende Fehlermeldung:



Konvertierungsprobleme

Wenn Datentypen konvertiert werden, müssen die Datentypen natürlich kompatibel sein. Folgende Konvertierungen sind hierbei zulässig:

Ausgangsformat	Zielformat
Decimal	Numeric
Decimal	Bigint, Integer, Smallint
Decimal	Float
Numeric	Decimal
Numeric	Bigint, Integer, Smallint
Numeric	Float
Bigint, Integer, Smallint	Decimal
Bigint, Integer, Smallint	Numeric
Bigint, Integer, Smallint	Float
Float	Numeric

Grundsätzlich ließen sich auch Spalten fester Länge in Spalten variabler Länge konvertieren und umgekehrt. Ist die neue Feldlänge kürzer, könnten allerdings Daten verloren gehen. Ich versuche allerdings – in meinem Beispiel – aus einem Zeichenfeld eine numerische Spalte zu machen und diese Konvertierung ist nicht erlaubt. Folgende Konvertierungen werden grundsätzlich nicht unterstützt:

- Eine numerische Spalte darf nicht in eine nichtnumerische Art geändert werden.
- Eine Zeichenspalte darf nicht in eine Spalte der Art DATE, TIME, TIMESTAMP oder in eine numerische Spalte geändert werden.
- Eine Spalte DATE, TIME oder TIMESTAMP kann nur in eine Spalte derselben Art geändert werden.

7.3.3
Seite 30

- Eine Zeichen-, Grafik-, DataLink- oder Unicode-Spalte darf nicht in eine Spalte mit einer nicht kompatiblen CCSID geändert werden.
- Eine binäre Spalte darf nicht in eine nicht binäre Datenart geändert werden.
- Eine Spalte, die keine binäre Spalte ist, darf nicht in eine binäre Spalte geändert werden.
- Eine DataLink-Spalte darf nicht in eine Spalte mit geringerer Länge geändert werden.
- Eine Spalte darf in eine benutzerdefinierte Datenart geändert werden, wenn die Art auf die neue Art umgestuft werden kann.
  Eine Spalte mit einer benutzerdefinierten Datenart darf nicht in eine andere Art geändert werden.
- Spalten dürfen nicht in DataLink-Spalten oder umgekehrt geändert werden.
- Spalten dürfen nicht in ROWID-Spalten oder umgekehrt geändert werden.

Die einzige Möglichkeit mein Problem zu lösen ist, ich lösche die alte Spalte und füge sie neu hinzu.

```
Alter Table ITPSQL.ADRESSE Drop Column PLZ    CASCADE;
Alter Table ITPSQL.ADRESSE ADD Column PLZ SmallInt
NOT NULL WITH Default 0;
```

Wenn in einer Tabelle ein Feld hinzugefügt wird, dann wird es an das Ende der Zeile angefügt und mit seinem Standardwert initialisiert. Um in einer Tabelle ein Feld hinzuzufügen, verwenden Sie folgende Syntax:

```
ALTER TABLE tabellenname ADD COLUMN neue_spalte CHAR (10)
NOT NULL WITH DEFAULT
```

Mit Hilfe der ALTER TABLE-Anweisung können Sie auch jederzeit nachträglich Integritätsbedingungen entfernen und/oder hinzufügen. Das ist praktisch, denn häufig wird der Bedarf an Begrenzungen erst festgestellt, wenn die Datenbank genutzt wird. Um Datenbank-Bedingungen nachträglich aufzubauen lautet die Syntax:

```
ALTER TABLE tabellenname ADD CONSTRAINT name_constraint
PRIMARY KEY (spaltenname)
```

Offensichtlich wird in diesem Beispiel eine Bedingung für den primären Schlüssel definiert. Man kann jedoch stattdessen jede Art von Datenbankbedingung hinzufügen:

```
ALTER TABLE tabellenname ADD CONSTRAINT name_constraint
FOREIGN KEY( spalte ) REFERENCES uebergeordnete_tabelle
( spalte )
ON DELETE NO ACTION   ON UPDATE NO ACTION ;
```

Natürlich können Sie auch existierende Integritätsbedingungen entfernen. Verwenden Sie hierfür die folgende Syntax:

```
ALTER TABLE tabellenname DROP PRIMARY KEY CASCADE
```

Im hier gezeigten „drop constraint"-Beispiel, wurde das CASCADE-Schlüsselwort verwendet, das genauso funktioniert, wie bei der Löschung einer Spalte. Alle Integritätsbedingungen, die von der gelöschten Bedingung abhängig sind, werden ebenfalls gelöscht. Hätte man stattdessen RESTRICT verwendet, dann wäre die Löschung dieser Integritätsbedingung aufgrund existierender abhängiger Bedingungen möglicherweise verhindert worden. Außerdem benötigen Sie den Namen Ihrer Datenbankbedingung, wenn Sie eine referentielle Datenbankbedingung oder einen UNIQUE Key löschen wollen:

```
ALTER TABLE tabellenname DROP UNIQUE KEY bedingungsname
CASCADE;
ALTER TABLE tabellenname DROP FOREIGN KEY bedingungsname
CASCADE;
```

Grundsätzlich gelten natürlich auch beim ALTER TABLE-Statement die Restriktionen bzgl. der Datenlage, die wir in den vorangegangenen Kapiteln bereits besprochen haben. Weitere Details über Einsatz und Wirkungsweise der Datenbankbedingungen entnehmen Sie bitte auch den vorangegangenen Abschnitten.

Ich verwende jetzt diese Syntax, um zunächst die Spalte „LKZ" in die Tabelle „ORT" einzufügen und um die geforderte Foreign Key-Bedingung zu erstellen. Zusätzlich definiere ich den UNIQUE Key für die Spalten LKZ, PLZ und Ortsname:

```
ALTER TABLE ITPSQL.ORT
ADD COLUMN LKZ INTEGER NOT NULL DEFAULT 0

ADD constraint ITPSQL.ORT_FK_DETAIL_DEKURZ
FOREIGN KEY( LKZ) REFERENCES ITPSQL.DETAIL

ADD constraint ITPSQL.ORT_UK_LKZ_PLZ_ORTSNAME
UNIQUE ( LKZ, PLZ, ORTSNAME) ;
```

Beachten Sie, dass ich das Schlüsselwort CONSTRAINT verwende, um den Integritätsnamen zu benennen. Würde ich den Systemnamen akzeptieren, könnte ich das Statement deutlich verkürzen:

```
ALTER TABLE ITPSQL.ORT
ADD COLUMN LKZ INTEGER NOT NULL DEFAULT 0

ADD FOREIGN KEY( LKZ) REFERENCES ITPSQL.DETAIL

ADD UNIQUE ( LKZ, PLZ, ORTSNAME)
```

Des Weiteren verzichte ich auf die Nennung des Primärschlüsselfeldes. Dies ist möglich, da der Primärschlüssel automatisch gefunden wird. Außerdem verzichte ich auf die Definition der Lösch- und Aktualisierungsregeln und akzeptiere dadurch die Standardeinstellungen.

Auffällig ist auch, dass ich für alle Veränderungen nur ein ALTER TABLE-Statement benötige.

Wenn Sie in einem ALTER TABLE-Statement mehrere Operationen gleichzeitig ausführen, werden die Änderungen entsprechend der genannten Reihenfolge abgearbeitet. Detaillierte Informationen bzgl. der Bearbeitung dieser komplexen Statements finden Sie im IBM SQL Referenzhandbuch.

Bevor Sie beginnen die Tabellen zu verändern, müssen Sie bedenken, welche Auswirkungen das Ändern der Tabellenstruktur auf die Anwendungen hat, welche die Datenbank verwenden. Aufgrund der ALTER TABLE-Anweisung erhält die Tabelle eine neue Aktualitäts-ID. Erinnern Sie sich noch an die Ausgangssituation: Meine Beispieldatei „ORT“ hatte zu Beginn der Änderungen die Dateiebenen-ID 1080201014845. Jetzt hat sich dieser Eintrag verändert:

Dateiebenen-ID nach der Änderung

Aus diesem Grund ist es möglich, dass gängige High-Level-Programme rekompiliert werden müssen, bevor Sie auf die geänderte Tabelle zugreifen können. Denken Sie also bitte daran, dass Änderungen, die Sie mit SQL machen, Auswirkungen auf Ihre RPG-Programme haben können.

Und abschließend ein kleiner Tipp:

Falls andere Programmierer immer noch DDS verwenden, um die Datenbankdateien zu definieren, sollten Sie keine ALTER TABLE-Statements zur Aktualisierung der Tabellenbeschreibungen einsetzen, da diese Änderungen verloren gehen, sobald DDS wieder verwendet wird.

7.3.3.5 COMMENT ON – Dokumentation Ihrer Datenbank

Unter Dokumentation versteht man die Nutzbarmachung von Informationen zur weiteren Verwendung. Ziel der Dokumentation ist es, die dokumentierten Objekte gezielt auffindbar zu machen und Funktionen innerhalb der Datenbank besser vermitteln zu können. Kollegen – Programmierer und Administratoren – können die Funktionen der einzelnen Tabellen und deren Spalten jederzeit ermitteln ohne zuvor den entsprechenden Ordner mit dem Textdokument suchen zu müssen. Zwar ist Papier-Dokumentation gut, interne Dokumentation der Datenbank selbst ist besser. Sie kann nicht verloren gehen oder falsch abgelegt werden und ist jederzeit verfügbar. Mit SQL ist die Datenbankdokumentation relativ einfach. Sie können jederzeit – auch nachträglich – den Objekten Ihrer Datenbank Kommentare hinzufügen. Als Basisanweisung dient hierfür die Anweisung „COMMENT ON“:

```
COMMENT ON TABLE tabellenname IS 'beschreibender Text';
COMMENT ON COLUMN tabellenname.spaltenname IS
'beschreibender Text'
```

Verwechseln Sie aber bitte die Anweisung COMMENT ON nicht mit den 50 Zeichen Texteingabe beim Erstellen einer Tabelle oder deren Spalten. Der Kommentar hat auch nichts mit dem DDS Schlüsselwort COLHDG zu tun. Mit der Anweisung COMMENT ON verfassen Sie Kommentare, die bis zu 2000 Zeichen umfassen können sowohl für die Tabelle als auch für jede Spalte einer Tabelle, so dass Sie vollständige Tabellen- und Spaltendokumentationen hinterlegen können und nicht nur Schlagwörter, die es zu interpretieren gilt.

Schauen Sie sich die folgenden Statements einmal an:

```
comment on table Itpsql.Adresse is
' Die Tabelle Adresse enthält für alle natürlichen und
juristischen Personen die Adressdaten u.a. die
Kundenadressen. Die Darstellung der Kundendaten erfolgt
durch die Verknüpfung mit der Tabelle PERSON und FIRMA';
```

```
comment on table Itpsql.Apa is
' Hier finden Sie die von den Kunden benannten
Ansprechpartner. Die vollständige Darstellung der Daten
erfolgt durch Verknüpfung mit der Tabelle PERSON
und ADRESSE';
```

7.3.3
Seite 34

```
comment on table Itpsql.Firma is
‘ Die Tabelle Firma enthält alle Kundendaten.
Um die vollständigen Informationen zu erhalten
ist eine Verknüpfung mit der Tabelle PERSON
und ADRESSE erforderlich’;
            :
            :
```

Wenn Sie einzelne Spalten dokumentieren wollen, können Sie dies mit dem oben dargestellten Statement tun. Sie müssen lediglich die Klausel „ON TABLE“ gegen die Klausel „ON COLUMN“ austauschen. Ich bevorzuge allerdings eine etwas andere Syntax:

```
comment on column Itpsql.Person
(
pekurz    is    ,PK als Verbindung zur Adresse‘,
vorname   is    ,der Vorname einer Person oder der Firmenname‘,
nachname  is    ,der Nachname einer Person
                oder die Firmenbezeichnung falls das Feld
                Vorname nicht ausreichend ist‘,
sex       is    ,Geschlecht männl., weibl. oder Firma‘,
anrede    is    ,Anrede: Herr, Frau, Firma etc.‘
:
);
```

Diese Beispiele illustrieren, wie man eine Tabelle, eine Spalte oder mehrere Spalten kommentiert. Die Syntax ist denkbar einfach und benötigt keine weiteren Erklärungen. Viel wichtiger erscheint mir die Frage: „Wie können wir die Kommentare lesen?“

Die Kommentare werden nicht angezeigt, wenn Sie sich die Objektbeschreibung einer Tabelle anzeigen – egal ob mit dem Befehl DSPFD (Display file description) oder mit iSeries Navigator.

Um Kommentare zu lesen, führen Sie ein SELECT auf die entsprechenden Katalog-Tabellen aus, die die notwendigen Informationen enthalten. Tabellenkommentare finden Sie im Katalog SYSTABLES und die Spaltenkommentare stehen im Katalog SYSCOLUMNS. Wenn Sie nicht mit einem SQL-Schema arbeiten, sondern mit einer i5/OS-Bibliothek, stehen die Kataloge im Schema QSYS2 zur Verfügung. Falls Sie ein SQL-Schema nutzen, sind die genannten Kataloge darin enthalten:

```
SELECT Table_name as "Tabelle", Long_comment as "Kommentar"
FROM qsys2.SYSTABLES
where Table_schema = 'ITPSQL' ;
```

Tabelle	Kommentar
ADRESSE	Die Tabelle Adresse enthält für alle natürlichen und juristischen Personen die Addressdaten u.a. die Kundenadre
APA	Hier finden Sie die von den Kunden benannten Ansprechpartner. Die vollständige Darstellung der Daten erfolgt du
FIRMA	Die Tabelle Firma enthält alle Kundendaten. Um die vollständigen Informationen zu erhalten ist eine Verknüpfung
PERSON	Hier finden Sie die Namen der Kunden und Ansprechpartner. Um die Adressdaten zu ermitteln, benötigen Sie ein

Datenbankdokumentation anzeigen

Auf diese Weise können Sie sich auch die vollständige Dokumentation einer Tabelle mit den Spalten anzeigen lassen:

```
SELECT b.table_name as „Tabelle“, a.table_schema,
a.long_comment as „Kommentar“,

column_name as „Spalte“, b.long_comment
as „Spaltenkommentar“

FROM qsys2.systables JOIN qsys2.syscolumns
b Using(Table_name)

where a.Table_Schema = ‘ITPSQL’ and
b.Table_Name = ‘PERSON’;
```

Tabelle	Kommentar	Spalte	Spaltenkommentar
PERSON	Hier finden Sie die Namen der Kunden un...	PEKURZ	PK als Verbindung zur Adresse
PERSON	Hier finden Sie die Namen der Kunden un...	VORNAME	der Vorname einer Person oder der Firmenname
PERSON	Hier finden Sie die Namen der Kunden un...	NACHNAME	der Nachname einer Person oder die Firmenbezeichnung falls das Fe
PERSON	Hier finden Sie die Namen der Kunden un...	SEX	Geschlecht männl., weibl. oder Firma
PERSON	Hier finden Sie die Namen der Kunden un...	ANREDE	Anrede: Herr, Frau, Firma etc.

Das Ergebnis der Abfrage

Wahrscheinlich können Kommentare wie diese nicht die komplette Papier-Dokumentation ersetzen, aber SQL-Kommentare können die Papier-Dokumentation sinnvoll ergänzen und insbesondere kritische Informationen werden auf diese Weise effektiv kommuniziert.

Der einzige Wehrmutstropfen besteht darin, dass die Kommentare nicht automatisch erzeugt werden, d. h. jede Dokumentation ist nur so gut wie die Person die sie schreibt. Ich selbst nehme mir auch immer wieder vor, genau zu dokumentieren, um diesen Vorsatz dann aufgrund nicht vorhandener Zeit genau so schnell wieder zu verwerfen. In der Datenbankdokumentation gelingt es mir allerdings seit einiger Zeit, mich selbst ein klein wenig zu zwingen, da der Aufwand relativ gering ist und es sehr viel schneller geht, wenn ich später die Dokumentation den Objekten entnehmen kann anstatt irgendein Dokument zu suchen, das ohnehin nie auf dem neuesten Stand ist. Diese Art der Datenbankdokumentation ist ein Versuch, der sich lohnen könnte. Probieren Sie es einfach mal aus.

7.3.3
Seite 36

7.3.3.6 CREATE VIEW – logische SQL-Sichten erstellen

Die Möglichkeit des Erzeugens von Basisrelationen mit dem CREATE TABLE-Befehl haben wir schon behandelt. Es existieren aber neben den Basisrelationen noch weitere Relationen, so genannte Sichten oder virtuelle Relationen (engl.: View). Dies sind von Basisrelationen abgeleitete Relationen, wobei in der Datenbank nicht deren Inhalt, sondern nur die Ableitungsregeln abgespeichert sind. Grundsätzlich ähnelt eine SQL-View der der logischen Datei, die typischerweise auf der iSeries verwendet wird. Daher ähnelt die Aufgabe einer View auch der Aufgabe einer logischen Datei. Der Sinn von Sichten liegt darin, den Benutzern andere Betrachtungsweisen der Daten zu ermöglichen, ohne dass die Relationen real existieren, wodurch Redundanzen vermieden werden. Auch im Falle des Zugriffschutzes gibt es wichtige Anwendungen: Zum Beispiel wenn nur auf einen Teil einer Relation von Dritten zugegriffen werden darf, können Sie die Zugriffsberechtigung über eine View kontrollieren. Schauen wir uns zunächst ein ganz einfaches Beispiel an:

Nehmen wir einmal an, dass es für die Bearbeitung der Aufträge erforderlich ist, nur die geplanten Aufträge zu selektieren, d.h. Sie benötigen alle Aufträge, die den Status 1 haben. Interessant sind nur die Spalten Firma, Ansprechpartner, Auftragseingangsdatum und die geplante Fertigstellung. Diese Daten können Sie problemlos mit der folgenden Abfrage selektieren:

```
SELECT    Fikurz, ApKurz, Eingangsdatum, Solldatum, Status
FROM      HdrOrder
WHERE     Status = 1
```

Als Ergebnis erhalte ich die folgende Datenmenge:

FIKURZ	APKURZ	EINGANGSDATUM	SOLLDATUM
10	0	12.02.2008	01.04.2008
20	0	02.02.2008	15.03.2008

Die Ergebnismenge

Benötigen Sie diese Datenmenge häufiger, könnte es sinnvoll sein, die Abfrage in einer SQL-View zu speichern:

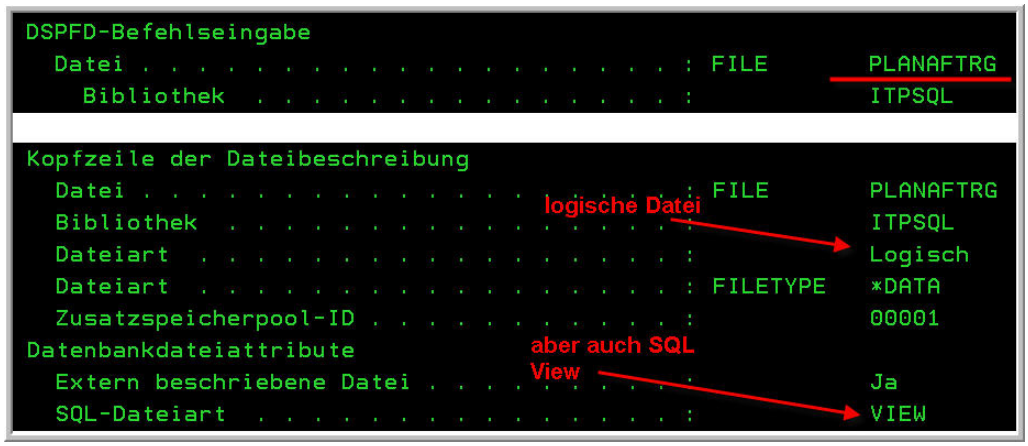
```
CREATE VIEW PlanAftrg
AS
(
SELECT Fikurz, ApKurz, Eingangsdatum, Solldatum, Status
FROM hdrorder
where Status = 1
);
```

7.3.3
Seite 38

Mit diesem SQL-Statements erstellen Sie eine SQL-Sicht, die nicht die Daten enthält, sondern nur die Regeln für den Abruf der Daten. Nachdem die View erstellt ist, kann Sie – mit einigen Ausnahmen – genutzt werden wie jede physikalisch vorhandene Tabelle.

Nutzung der View PlanAftrg

Wie Sie sehen, definiert das View einfach ein SELECT-Statement als neues Gebilde in der Datenbank. Schauen wir uns jetzt einmal an, welches Objekt wir im Betriebssystem i5/OS erzeugt haben:

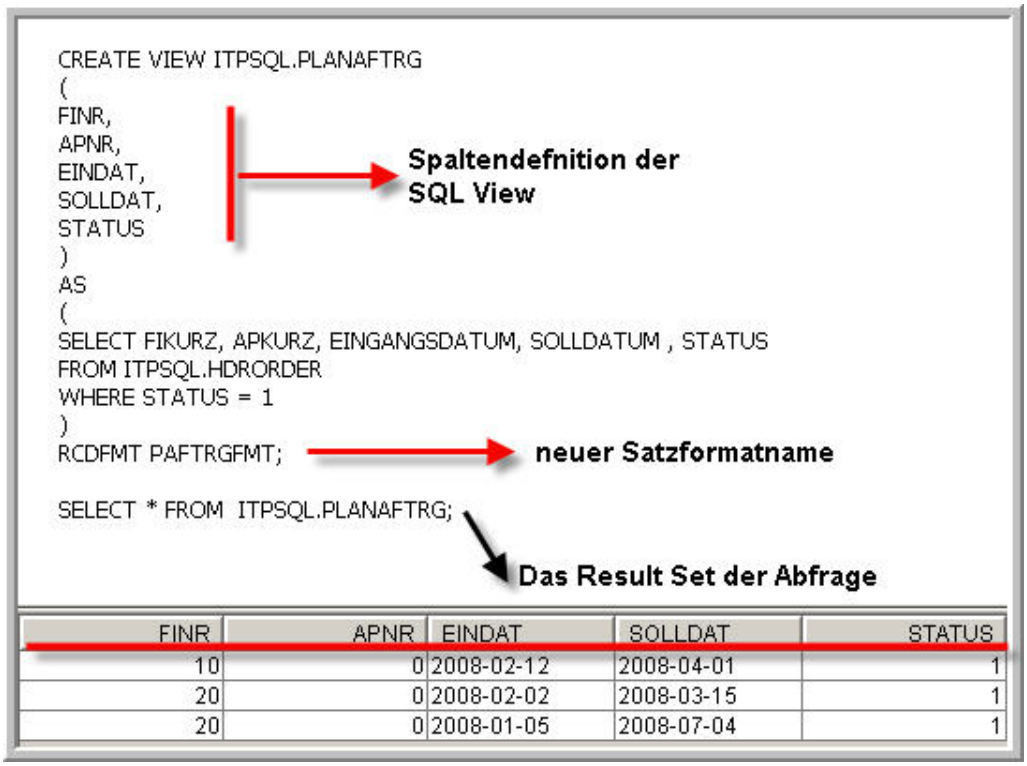


Das Ergebnis im i5/OS

Eine View ist eine logische Datei und kann im i5/OS-Umfeld genauso genutzt werden wie jede logische Datei, die mittels DDS erzeugt wurde. Es ist ohne Probleme möglich, die SQL-View z. B. in einem RPG-Programm zu nutzen. Bis zum Release V5R4 gab es dabei ein paar Schwierigkeiten, weil der Satzname einer SQL-View stets identisch mit dem Dateinamen ist. Als erfahrener RPG-Programmierer werden Sie wissen, dass Sie dann in einem RPG-Programm den Satznamen umbenennen müssen. Seit dem Release V5R4 ist es jedoch ohne weiteres möglich, der SQL-View einen separaten Satznamen zuzuweisen. Dafür müssen Sie lediglich das CREATE VIEW-Statement ein klein wenig erweitern:

```
CREATE VIEW PlanAftrg
AS
(
SELECT Fikurz, ApKurz, Eingangsdatum, Solldatum, Status
FROM HDROrder
where Status = 1
) RCDFMT PAFTRGFMT;
```

Außerdem können Sie die Spalten in einer View individuell benennen:



Spaltendefinition

Sie sehen, dass ich nach dem CREATE VIEW-Statement neue Spaltennamen definiere. Ich denke es versteht sich von selbst, dass die Spaltennamen den SQL-Namenskonventionen entsprechen müssen und dass Sie ebenso viele Spalten benennen müssen wie Ihr Result Set aufweist. Wenn Sie – wie im ersten Beispiel – darauf verzichten die Spalten zu benennen, werden die Spaltennamen aus der Basistabelle abgeleitet. In den meisten Fällen können Sie wählen, ob Sie die Spaltennamen explizit nennen. Nur wenn das Result Set unbenannte Spalten oder doppelte Spaltennamen enthält, haben Sie keine Wahl und werden zur expliziten Nennung der Spaltennamen gezwungen.

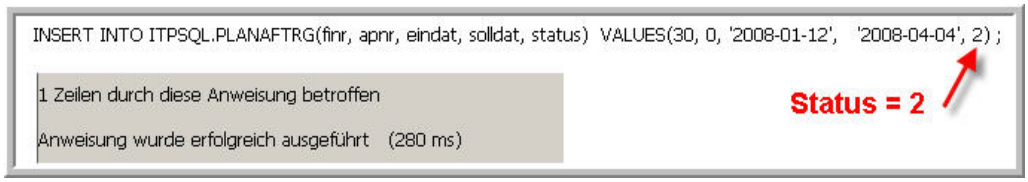
7.3.3
Seite 40

Dennoch gibt es einige entscheidende Unterschiede zwischen einer logischen DDS-Datei und einer SQL View.

- Eine SQL View ist immer eine nicht geschlüsselte logische Datei (nonkeyed logical files). Sie dürfen daher in einer SQL View keine Sortierkriterien hinterlegen, d. h. Sie werden in einer SQL View niemals die Klausel ORDER BY finden. Versuchen Sie es dennoch, erhalten Sie entsprechende Fehlermeldungen. Aus diesem Grund erzeugt eine SQL-Sicht aber auch keinerlei System-Overhead, denn der Zugriffspfad einer SQL View muss seitens des Betriebssystems nicht gepflegt werden. Daher können Sie auch eine beliebige Anzahl SQL-Sichten erzeugen, ohne dass die System-Performance beeinträchtigt wird.
- Außerdem können Sie eine SQL View nicht nur auf Grundlage einer Tabelle erzeugen. Es ist ebenso möglich die SQL View von einer bestehenden SQL View abzuleiten.
- Und noch etwas ist erwähnenswert: Sie sind es gewohnt, dass Sie eine logische DDS-Datei für UPDATE-, DELETE- oder INSERT-Operationen verwenden können. Dies ist bei einer SQL View nicht unbedingt der Fall! Nur wenn die folgenden Kriterien erfüllt sind, können Sie Ihre SQL View für derartige Operationen nutzen:

  – Es darf nur eine Basistabelle geben.
  – Im SELECT Statement existiert keine GROUP BY oder HAVING-Klausel.
  – Das SELECT Statement enthält keine Aggregatfunktion in der Spaltenliste.
  – Der UNION, UNION ALL, EXCEPT oder INTERSECT Operator wird nicht verwendet.
  – Im SELECT Statement darf nicht die DISTINCT Klausel eingesetzt werden.

In meinem Beispiel ist die SQL View „PLANAFTRG“ „Deletable“. Sie weist die oben genannten Merkmale auf und kann daher für INSERT-, UPDATE- und DELETE-Operationen sowohl im SQL- als auch im RPG-Umfeld problemlos genutzt werden wie das nachstehende Beispiel zeigt:



Einfügen einer Zeile in die Tabelle HDRORDER

Eines ist auffällig: Ich kann mit Hilfe der SQL View Zeilen einfügen, die nicht den Selektionskriterien der SQL View entsprechen. In meinem Beispiel füge ich eine Zeile mit dem Statuswert „2“ ein. Wenn Sie dieses Verhalten unterbinden wollen, steht Ihnen die Klausel WITH CHECK OPTION zur Verfügung. Für die Klausel können Sie zwei unterschiedliche Werte verwenden: WITH **LOCAL** CHECK OPTION und WITH **CASCADED** CHECK OPTION. Eines vorweg: Die Klausel WITH CHECK OPTION ist bedeutungslos, wenn die SQL View nicht änderbar ist, d.h. Sie sollten bei einer read-only SQL View auf die Verwendung der Klausel verzichten.

Wirkung und Unterschied zwischen LOCAL und CASCADED lassen sich am einfachsten anhand eines Beispiels erklären. Schauen wir uns an, was passiert:

Ich erstelle meine View „PLANAFTRG“ neu, verwende dieses Mal aber die Klausel WITH LOCAL CHECK OPTION.

```
CREATE VIEW PlanAftrg
AS
(
SELECT Fikurz, ApKurz, Eingangsdatum, Solldatum, Status
FROM HDROrder
where Status = 1
) ;
```

Wiederhole ich jetzt die soeben ausgeführte INSERT-Anweisung, erhalte ich das folgende Ergebnis:

```
INSERT INTO ITPSQL.planaftrg(finr, apnr, eindat, solldat, status)
VALUES(30, 0, '2008-01-12',   '2008-04-04', 2) ;
```

Nachricht: [SQL0161] INSERT oder UPDATE nicht zulässig, da eine Ergebniszeile nicht der Definition von Sicht PLANAFTRG in ITPSQL entspricht. Ursache . . . . : INSERT oder UPDATE konnte nicht ausgeführt werden, da eine Ergebniszeile nicht der Definition von Sicht PLANAFTRG in ITPSQL entsprach. Entweder enthält die Sicht oder eine untergeordnete Sicht eine Klausel WITH CHECK OPTION. Fehlerbeseitigung: Die Daten, die eingefügt oder aktualisiert werden sollen, so ändern, dass sie der Definition für die Sicht entsprechen.

Wirkung der Klausel WITH CHECK OPTION

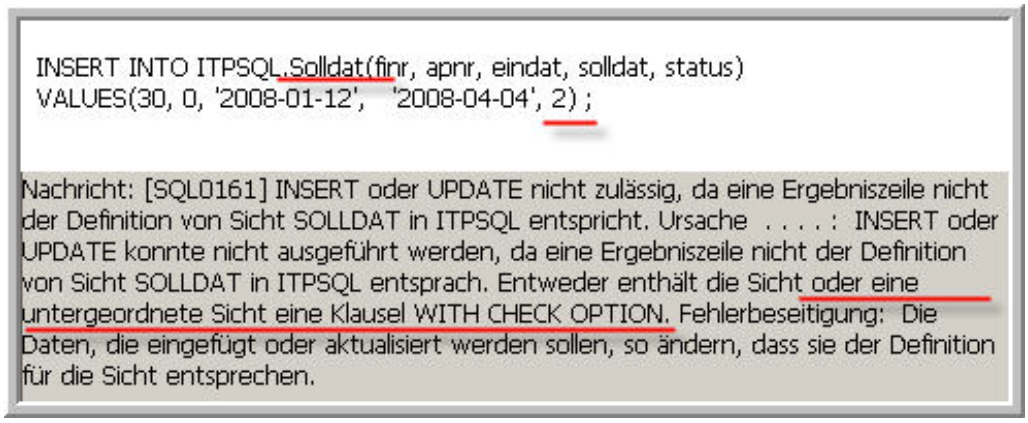
7.3.3
Seite 42

Es ist nicht mehr möglich Werte zu erfassen, die der Definition der SQL View widersprechen. Selbst wenn ich eine weitere View erstelle, die sich von der SQL-Sicht „PLANAFTRG“ ableitet, wird die INSERT-Anweisung abgelehnt. Probieren wir es aus:

Ich erstelle eine zweite SQL View, die mir alle geplanten Aufträge für den Monat April selektiert. Ich nenne die View „SOLLDAT“ und verwende keine WITH LOCAL CHECK OPTION-Klausel:

```
CREATE VIEW ITPSQL.SOLLDAT
AS        (
          SELECT FINR, APNR, EINDAT, SOLLDAT, STATUS
          FROM ITPSQL.PLANAFTRG
          WHERE MONTH(SOLLDATUM) = 4
          ) ;
```

Versuche ich jetzt, die soeben abgelehnte Datenerfassung über die neue SQL View „SOLLDAT“ auszuführen, habe ich auch keinen Erfolg, da die darunter liegenden WITH CHECK OPTION-Klauseln stets berücksichtigt werden.



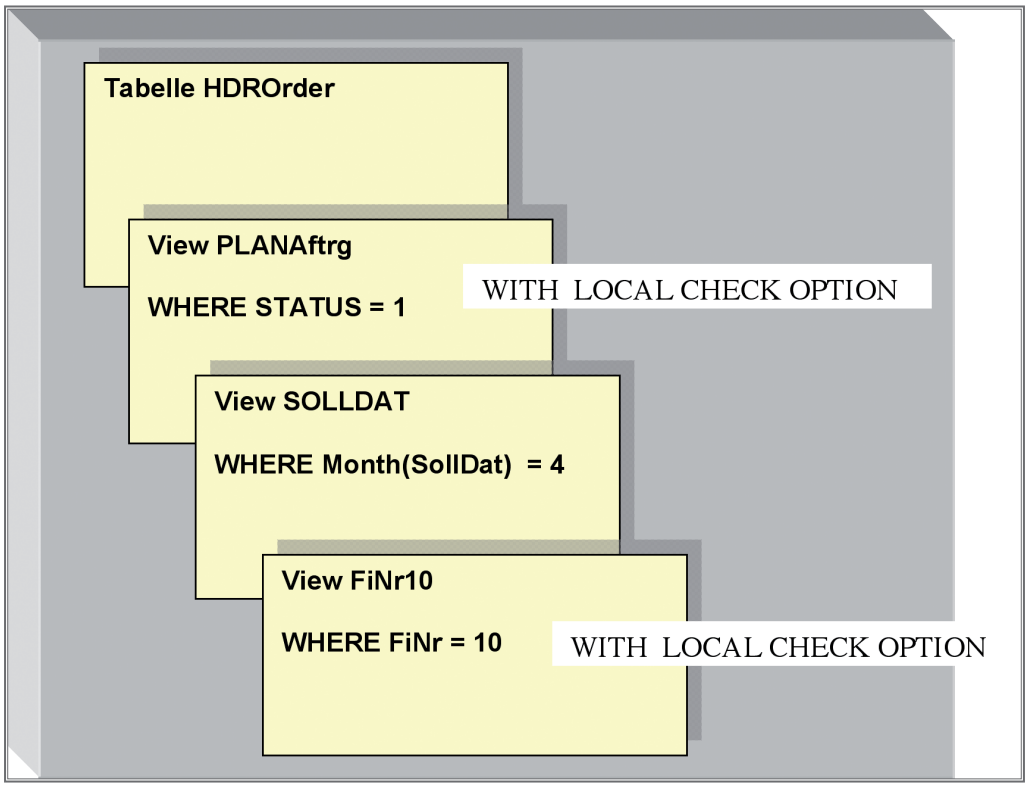
Wirkung der Klausel WITH CHECK OPTION

Bislang ist es unerheblich, ob Sie die Klausel WITH LOCAL CHECK OPTION oder WITH CASCADED CHECK OPTION verwenden. Das Ergebnis ist stets identisch. Erst wenn wir eine weitere View verwenden, können Sie einen Unterschied feststellen.

Ich definiere also eine weitere SQL-Sicht, die mir alle geplanten Aufträge der Kundennummer 10 selektiert, die im Monat April fertig gestellt werden müssen:

```
CREATE VIEW ITPSQL.FINR10
AS        (
          SELECT FINR, APNR, EINDAT, SOLLDAT , STATUS
          FROM ITPSQL.SOLLDAT
          WHERE FINR=10
          ) WITH LOCAL CHECK OPTION;
```

Verdeutlichen wir uns noch einmal unsere Situation:



Die aktuelle Situation

Wenn ich jetzt versuche folgende INSERT-Anweisung auszuführen, werde ich eine Fehlermeldung erhalten:

INSERT INTO ITPSQL.FiNR10 VALUES(10, 0, ‚2008-02-01‘, ‚2008-**04**-10‘, **2**);

Der Wert des Statusfeldes entspricht nicht den Kriterien der View “PLANAUFTRG”. Der INSERT Befehl schlägt fehl. Jetzt verändere ich mein Statement folgendermaßen:

INSERT INTO ITPSQL.FiNR10 VALUES (10, 0, ‘2008-02-01’, ‘2008-**06**-10’, **1**);

Die INSERT-Anweisung wird ausgeführt, obwohl der Fertigstellungsmonat nicht April ist und somit gegen die Bedingung der View „SOLLDAT“ verstößt.

Hier zeigt sich jetzt der Unterschied zwischen LOCAL und CASCADED. Da ich in der SQL View „FiNr10“ die Klausel WITH LOCAL CHECK OPTION verwende, werden nur die Bedingungen der Sichten geprüft, die mit der Anweisung WITH CHECK OPTION erstellt wurden. Anders verhält es sich, wenn ich in der SQL View „FiNr10“ die Klausel WITH CASCADED CHECK OPTION verwende. In diesem Fall müssen die Daten alle Bedingungen der zugrunde liegenden Sichten erfüllen unabhängig davon, ob die jeweilige SQL View mit oder ohne WITH CHECK OPTION-Klausel erstellt wurde.

7.3.3
Seite 44

Mit Hilfe der WITH CHECK OPTION-Klausel können Sie somit gezielt die Bearbeitung einer bestimmten Datenmenge erlauben. Nehmen wir einmal an, Ihre Außendienstmitarbeiter sollen nur auf einen bestimmten Teil der Kundendaten Vollzugriff erhalten, dann könnten Sie die Daten auf verschiedene physische Tabellen verteilen, um die entsprechenden Berechtigungen zu gewähren. Sicherlich eine schlechte Lösung, da Sie für Auswertungen und dergleichen mehr laufend UNION-Anweisungen einsetzen müssten und damit sehr schnell an die physikalischen Grenzen des SQLs stoßen. Eine weitere Möglichkeit: Sie implementieren entsprechende Prüfalgorithmen in den Applikationen. Soweit okay, nur wie verhindern Sie den Zugriff, wenn derartige Prüfalgorithmen nicht integriert werden können? Denken Sie nur an Query/400 oder den QueryManager! Am einfachsten ist es, wenn Sie Ihre Kundendaten logisch durch den Aufbau verschiedener SQL Views unterteilen, die den entsprechenden Mitarbeitern per Definition nur den Zugriff auf einen eingeschränkten Datenbereich ermöglichen. Dabei ist es egal mit welchem Tool der Außendienstmitarbeiter auf die Daten zugreift.

7.3.3.7 CREATE Alias – Alias-Namen erstellen

Alias-Namen sind kurze, gut merkbare Bezeichnungen, die anstelle von langen, schwer zu merkenden Namen benutzt werden. Ein Alias definiert einfach einen Pointer auf eine Tabelle, auf eine View, eine Tabellenpartition oder auf eine spezifische Teildatei innerhalb der Tabelle. DB2 UDB for iSeries implementiert dies als eine DDM File (Distributed Data Management File), die auf das lokale System zurückverweist. DDM Files wurden früher oft für die Kommunikation mit Remote-Datenbanken verwendet; es besteht aber kein Grund, sich mit diesem Konzept weiter zu beschäftigen. Die Erstellung eines ALIAS-Namens ist denkbar einfach:

```
CREATE ALIAS ITPSQL.ZUSATZINFO FOR ZINFO;
```

Nachdem der Alias erzeugt wurde, existiert er als permanentes Objekt in der Datenbank. Die Tabelle ZINFO kann anschließend sowohl mit dem ursprünglichen als auch mit dem Alias-Namen angesprochen werden. Falls die Tabelle, auf die der Alias-Name verweist, nicht vorhanden ist, wenn Sie den Alias-Namen erstellen, erhalten Sie eine Warnung. Der Alias-Name wird aber dennoch erstellt. Erst wenn Sie den Alias-Namen benutzen und die entsprechende Tabelle ist nicht vorhanden, erhalten Sie eine Fehlermeldung und das SQL-Statement wird abgebrochen.

Im iSeries-Umfeld haben Alias-Namen noch eine besondere Stellung. Mit Hilfe der Alias-Namen sind Sie in der Lage, unterschiedliche Member einer Tabelle gezielt anzusprechen. Ein „Member“ bzw. „Teildatei” ist nichts Neues: Jede Tabelle hat mindestens eine Teildatei. Diese Teildateien repräsentieren ein spezifisches Datenset innerhalb der Tabelle. In einigen Datenbanken heißen diese Teildateien Data Sets. Für gewöhnlich dienen Data Sets dazu, Datensatzgruppen logisch zu trennen, um die Performance zu verbessern. Werden beispielsweise alle Umsatzdaten je Monat in verschiedenen Teildateien gebündelt, laufen Abfragen zu den einzelnen Monaten evtl. viel schneller ab, als Abfragen, die erst die gesamte Bewegungsdatei durchsortieren müssen. In manchen Fällen kann diese Art von Verarbeitung noch gerechtfertigt sein, aber im Allgemeinen glaube ich, dass aufgrund von EVIs und besserer Query-Optimierung der Einsatz von Teildateien nicht mehr zwingend ist. Wenn Sie jedoch glauben, dass das Teildateikonzept für Ihre Applikation zwingend ist, oder Sie mit alten Datenbeständen arbeiten müssen, können Sie sich mit Alias-Namen das Leben deutlich erleichtern. Standardmäßig erwartet SQL, dass alle Datenbankdateien nur eine Teildatei haben. Sie können das Problem lösen, indem Sie mit Datenbanküberschreibungen arbeiten. Wenn auch Sie es als ausgesprochen lästig empfinden, wiederholte OVRDBF-Anweisungen (Override Database File) auszuführen, um auf die Teildateien zugreifen zu können, sollten Sie sich entsprechende Aliase anlegen. Anschließend können Sie die Alias-Namen direkt im

SQL-Statement verwenden, ohne dass zuvor ein Override auf die Datenbankdatei zur Selektion der gewünschten Teildatei erforderlich ist. Der wesentliche Vorteil bei dieser Vorgehensweise besteht darin, dass Aliase permanente Objekte sind und somit bestehen bleiben, bis sie gezielt mit der Anweisung DROP ALIAS gelöscht werden. Ein Override hingegen hat nur für die Dauer des aktuellen Jobs Bestand. Auch hier möchte ich einmal exemplarisch die Verwendung der Alias-Namen zeigen. Gehen wir von der folgenden Ausgangssituation aus: Angenommen es existiert eine physische Datei „UMSATZ2006". Die Umsätze eines Monats werden in einzelnen Membern organisiert, d.h. die Tabelle hat 12 Teildateien, eine für jeden Monat. Ob eine physische Datei mehrere Teildateien hat und wie die einzelnen Member heißen, können Sie mit Hilfe des i5/OS-Befehls DSPFD (Display File Description) herausfinden.

Wenn Sie diese Tabelle in einer SQL-Abfrage verwenden, werden Sie immer die erste Teildatei abfragen. SQL kennt keine physischen Dateien mit mehr als einer Teildatei. Das Problem lässt sich relativ einfach lösen, indem Sie für jedes Member einen Alias-Namen einrichten:

```
create Alias Itpsql.JAN2006 for Itpsql.umsatz2006(JANUar)
```

```
create Alias Itpsql.Feb2006 for Itpsql.umsatz2006(februar)
```

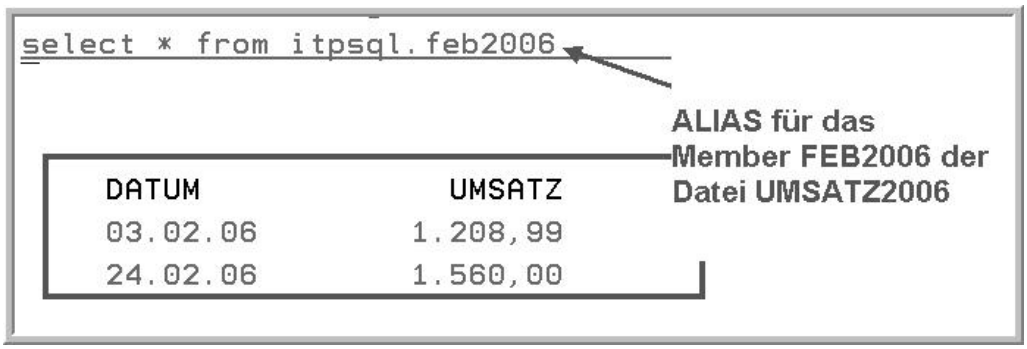
```
create Alias Itpsql.MAE2006 for Itpsql.umsatz2006(MAERZ)
```

Die aus diesen Anweisungen resultierenden Aliases befinden sich anschließend als DDMFs in der Bibliothek ITPSQL:

```
FEB2006      ITPSQL      DDMF
JAN2006      ITPSQL      DDMF
MAE2006      ITPSQL      DDMF
```

Aliases werden DDM-Files

Jetzt können Sie die Alias-Namen direkt im SQL-Statement verwenden.



Verwendung von ALIAS-Namen

Alias-Namen sind für alle SQL-Operationen wie SELECT, UPDATE, INSERT, DELETE, UNION und alle Arten von JOINS etc. zulässig. Zu beachten ist allerdings, dass SQL bei Ausführung der Anweisung CREATE ALIAS keinen Existenztest durchführt und somit auch Alias-Namen erzeugt für Dateien und Teildateien, die nicht existieren. Beim Zugriff auf solche „blinden" Aliases erhalten Sie zur Ausführungszeit eine Fehlermeldung. Falls Sie die physische Datei anschließend erstellen, wird die Datei auch nachträglich dem Alias-Namen zugeordnet.

7.3.3
Seite 48

7.3.3.8 Der DB2 UDB for iSeries Datenbankkatalog

Das DBMS sammelt eine Reihe von Informationen über Ihre Datenbanken und den darin enthaltenen Tabellen. Diese Informationen bezeichnet man als Datenbankkatalog.

Die Katalogsichten liefern Informationen über Tabellen, User-defined Functions, benutzerdefinierte Datentypen (distinct types), Parameter, Prozeduren, Sichten (Views), Indizes, Alias-Namen, Datenbankbedingungen (constraints), Datenbanktriggers, etc. Alle Katalog-Tabellen sind übrigens Sichten (Views). Sie basieren auf den folgenden Tabellen im Schema QSYS: QADBCCST QADBKFLD QADBXSFLD QADBFDEP QADBPKG QADBXTRIGB QADBFCST QADBXRDBD QADBXTRIGC QADBIFLD QADBXREF QADBXTRIGD. Die Basistabellen nennt man auch manchmal Cross-Referenzdateien. Die Öffentlichkeit (*PUBLIC) besitzt im Übrigen keine Rechte an diesen Tabellen und Sie sollten diese Tabellen auch nicht direkt nutzen, sondern immer über die entsprechenden Sichten auf die Informationen zugreifen.

Grundsätzlich lassen sich drei Katalogklassen unterscheiden:

„iSeries Katalogtabellen und -views"
Die Klasse „Katalogtabellen and -sichten" sind an den ANSI- und ISO-Standard angelehnt aber damit nicht identisch. Vielmehr sind sie kompatibel zu früheren i5/OS Releases. Die Sichten dieser Klasse existieren im Schema QSYS und QSYS2 und enthalten alle grundlegenden Informationen über Ihre Tabellen, Parameter, SQL-Prozeduren und Funktionen, Views, Indizes, u. v. m. Wenn Sie ein SQL-Schema erstellen (CREATE SCHEMA), dann werden diese Katalogsichten automatisch auch in Ihrem Schema erstellt. Sie enthalten dann natürlich nur Informationen über die entsprechenden Objekte dieses Schemas. Unter Berücksichtigung dieser Aspekte sollten Sie, wenn Sie Informationen über alle Tabellen im System abrufen möchten, immer die Bibliothek QSYS2 adressieren:

```
SELECT * FROM QSYS2.SYSTABLES
```

Wollen Sie nur Daten für ein bestimmtes Schema sehen, dann verwenden Sie:

```
SELECT * FROM ITPSQL.SYSTABLES
```

Vorausgesetzt es handelt sich um ein SQL-Schema.

Ergänzung 5/2008

7.3.3
Seite 50

„ODBC and JDBC Katalog-Views"
Der ODBC and JDBC Katalog wurde ausschließlich für die ODBC- und JDBC-Schnittstelle entwickelt. Die Tabelle „SQLCOLUMNS" ist z. B. kompatibel mit entsprechenden Sichten für DB2 UDB for z/OS. Die ODBC und JDBC Katalogsichten finden Sie im Schema SYSIBM.

ANSI- und ISO-Katalogsichten
Die ANSI- and ISO-Katalogsichten wurden erstellt, um dem ANSI und ISO-Standard zu erfüllen. Sie werden daher auch modifiziert, wenn einer der Standards dies erfordert. Aus diesem Grund sind manchmal auch Spalten in den Tabellen und Sichten enthalten, die derzeit noch keine Informationen enthalten. Sie wurden bereits für zukünftige Standards implementiert. Es gibt zwei Versionen dieser Kataloggruppe. Zum einen existieren die Katalogsichten in der Bibliothek QSYS2, zum anderen in der Bibliothek SYSIBM. Die Kataloge wurden doppelt implementiert um zu verhindern, dass Anwender Informationen über Tabellen erhalten, für die die Zugriffsrechte fehlen. Über den Katalog in der Bibliothek QSYS2 erhalten Sie nur Informationen, wenn Sie über entsprechende Berechtigungen für die Tabellen verfügen. Nehmen wir einmal an, Sie dürfen auf die Kataloge im Schema QSYS2 zugreifen und Sie wollen Informationen über die Tabelle „APA" im Schema „ITPSQL" selektieren. Allerdings haben Sie keine Berechtigung die Tabelle „APA" zu nutzen.

```
select * from qsys2/tables
where table_schema = 'ITPSQL' and Table_name = 'apa'
```

Sie erhalten keine Informationen, da Ihnen die Berechtigungen für die Tabelle „APA" fehlen. Fragen Sie hingegen den entsprechenden Katalog im Schema SYSIBM ab, erhalten Sie die gewünschten Informationen.

```
select * from SYSIBM/tables
where table_schema = 'ITPSQL' and Table_name = 'apa'
```

Sie müssen also darauf achten, dass Sie nur privilegierten Personen erlauben, die Kataloge im Schema SYSIBM zu nutzen, wenn Sie verhindern wollen, dass unbefugte Personen Informationen über alle Tabellen erhalten können.

Schauen Sie sich bei Gelegenheit einfach mal die Sichten und Tabellen des Datenbankkatalogs an. Sie enthalten viele nützliche Informationen. Wenn Sie den Datenbankkatolog nutzen, sollten Sie allerdings ein paar Regeln beachten:

- Die Daten sind case-sensitive. Sie müssen also auf korrekte Groß- und Kleinschreibung in den Abfragen achten.
- Sie sollten in Ihren Katalogabfragen immer die langen SQL-Namen verwenden und nicht die kurzen i5/OS-Namen. Die Kurznamen werden derzeit in den Katalogen nur gepflegt, um die Kompatibilität zu früheren Releases aufrecht zu erhalten. Die Kurznamen gehören

aber nicht zum SQL-Standard und könnten somit zukünftig aus den Katalogen entfernt werden.

- Die Katalog-Tabelle QSYS.QADBXRDBD enthält alle Informationen über Ihre Datenbankrelationen. Sie sollten insbesondere diese Katalog-Tabelle regelmäßig sichern.
- Wenn Sie eine ILE-Funktion oder eine SQL-Funktion zurücksichern, werden die Informationen bzgl. der Funktionen automatisch in die Katalogtabellen SYSROUTINES und SYSPARMS eingefügt. Das geschieht allerdings nicht, wenn es sich um NICHT-ILE-Prozeduren oder -Funktionen handelt. Um diese Prozeduren oder Funktionen zurückzusichern, müssen Sie die entsprechenden Katalogsichten manuell aktualisieren, indem Sie auch die Katalogsichten zurückspeichern oder aber den Sourcecode nutzen und die Prozedur/Funktion neu erstellen.
- Alle Katalog-Sichten in dem Schema QSYS2 bzw. SYSIBM sind Systemobjekte. D.h. falls Sie die Katalogsichten wiederum für eigene Views nutzen, werden diese SQL-Sichten gelöscht, wenn das Betriebssystem installiert wird. Sie müssen also derartige Views sichern, bevor Sie das Betriebssystem z. B. während eines Releasewechsels installieren. Später können Sie Ihre Sichten dann wieder zurücksichern. Katalogsichten in QSYS sind auch Systemobjekte. Sie werden allerdings nicht gelöscht, wenn das Betriebssystem installiert wird. Es versteht sich daher von selbst, dass es – wenn möglich – geschickter ist, eigene Views auf die Katalogtabellen in QSYS zu legen.

7.3.3
Seite 52

7.3.4 Integritätsbedingungen

Bereits im Kapitel 7.2.4.1 habe ich ausführlich über Vorteile und Wirkung der Integritätsregeln gesprochen. Allerdings wurden dabei ausschließlich native i5/OS-Befehle für die Definition eingesetzt. In den folgenden Kapiteln zeige ich Ihnen, wie Sie die Integritätsregeln mit Hilfe von SQL implementieren können.

7.3.4.1 UNIQUE Constraints

Unique Constraints stellen sicher, dass sich die Daten einer Spalte oder einer Gruppe von Spalten von allen anderen Zeilen der Tabelle unterscheiden. Sie kennen die Wirkung einer UNIQUE-Spalte bereits aus alten DDS-Tagen:

```
A
                          UNIQUE
A                          R ORTFMT
A                             OKURZ            .....
A                             PLZ                    .....
A                             ORTNAM          .....
A                             .....

A                          K PLZ
```

Anschließend haben Sie die DDS-Source mit folgendem i5/OS-Befehl gewandelt:

```
CRTPF FILE(ITPSQL/ORT)
SRCFILE(ITPSQL/QDDSSRC) SRCMBR(ORT)
```

Nun verfügen Sie über eine physische Datei mit der Bezeichnung „ORT“ in der Bibliothek „ITPSQL“. Die physische Datei wird in aufsteigender Reihenfolge eindeutig nach dem Feld „PLZ“ sortiert.

7.3.4
Seite 2

Auch SQL kann Spalten eindeutig definieren. Das Schlüsselwort UNIQUE könnte im CREATE TABLE-Statement folgendermaßen verwendet werden:

```
CREATE TABLE ITPSQL.ORT
            (
            OKURZ …..            ,

PLZ
CHAR(10)             UNIQUE             NOT NULL DEFAULT ' ' ,
                        :
                        :
);
```

Auch jetzt erhalten Sie eine physische Datei namens „ORT“ deren Sätze aufsteigend und eindeutig nach Spalte „PLZ“ sortiert werden. In beiden Fällen wurde also ein eindeutiger Zugriffspfad für die Tabelle „ORT“ erstellt und dennoch gibt es einen entscheidenden Unterschied: Mit der SQL-Anweisung haben Sie eine eindeutige Integritätsbedingung für die Tabelle „ORT“ definiert – mit der DDS-Beschreibung nicht.

Dies lässt sich schnell überprüfen, denn es gibt zum Beispiel die Systemtabelle QSYS2.SYSCST, in der alle Integritätsbedingungen verzeichnet sind.

```
SELECT *
FROM qsys2.syscst
WHERE Table_name like 'ORT%' ;
```

CONSTRAINT_SCHEMA	CONSTRAINT_NAME	CONSTRAINT_TYPE	TABLE_SCHEMA	TABLE_NAME
ITPSQL	Q_ITPSQL_ORT_PLZ_00001	UNIQUE	ITPSQL	ORT

UNIQUE Constraint

Sie sehen, es existiert UNIQUE Constraint für die Tabelle „ORT“. Die Integritätsbedingung hat automatisch den Namen „Q_ITPSQL_ORT_PLZ_00001“ vom System erhalten.

Wenn Sie die systemgenerierten Namen verhindern wollen, müssen Sie die Syntax der SQL-Anweisung nur ein klein wenig ändern:

```
CREATE TABLE ITPSQL.ORT
(
OKURZ …..            ,
PLZ
CHAR(10)  CONSTRAINT ITPSQL.ORT_PLZ   UNIQUE NOT NULL DEFAULT ' ' ,
                        :
                             :
);
```

Jetzt erhält Ihre Integritätsbedingung den Namen „ORT_PLZ“.

Constraints können auf zwei Arten definiert werden: als Spalten-Constraints – wie in meinem Beispiel – oder als Tabellen-Constraints. Ein Tabellen-Constraint gehört nicht zu einer bestimmten Spalte und wird vorrangig verwendet, wenn mehr als eine Spalte für die Bedingung notwendig ist. Jeder Spalten-Constraint kann auch als Tabellen-Constraint geschrieben werden, aber nicht jeder Tabellen-Constraint als Spalten-Constraint. Ein Spalten-Constraint ist vielmehr nur eine alternative Schreibweise, wenn der Constraint sich lediglich auf eine Spalte auswirkt.

Mein Beispiel kann tatsächlich auch nur als Tabellen-Constraint korrekt umgesetzt werden, denn die Spalten „PLZ“ und "Ortsbezeichnung müssen gemeinsam eindeutig sein. Folglich müsste ich folgendes SQL-Statement einsetzten, um die Bedingung korrekt umzusetzen:

```
CREATE TABLE  ITPSQL.ORT
(
OKURZ  …..
PLZ
CHAR(10)
NOT NULL DEFAULT ' ' ,
ORTSNAME FOR COLUMN ONAME              VARCHAR(150) ALLOCATE(50)

NOT NULL DEFAULT '' ,
ORTINSERTDATE FOR COLUMN ORTINSDAT     TIMESTAMP DEFAULT NULL ,
ORTINSERTUSER FOR COLUMN ORTINSUSR     VARCHAR(150) ALLOCATE(50)
          NOT NULL DEFAULT '' ,
ORTUPDATEDATE FOR COLUMN ORTUPDDAT     TIMESTAMP  DEFAULT NULL ,
```

```
ORTUPDATEUSER FOR COLUMN ORTUPDUSR       VARCHAR(150) ALLOCATE(50)
                                         NOT NULL DEFAULT '' ,
CONSTRAINT ITPSQL.ORT_PLZ_OrtsName       UNIQUE (PLZ, ORTSNAME)
);
```

Die RI-Bedingung steht jetzt am Ende aller Spaltendefinitionen und wird durch die Klausel CONSTRAINT eingeleitet. Ich verwende in meinem Beispiel einen benutzerdefinierten Integritätsnamen. Dass es sich um UNIQUE Constraint handelt, erkennen Sie am Schlüsselwort UNIQUE. Dem Schlüsselwort folgen die entsprechenden Spalten, die gemeinsam die UNIQUE-Bedingung erfüllen müssen.

7.3.4.2 Primärschlüssel-Constraints

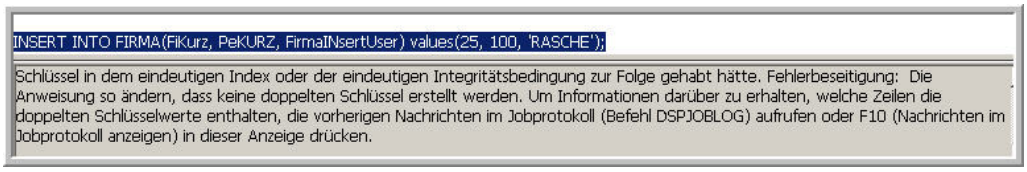
Den physischen Dateien der iSeries können Sie mit Hilfe einer DDS-Beschreibung eine Eingangsreihenfolge zuweisen. Mit SQL gibt es diese Möglichkeit nicht. Sie können eine Datei nicht als Datei mit Schlüsselfolge deklarieren, da SQL eine mengenorientierte und keine dateibasierte Sprache ist. Eine SQL-Tabelle kann jedoch als Tabelle mit UNIQUE- oder Primärschlüsselbedingungen deklariert werden. Der Primärschlüssel besteht aus einer oder aus mehreren Spalten, die eine einzelne Zeile in der SQL-Tabelle eindeutig definieren. Primärschlüssel können während der Erstellung der Tabelle deklariert oder später mit Hilfe der Anweisung „ALTER TABLE“ hinzugefügt werden.

Schauen Sie sich nachfolgendes Beispiel an:

```
CREATE TABLE ITPSQL.FIRMA
(
FIKURZ       INTEGER               NOT NULL  UNIQUE,
PEKURZ       INTEGER               NOT NULL DEFAULT 0 ,
FIRMENZUSATZ FOR COLUMN FZUSATZ    VARCHAR(150) ALLOCATE(50)
                                   NOT NULL DEFAULT '' ,
:
);
```

Die Anweisung erstellt die Tabelle „FIRMA“ im Schema „ITPSQL“. Die Tabelle besitzt einen eindeutigen Zugriffspfad, der aus der Spalte „FiKURZ“ besteht. Das bedeutet, dass keine Einträge in der Tabelle einen identischen „FiKURZ“-Wert haben können. Wenn Sie die beiden folgenden SQL-Anweisungen ausge-

ben, wird die zweite Anweisung fehlschlagen, weil sie gegen die primäre Schlüsselklausel dieser Tabelle verstoßen würde:

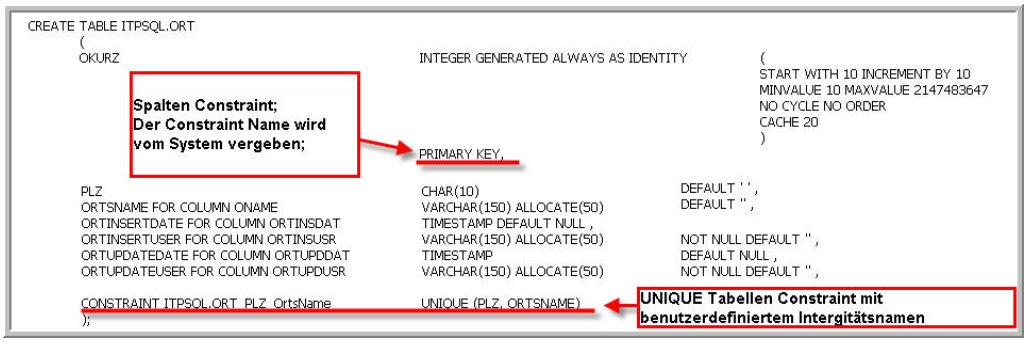


Fehlermeldung aufgrund eines doppelten Primärschlüsselwertes

Das gleiche Ergebnis erziele ich aber auch mit der folgenden Definition:

```
CREATE TABLE ITPSQL.FIRMA
(
FIKURZ INTEGER  PRIMARY KEY,
PEKURZ INTEGER NOT NULL DEFAULT 0 ,
FIRMENZUSATZ FOR COLUMN FZUSATZ        VARCHAR(150) ALLOCAtE(50)
                                       NOT NULL DEFAULT '' ,
:
);
```

Sie sehen, die Syntax der Primärschlüsseldefinition ähnelt der Syntax der UNIQUEConstraints. Lediglich das Schlüsselwort ist ein anderes. Sie können auch Primärschlüssel über mehrere Spalten definieren und selbstverständlich ist es auch möglich für eine Tabelle einen Primär- und mehrere UNIQUE Constraints zu definieren:



Primärschlüssel und UNIQUE Constraint

Die Theorie der relationalen Datenbanken schreibt im Übrigen vor, dass jede Tabelle einen Primärschlüssel haben muss. Diese Regel wird von der DB2/UDB for iSeries allerdings nicht durchgesetzt. Dennoch sollten Sie der Regel folgen, so wie ich es in meinem Datenbankentwurf getan habe.

Desweiteren haben wir festgestellt, dass eine Tabelle mit UNIQUE-Constraint die gleichen Daten aufnehmen kann wie eine Tabelle mit PRIMARY KEY, denn technisch gesehen ist ein Primärschlüssel (Primary Key) einfach eine Kombination aus Unique Constraint mit NOT-NULL-Bedingung. Gemäß ANSI-Standard gibt es allerdings einen entscheidenden Unterschied: NULL-Werte werden beim UNIQUE Constraint als nicht gleich erachtet. Das bedeutet, dass man trotz eines mehrspaltigen Unique Constraint eine unbegrenzte Anzahl Zeilen erzeugen kann, in denen mindestens eine der beschränkten Spalten den NULL-Wert hat. Die DB2/UDB for iSeries folgt diesem Standard nicht. Passen Sie also diesbezüglich auf, wenn Sie eine universelle Datenbank für verschiedene Datenbank-Server erstellen wollen.

7.3.4.3 Fremdschlüssel

Ein Fremdschlüssel-Constraint (Foreign Key) gibt an, dass die Werte einer Spalte oder einer Gruppe von Spalten mit den Spaltenwerten einer anderen Tabelle übereinstimmen müssen. Man sagt, dass dadurch die referentielle Integrität zwischen zwei zusammenhängenden Tabellen gesichert wird.

Nehmen wir die Firmentabelle, die wir schon mehrere Male verwendet haben:

```
CREATE TABLE ITPSQL.FIRMA
(
FIKURZ INTEGER NOT NULL DEFAULT 0 ,
PEKURZ INTEGER NOT NULL DEFAULT 0 ,
FIRMENZUSATZ FOR COLUMN FZUSATZ   VARCHAR(150) ALLOCATE(50)
                                  NOT NULL DEFAULT '' ,
FIRMAINSERTDATE FOR COLUMN FIRINSDAT  TIMESTAMP DEFAULT NULL ,
FIRMAINSERTUSER FOR COLUMN FIRINSUSR  CHAR(10) NOT NULL DEFAULT ' ' ,
FIRMAUPDATEDATE FOR COLUMN FIRUPDDAT  TIMESTAMP DEFAULT NULL ,
FIRMAUPDATEUSER FOR COLUMN FIRUPDUSR  CHAR(10) NOT NULL DEFAULT ' ' ,
CONSTRAINT ITPSQL.FIRMA_PK_FiKurz PRIMARY KEY( FIKURZ )
) ;
```

Wir wissen, dass wir in der Tabelle „HDROrder“ die Aufträge dieser Kunden speichern wollen. Wir wollen sichergehen, dass die Auftragstabelle nur Aufträge von tatsächlich existierenden Kunden aufnimmt. Also definieren wir einen Fremdschlüssel-Constraint in der Auftragstabelle, der sich auf die Kundentabelle bezieht:

```
CREATE TABLE ITPSQL.HDRORDER (
HDKURZ INTEGER CONSTRAINT ITPSQL.HDROrder_PK_HDKurz PRIMARY KEY


DEFAULT 0 ,
FIKURZ INTEGER NOT NULL DEFAULT 0 ,
APKURZ INTEGER NOT NULL DEFAULT 0 ,
                          :
                          :
CONSTRAINT ITPSQL.HDROrder_FK_FIRMA_HdKurz   FOREIGN KEY (FiKurz)
REFERENCES ITPSQL.FIRMA (FiKurz)
 );
```

Jetzt ist es unmöglich, Aufträge mit Firmennummern zu erzeugen, die nicht in der Firmentabelle auftauchen. Man sagt auch, dass die Tabelle „HDROrder“ auf die Tabelle „FIRMA“ verweist. Die Tabelle „HDROrder“ ist somit abhängig vom Inhalt der Tabelle „FIRMA“. Daher bezeichnet man die Tabelle „HDROrder“ auch als CHILD-Tabelle und die Kundentabelle ist die PARENT-Tabelle.

Beachten Sie bitte, dass eine referentielle Integritätsbedingung mit SQL nicht erstellt wird, wenn die bestehende Datenbasis nicht den Bedingungen entspricht. Sie erhalten, wenn Sie SQL einsetzen keine CHECK-PENDING-Situationen.

Die Syntax der FOREIGN Key Constraints verhält sich wieder analog der bereits vorgestellten Syntaxen. So kann ich die Fremdschlüsseldefinition auch abkürzen, indem ich sie als Spalten-Constraint definiere:

```
CREATE TABLE ITPSQL.HDRORDER (
HDKURZ INTEGER CONSTRAINT ITPSQL.HDROrder_PK_HDKurz PRIMARY KEY
DEFAULT 0, FIKURZ INTEGER NOT NULL DEFAULT 0 REFERENCES ITPSQL.
FIRMA (FiKurz),
APKURZ INTEGER NOT NULL DEFAULT 0 ,
                          :
                          :
```

Aber es geht auch noch kürzer, denn wenn Sie die Spaltenliste weglassen, wird der Primärschlüssel der Parent-Tabelle als Zielspalte verwendet. Das gilt natürlich gleichermaßen für Tabellen- und Spalten-Constraint-Definitionen.

```
CREATE TABLE ITPSQL.HDRORDER (
HDKURZ INTEGER CONSTRAINT ITPSQL.HDROrder_PK_HDKurz PRIMARY KEY

                                                DEFAULT 0 ,
FIKURZ INTEGER NOT NULL DEFAULT 0 REFERENCES ITPSQL.FIRMA,
APKURZ INTEGER NOT NULL DEFAULT 0 ,
                          :
                          :
```

Ein Fremdschlüssel kann auch auf eine Gruppe von Spalten verweisen. Die Integritätsbedingung muss dann – wie gezeigt – als Tabellen-Constraint geschrieben werden. Hier folgt ein bedeutungsloses Syntaxbeispiel:

```
CREATE TABLE t1
(
a         integer PRIMARY KEY,
b         integer,
c         integer,
CONSTRAINT ITPSQL.BSP_FK   FOREIGN KEY (b, c)
REFERENCES andere_tabelle (c1, c2)
);
```

Anzahl und Typ der jeweiligen Tabellenspalten müssen natürlich übereinstimmen.

Wenn Sie bereits das Kapitel 7.2.4.1 gelesen haben, dann sollte Ihnen aufgefallen sein, dass wir bisher weder DELETE- noch UPDATE-Regeln in unsere Fremdschlüsselbedingung verwenden. Anders gesagt, wir haben bisher für die Regeln die Default-Einstellungen genutzt. Natürlich können Sie die Regeln im SQL-Statement auch direkt benennen, indem Sie die Klauseln „ON DELETE“ beziehungsweise „ON UPDATE“ einsetzen:

```
CREATE TABLE ITPSQL.HDRORDER
(
HDKURZ INTEGER CONSTRAINT ITPSQL.HDROrder_PK_HDKurz PRIMARY KEY
DEFAULT 0 ,
FIKURZ INTEGER NOT NULL DEFAULT 0
REFERENCES ITPSQL.FIRMA (FiKurz)
ON DELETE RESTRICT  ON UPDATE RESTRICT,
APKURZ INTEGER NOT NULL DEFAULT 0 ,
                          :
);
```

Die Reihenfolge der Klauseln ist hierbei unerheblich. Sie können zwischen den Optionen „RESTRICT“, „CASCADE“, „NO ACTION“, „SET NULL“ und „SET DEFAULT“ in der DELETE-Klausel wählen. Für die UPDATE-Regel stehen Ihnen nur die beiden Werte „RESTRICT“ und „NO ACTION“ zur Verfügung. Die Erläuterungen zur Wirkungsweise finden Sie im Kapitel 7.2.4.1.3.1.

7.3.4.4 Check Constraints

Check Constraint ist der allgemeinste Constraint-Typ. Er erlaubt es Ihnen, einen beliebigen Ausdruck anzugeben, den jeder Wert einer Spalte erfüllen muss. Um zum Beispiel in der Spalte „MWST“ der Tabelle „WERT“ nur die Werte „0.16“, „0.19“ oder „0.07“ zuzulassen, können Sie Folgendes tun:

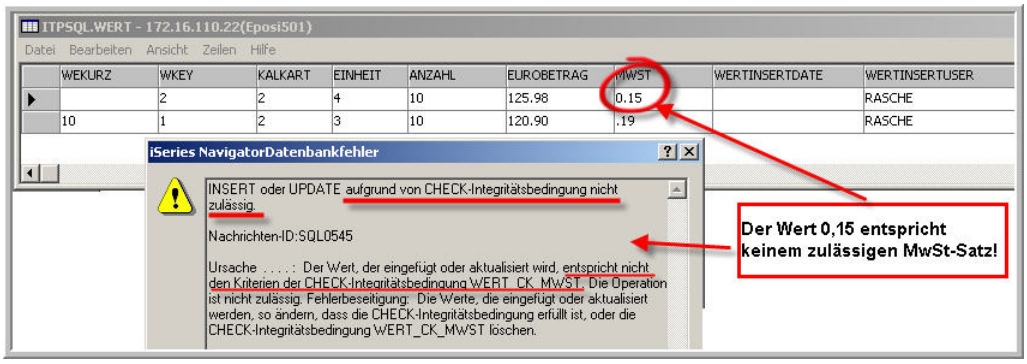
```
CREATE TABLE ITPSQL.WERT (
WEKURZ      INTEGER  …
                    :
                    :
MWST DECIMAL(11, 2) DEFAULT NULL
CONSTRAINT ITPSQL.WERT_CK_MWST        CHECK( MWST IN ( 0.16 ,
0.19 )  ),
WERTINSERTDATE …
);
```

Wie Sie sehen, steht die CHECK-Constraint-Definition als Spalten-Constraint nach der Definition des Default-Wertes. Die Reihenfolge der Klauseln, die Sie auf Spaltenebene einsetzen, können immer in beliebiger Reihenfolge gelistet werden.

7.3.4
Seite 10

Check Constraint besteht aus dem Schlüsselwort CHECK, gefolgt von einem beliebigen SQL-Ausdruck – wobei der Ausdruck geklammert wird. Im Ausdruck sollten Sie die auf diese Art beschränkte Spalte verwenden, ansonsten wäre der Constraint nicht besonders sinnvoll. Außerdem habe ich dem CHECK Constraint einen Namen gegeben, um spätere Fehlermeldungen klarer zu gestalten.

Schauen wir uns jetzt an, was passiert, wenn wir fehlerhafte Daten einfügen:



Check Constraint wird verletzt

Ein Check Constraint kann sich auch auf mehrere Spalten beziehen. Betrachten wir hierzu wieder die Tabelle „HDROrder“.

```
CREATE TABLE ITPSQL.HDRORDER
(
EINGANGSDATUM FOR COLUMN EINDAT  DATE  DEFAULT CURRENT_DATE ,
SOLLDATUM FOR COLUMN SOLLDAT  DATE        DEFAULT NULL ,
ISTDATUM FOR COLUMN ISTDAT     DATE         DEFAULT NULL ,
TEXT VARCHAR(150) ALLOCATE(50)  NOT NULL DEFAULT '' ,
STATUS INTEGER NOT NULL DEFAULT 0
:
);
```

Nehmen wir an, Sie wollen sicherstellen, dass das Auftragseingangsdatum – also die Spalte „EINDAT“ – niemals hinter dem geplanten oder tatsächlichen Fertigstellungsdatum liegen soll.

```
CREATE TABLE ITPSQL.HDRORDER
(
:
EINGANGSDATUM FOR COLUMN EINDAT  DATE  DEFAULT CURRENT_DATE ,
SOLLDATUM FOR COLUMN SOLLDAT  DATE        DEFAULT CURRENT_DATE ,
CONSTRAINT ITPSQL.HDRORDER_CK_EinDat
CHECK(Eindat <= SollDat and eindat <= IstDat)  ,
ISTDATUM FOR COLUMN ISTDAT     DATE         DEFAULT CURRENT_DATE
,
TEXT VARCHAR(150) ALLOCATE(50)  NOT NULL DEFAULT '' ,
STATUS INTEGER NOT NULL DEFAULT 0
CONSTRAINT ITPSQL.HDRORDER_CK_Status CHECK(STATUS IN (1,2,3,4)),
:
HDRORDERUPDATEUSER FOR COLUMN HDRUPDUSR  CHAR(10) NOT NULL DE-
FAULT ' ' ,
CONSTRAINT ITPSQL.HDRORDER_PK_HDKURZ PRIMARY KEY( HDKURZ )
);
```

Zwei Constraints sollten Ihnen bekannt vorkommen. Der dritte verwendet eine neue Syntax. Er steht nicht bei einer bestimmten Spalte, sondern als getrenntes Element in der Spaltenliste. Es ist nicht möglich, diesen CHECK Constraint als Spalten-Constraint zu deklarieren, wenn in der Bedingung zwei verschiedene Spalten der Tabelle benutzt werden. Folglich müssen Sie den Constraint als eigenständige Spalte deklarieren – wie ich es im Beispiel getan habe. Die Reihenfolgen der Spaltendefinitionen und CHECK Constraints ist dabei beliebig. Eine Alternative haben Sie natürlich, indem Sie derartige CHECK-Integritätsprüfungen als Tabellen-Constraint am Ende der Spaltendefinitionen verwenden.

Ob Sie eher Spalten-Constraints verwenden oder Tabellen-Constraints zwischen die Spaltendefinitionen mischen, ist bezüglich der Wirkungsweise völlig egal. Es ist letztendlich Geschmackssache, welche Varianten Sie bevorzugen!

7.3.4
Seite 12

7.3.5 SQL-INDIZES

Was ist ein SQL-Index? Es gibt nicht den SQL-Index, sondern zwei unterschiedliche Varianten. Beginnen wir also mit der Frage: Welche SQL-Indizes gibt es? Ein Index ist eine Sammlung von Pointern, die auf die Sätze einer Tabelle zeigen. Sobald Sie den Index erstellen, erzeugt der Datenbankmanager die Struktur und verwaltet ihn als eigenständiges Objekt. Die Pflege der Indizes kostet Systemressourcen, denn schließlich muss jeder Satz, den Sie in einer physischen Tabelle einfügen, auch in den abhängigen Indizes eingefügt werden. Auch wenn Sie Sätze ändern oder löschen wollen, muss das System die Indizes entsprechend modifizieren. Aber das kennen Sie ja bereits von den logischen DDS-Dateien. In einem Punkt unterscheidet sich der SQL-Index deutlich von den bekannten DDS-Dateien: Sie können den SQL-Index nicht direkt in einer SQL-Abfrage nutzen! Eine logische DDS-Datei kann dagegen problemlos in der FROM-Klausel des SELECT-Statements verwendet werden. Aus Gründen der Performance sollten Sie dies allerdings unterlassen. IBM entwickelte nämlich große Teile des Query Optimizers und Teile der Runtime Engine neu. Man entschied sich, den neuen Query Optimizer (SQE) parallel zum bisherigen Optimizer (CQE) zu installieren, und garantierte dadurch, dass bestehende Applikationen weiterhin problemlos arbeiteten. Gleichzeitig konnten alte und neue Anwendungen von der neuen Technologie profitieren. Der alte Optimizer wird allerdings nicht weiterentwickelt; er verbleibt lediglich aus Kompatibilitätsgründen auf dem System. Von den technologischen Entwicklungen profitieren daher hauptsächlich SQL-Statements, die von der SQE ausgeführt werden. Nutzen Sie allerdings die logischen DDS-Dateien in den SQL-Abfragen, dann wird Ihr SQL-Statement immer von der CQE ausgeführt werden. Sie profitieren dabei also nicht von neuen Technologien.

Ergänzung 7/2008

7.3.5.1 Der Binary Radix Tree Index

In den meisten Datenbanken ist der Standardindex ein Binary Radix Tree Index (Binärbaum). Die DB2 for i5/OS nutzt seit 1988 die Binary Radix Tree Indizes. Hierbei handelt es sich um einen binären Suchbaum. Der binäre Suchbaum ist eine spezielle Baumstruktur, bei der jeder Knoten höchstens zwei Unterbäume – einen linken und einen rechten – besitzt. Ein binärer Suchbaum ist eigentlich nichts anderes als ein sortierter Binärbaum. Er verfügt über eine Wurzel sowie über innere und äußere Knoten. Die Wurzel ist der Grundknoten des Baumes, da sie keinen Vorgänger besitzt. Wenn dieser Knoten leer ist, dann ist der gesamte Baum leer. Äußere Knoten nennt man die, die keinen Nachfolger haben. Sie enthalten die Adresse des physischen Satzes aus der zugrundeliegenden Tabelle. Diese Knoten befinden sich meistens auf der untersten Ebene eines Baumes. Alle anderen Knoten, denen ein Suchbegriff zugewiesen ist, heißen innere Knoten. Jeder innere Knoten hat genau zwei Nachfolger und enthält einen Teil des Schlüsselwertes. Außerdem gilt, dass alle Knoten im linken Teilbaum kleiner und alle im rechten Teilbaum größer sind. Nachfolgend sehen Sie ein Beispiel für die Tabelle „ORT“, Spalte „Ortsname“:

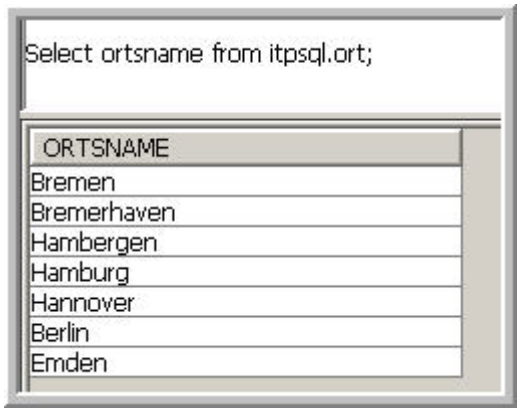
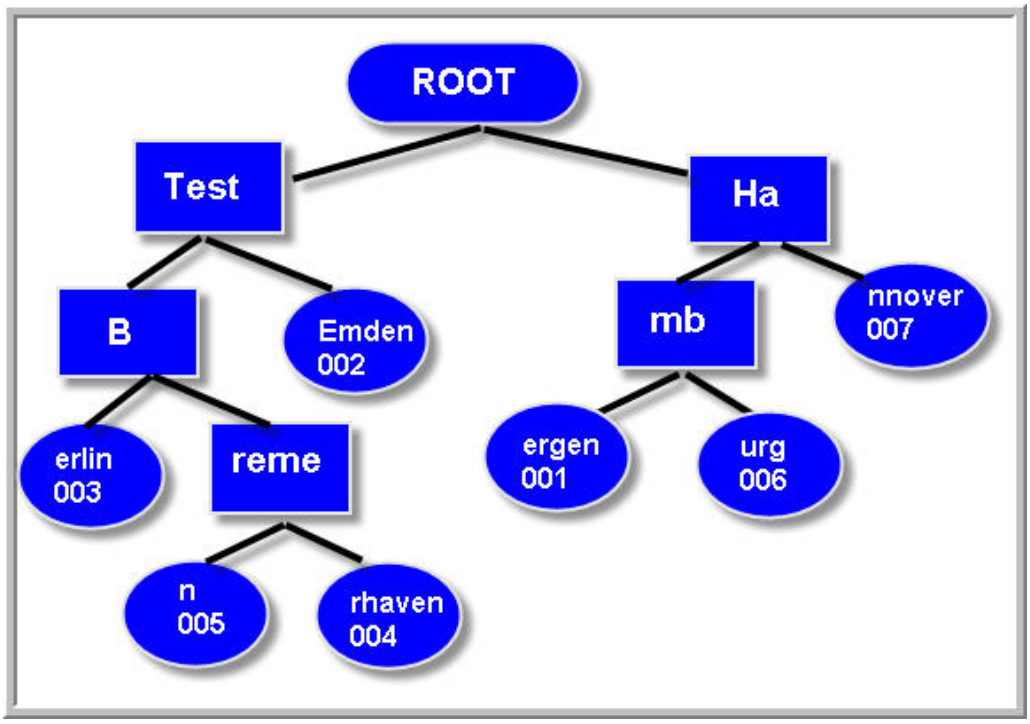


Tabelle „ITPSQL.ORT“

Der zugehörige Binärindex für die Spalte „Ortsname“ könnte folgendermaßen aufgebaut sein:



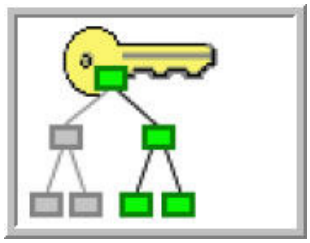
Binary Radix Tree Index „I_Oname“

Angenommen wir suchen jetzt den Ort „Berlin“. Dann wird – ausgehend von der Wurzel – überprüft, ob der jeweilige Knoten kleiner oder größer ist als der gesuchte Wert. Wenn er kleiner ist – wie in unserem Beispiel –, dann wird im linken Teilbaum weitergesucht, wenn er größer ist, im rechten. So kann bei jedem Suchschritt ein ganzer Teilbaum von der Weitersuche ausgeschlossen werden. Zum Einsatz kommen dabei stets binäre Suchalgorithmen. In dieser Baumstruktur können große Mengen unterschiedlicher Schlüsselwerte effizient gespeichert werden. Beim Sortieren und Selektieren von Daten kann sich ein Binärindex als ausgesprochen nützlich erweisen.

Die SQE kann die Binärindizes für den Datenzugriff unterschiedlich nutzen.

Index Radix Probe

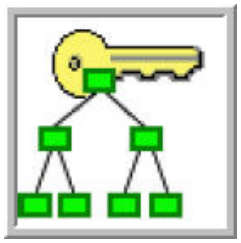
Wenn der Optimizer die Methode „Radix Index Probe“ verwendet, dann stehen die benötigten Sätze im Mittelpunkt des Zugriffs. Mit wenigen Zugriffen kann der Index geprüft und selektiert werden. Vor dem Lesen des jeweiligen Schlüsseleintrags erfolgt in diesem Fall die Überprüfung der Selektionskriterien. Dies reduziert deutlich die Anzahl der zu verarbeiteten Datensätze, denn nur Sätze, deren Spaltenwerte den lokalen Selektionskriterien entsprechen, werden in den Arbeitsspeicher geladen. Sie können in einem zweiten Schritt gezielt anhand der relativen Satznummern aus der Tabelle abgerufen werden. Die Sätze stehen in Schlüsselfolge sortiert zur Verfügung.



Visual Explain Icon Index Radix Probe

Index Radix Scan

Wenn der Optimizer hingegen die Methode „Index Radix Scan“ nutzt, entfällt die vorzeitige Prüfung der Selektionskriterien. Alle Einträge des Index werden in den Arbeitsspeicher geladen. Das ist häufig der Fall, wenn die lokalen Selektionskriterien nicht vollständig den Schlüsselspalten des Index entsprechen. Dennoch bringt auch diese Methode Vorteile. Da die geladene Seite eines Index wesentlich mehr Sätze abbildet als die vergleichbare Seite der Tabelle, reduzieren sich die I/O-Kosten deutlich. Außerdem stehen die Daten auch hier anschließend in sortierter Schlüsselfolge zur Verfügung.



Visual Explain Icon Index Radix Scan

7.3.5.2 Binärindexerstellung

Sie haben schon häufig Binärindizes erstellt, denn auch unsere, mit DDS erstellten logischen Dateien sind binäre Suchbäume. Hier ein Beispiel:

```
                                                                D_ORT
..A.Name++++++.LängeDDsF......Funktionen++++++++++++++++++
*** Datenanfang ****************************************
  R ORTFMT_                    PFILE(ITPSQL/ORT)
  K ONAME
```

Logische DDS-Datei „D_Ort“

Das Objekt wird mit dem Befehl „Create Logical File“ (CRTLF) erstellt. Wenn ich allerdings einen SQL-Index erstelle, nutze ich das SQL-Statement „CREATE INDEX“. Das Statement ist einfach zu handhaben, denn es enthält nur wenig unterschiedliche Klauseln:

```
CREATE INDEX I_Ortsname ON ORT (Oname);
```

```
Label on Index I_Ortsname
IS ‚Schlüssel: Ort NICHT eindeutig‘;
```

Mit dem zweiten SQL-Statement hinterlege ich noch einen beschreibenden Text, der vergleichbar mit dem DDS-Schlüsselwort „TEXT“ ist. Der eben erstellte Index ist kein UNIQUE-Index. Doch auch eindeutige Indizes lassen sich schnell erstellen:

```
CREATE UNIQUE WHERE NOT NULL INDEX I_LKZ_PLZ_ORT
ON ORT(LKZ, PLZ, ORTSNAME);
```

```
Label on Index I_LKZ_PLZ_ORT
IS ‚Schlüssel eindeutig‘;
```

Die Klausel „WHERE NOT NULL“ schließt NULL-Werte für die Spalten aus. Den physischen Dateien können Sie mit Hilfe einer DDS-Beschreibung eine Eingangsreihenfolge zuweisen. Bei SQL gibt es diese Möglichkeit nicht. Sie können eine Datei nicht als Tabelle mit Schlüsselfolge deklarieren – wie Sie bereits im Kapitel 7.3.3 erfahren haben. Eine SQL-Tabelle kann jedoch – wie Sie wissen – als Tabelle mit UNIQUE- oder Primärschlüsselbedingungen deklariert werden. Nähere Informationen hierzu enthalten die Kapitel 7.3 4.1 und 7.3.4.2.

7.3.5
Seite 6

Schauen wir uns in einem nächsten Schritt an, ob es Unterschiede zwischen den SQL-Indizes und logischen DDS-Dateien gibt. Geben Sie hierfür auf einer 5250-Befehlszeile folgenden Befehl ein:

```
WRKF FILE(ITPSQL/*ALL) FILEATR(LF)
```

Daraufhin erhalten Sie eine Bildschirmausgabe mit den vorhandenen logischen Dateien im Schema „ITPSQL“:

```
Ausw  Datei       Bibliothek  Attribut    Text
 __   __________  __________
 __   D_ORT       ITPSQL      LF          Ortsname     KeyValue: Ort
 __   I_LKZ00001  ITPSQL      LF          Schlüssel eindeutig
 __   I_ORTSNAME  ITPSQL      LF          Schlüssel: Ort NICHT eindeutig
```

Ergebnis des Befehls „WRKF“

Auf den ersten Blick kann man keine Unterschiede zwischen den Indizes und den logischen DDS-Dateien feststellen. Tatsächlich können Sie die Objekte unabhängig von Ihrem Erstellungsbefehl administrieren und auch in Ihren HLL-Programmen wie gewohnt einsetzen. Dennoch gibt es Unterschiede:

- Das Dateiattribut „Pagesize“ bestimmt die logische Seitengröße des Zugriffspfades. Sie beträgt für SQL-Indizes in der Regel 64 KB. Logische DDS-Dateien haben häufig 8 KB große Seite. Die Seitengröße des Zugriffspfads bestimmt letztendlich die Anzahl Bytes, die aus dem Zusatzspeicher in den Speicher-Pool des Jobs für eine Fehlseite verschoben werden. Diesen Parameter finden Sie auch im Befehl „Create Logical File“ (CRTLF) in den erweiterten Parametern. Der Unterlassungswert „*KEYLEN“ sorgt dafür, dass die Seitengröße anhand der Schlüssellänge ermittelt wird. Die logische Seitengröße für den Zugriffspfad eines SQL-Index definieren Sie im Statement „CREATE INDEX“ mit der Klausel „PAGESIZE“:

  ```
  CREATE UNIQUE WHERE NOT NULL INDEX I_LKZ_PLZ_ORT
  ON ORT(LKZ, PLZ, ORTSNAME) PAGESIZE 64;
  ```

  Der Unterlassungswert für einen SQL-Index ist 64 KB.

Es gibt einen weiteren wichtigen Unterschied, den Sie kennen sollten, bevor Sie beginnen, Indizes zu erstellen. SQL-Indizes können sich die Zugriffspfade in der Regel nicht mit anderen Binärindizes teilen.

Schauen Sie sich dazu die folgenden Beispiele an:

Ich erstelle als Erstes die logische DDS-Datei „D_PLZ_ONAME“ mit den Feldern „Postleitzahl“ (PLZ) und „Ortsname“ (ONAME). Anschließend wird die logische DDS-Datei „D_PLZ“ erzeugt, die nur das Schlüsselfeld „PLZ“ enthält. Um später die erstellten Objekte besser unterscheiden zu können, wähle ich für alle logischen DDS-Objekte das Präfix „D“ und für alle Indizes das Präfix „I“ im Objektnamen.

Schauen wir uns jetzt die Dateibeschreibung der logischen Datei „D_PLZ“ an:

```
Anzahl eindeutiger Teilschlüsselwerte . :              Datei "D_PLZ"
  Schlüsselfeld 1 . . . . . . . . . . . :                       1
  Schlüsselfelder 1 - 2 . . . . . . . . :                       7
Datei, der der Zugriffspfad gehört  . . :       ITPSQL/D_PLZ_ORT
Teildatei . . . . . . . . . . . . . . . :       D_PLZ_ORT
```

Zugriffspfadnutzung der logischen DDS-Datei „D_PLZ“

Sie sehen, die Datei „D_PLZ“ nutzt den bestehenden Zugriffspfad der Datei „D_PLZ_ORT“. Es entsteht somit auch kein System-Overhead, da weiterhin nur ein Zugriffspfad vom Datenbankmanager gepflegt werden muss.

Das gleiche Beispiel führe ich mit SQL aus:

```
CREATE INDEX I_plz_oname ON ort(plz, oname);
```

```
CREATE INDEX I_plz ON ort(plz);
```

Schauen wir uns auch hier die Eigenschaften der erstellten Objekte an – als Erstes den Index „I_plz_oname“:

```
Indexgröße  . . . . . . . . . . . . . . . :       Datei "I_PLZ_ONAME"
Zugriffspfad gültig . . . . . . . . . . . :       Ja
Impl. gemeinsame Ben. des Zugriffspfads :         Nein
Zugriffspfad aufgezeichnet  . . . . . . . :       Nein
```

Zugriffspfadnutzung SQL-Index „I_plz_oname“

7.3.5
Seite 8

Obwohl die logische DDS-Datei „D_plz_oname“ mit diesen Schlüsselfeldern bereits existiert, wird ein eigenständiger Zugriffspfad für den SQL-Index aufgebaut. Aufgrund der unterschiedlichen logischen Seitengröße können die erstellten Indizes nicht die Zugriffspfade der logischen DDS-Dateien nutzen. Und wie sieht es mit dem zweiten SQL-Index aus?

Zugriffspfadnutzung SQL-Index „I_plz“

Auch für diesen Index wird ein eigenständiger Zugriffspfad erstellt. Anders als logische DDS-Dateien nutzt der SQL-Index nur bestehende Zugriffspfade, wenn sie zu 100 Prozent deckungsgleich sind. Es existieren hier also zwei SQL-Zugriffspfade, die beide gepflegt werden müssen, und zusätzlich der Zugriffspfad der logischen DDS-Dateien. Ich denke, das Beispiel zeigt deutlich, wie schnell Sie Ihre Datenbank belasten und unnötigen System-Overhead produzieren können. Es existieren jetzt drei Zugriffspfade auf dem System, obwohl tatsächlich nur einer notwendig wäre: Es ist ausreichend, den SQL-Index „I_fikurz_apkurz“ mit den Schlüsselfeldern „fikurz“ und „apkurz“ zu erstellen.

Wenn es – aus welchen Gründen auch immer – erforderlich sein sollte, weiterhin mit DDS-Dateien zu arbeiten, sollten Sie die logischen DDS-Dateien löschen und neu erstellen. Passen Sie auf, was passiert, wenn ich die Dateien „D_plz“ und „D_plz_oname“ zunächst lösche:

```
DLTF FILE(D_plz)
DLTF FILE(D_plz_oname)
```

und anschließend neu erstelle:

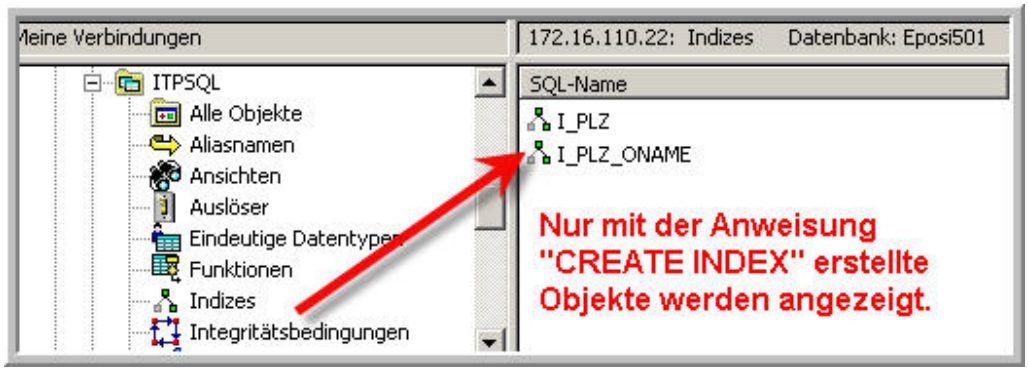
```
CRTLF FILE(D_plz_oname) SRCFILE(QDDSSRC)
CRTLF FILE(D_plz) SRCFILE(QDDSSRC)
```

Beide Objekte nutzen jetzt den bestehenden Zugriffspfad des SQL-Index:

Dateiattribute der logischen DDS-Datei „D_PLZ“

Logische DDS-Dateien können aufgrund der kleineren Seitengröße die Zugriffspfade bestehender SQL-Indizes nutzen.

Erschwerend kommt hinzu, dass es bisher keine gute Darstellung gemeinsam genutzter Zugriffspfade gibt. Auch System i Navigator ist hier nicht besonders hilfreich! Öffnen Sie Ihre Datenbank sowie das Schema, das die Indizes enthält. Klicken Sie auf den Eintrag „Indizes“. Es werden nur die SQL-Indizes angezeigt!



System i Navigator V5R4 Eintrag „Indizes“

Unsere logischen DDS-Dateien finden wir im Eintrag „Ansichten“.

System i Navigator V5R4 Eintrag „Ansichten“

7.3.5
Seite 10

SQL versteht logische DDS-Dateien in erster Linie als SQL-Ansicht (VIEW), weil sie beispielsweise eingeschränkte Satzformate darstellen und Selektionskriterien enthalten können. Das sind Funktionen, die ein SQL-Index nicht abbilden kann. Bis Release V5R4 bietet System i Navigator keine Möglichkeit, sich die gemeinsame Zugriffspfadnutzung anzuzeigen. Hier bleibt momentan nur die Rückkehr zur 5250-Befehlszeile. Etwas besser wird die Darstellung im Release V6R1:

SQL-Name	Schlüsselspalten	Typ des Eignerindex	Name des Eignerindex
I_CUSKDNR	KDKDNR (Auf)	Integritätsbedingung über Prim...	P_KDKDNR
I_CUSORT	KDORT (Auf)	Index	I_KDORT
I_HIANL	HIANL (Auf)	Index	I_HIANL
I_KDKDNR	KDKDNR (Auf)	Integritätsbedingung über Prim...	P_KDKDNR
I_KDORT	KDORT (Auf)	Index	I_KDORT
I_LOWER_HIANL	LOWER (HIANL) (Auf)	Index	I_LOWER_HIANL
I_MED_KD	HIMED (Auf), HIKDN...	Index	D_MED_KDNR
I_PLUS_BSTQTY...	PRDQTY + PRDBSTQ...	Index	I_PLUS_BSTQTY_QTY
S_KDORT	KDORT (Auf)	Index	S_KDORT

System i Navigator-Darstellung gemeinsam genutzter Zugriffspfade

System i Navigator kann seit Release V6R1 für SQL-Indizes die genutzten Zugriffspfade darstellen. Der genutzte Zugriffspfad heißt „Eignerindex“.

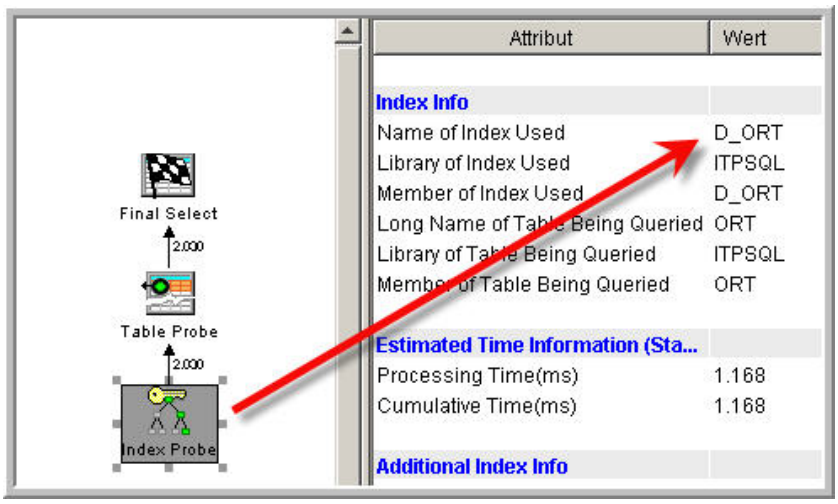
Weil logische DDS-Dateien im System i Navigator als SQL-View gezeigt werden, könnte man vermuten, dass der SQL-Optimizer bestehende Zugriffspfade logischer DDS-Dateien für SQL-Abfragen nicht nutzt. So ist es aber nicht. Sowohl in der Optimierungsphase zur Informationsbeschaffung als auch zur Ausführungszeit für den Datenzugriff können logische DDS-Dateien verwendet werden. Schauen wir uns auch hierzu ein einfaches Beispiel an. Zunächst lösche ich wieder alle SQL-Indizes:

```
DROP INDEX I_PLZ;
DROP INDEX I_PLZ_ONAME;
```

Anschließend führe ich das folgende SQL-Statement aus:

```
Select *
from ort
WHERE ONAME = ‘Bremen’;
```

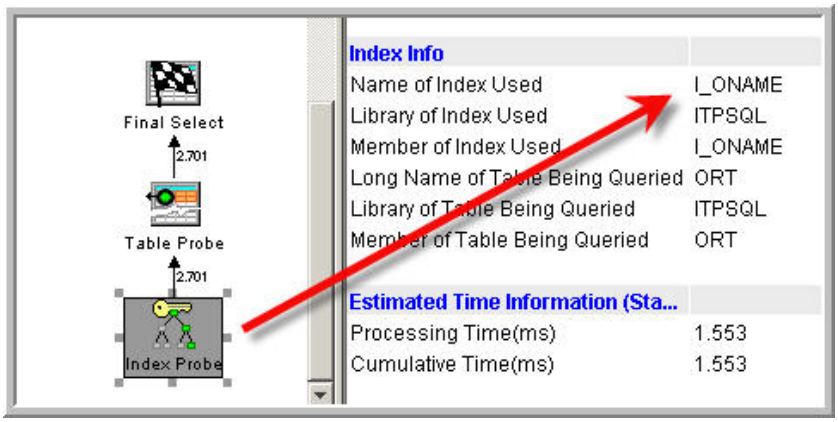
Den Ausführungsplan schaue ich mir mit Visual Explain an:



SQL-Abfrage mit logischer DDS-Datei

Der Optimizer nutzt für die Ausführung der Abfrage einen Zugriff über den Index. Dabei wird die logische DDS-Datei „D_ORT“ verwendet. Anhand der logischen DDS-Datei kann die Adresse der benötigten Sätze ermittelt werden. Der Datenabruf erfolgt anschließend wahlfrei. Schauen wir uns in einem weiteren Schritt an, was passiert, wenn ich zusätzlich einen SQL-Index erstelle:

```
CREATE INDEX I_ONAME ON ORT(ONAME);
```



Ausführung mit SQL-Index

7.3.5
Seite 12

Der Optimizer entscheidet sich tatsächlich für den SQL-Index und gegen unsere logische DDS-Datei. Der Optimizer bevorzugt SQL-Indizes! Grund hierfür ist die größere logische Seitengröße. Nur wenn Sie gezielt einzelne oder wenige Sätze benötigen, sind kleine logische Seiten vorteilhaft. Wenn aber Schlüsselwerte kombiniert, gruppiert und große Datenmengen abgerufen oder der Zugriffspfad seitenweise gelesen wird, dann sind größere logische Seiten vorteilhafter.

Was ist die Konsequenz? Mittelfristig sollten Sie versuchen, auf logische DDS-Dateien zu verzichten! Ersetzen Sie die DDS-Objekte – wie gezeigt – durch vergleichbare Indizes oder erstellen Sie die Objekte neu, nachdem die notwendigen SQL-Indizes aufgebaut wurden. Vielleicht lässt es sich nicht vermeiden, dass Sie Ihre HLL-Programme neu kompilieren müssen, aber wir werden nicht dauerhaft die alten und neuen Techniken parallel nutzen können. Wir müssen versuchen, alte Programme und neue Datenbanktechniken zu integrieren. Glauben Sie nicht, dass Sie alle SQL-Indizes zusätzlich erstellen können. Die Meinung – „Ein paar Zugriffspfade mehr oder weniger können doch nicht stören!“ – ist leider ein Irrtum! Das Problem der Zugriffspfadwartung wird zunächst häufig unterschätzt. Ich habe mit Anwendern gesprochen, die mir erzählten, dass

- das Einfügen eines Satzes mittlerweile länger als zwei Minuten dauert,
- die Zeit für die nächtlichen Datenübernahmen nicht mehr ausreicht,
- die CPU-Auslastung seit kurzem bei 100 Prozent liegt.

Auch hat sich mittlerweile gezeigt, dass eine neue Maschine mit noch mehr Arbeitsspeicher und höherem CPW-Wert nicht immer die bestehenden Probleme löst!

7.3.5.3 Der Encoded Vector Index (EVI)

Um die Verwirrung zu vervollständigen, hat IBM bereits 1998 mit Release V4R2 mit dem Encoded Vector Index (EVI) einen weiteren, effizienten Indextyp geschaffen, der aufgrund seiner geringen Größe und des einfachen Aufbaus gute Möglichkeiten bietet, effizient große Datenbanktabellen abzufragen. Für dieses Objekt wird kein binärer Suchbaum erstellt. Das Konzept der Bitmap-Indizes wird hier zugrunde gelegt. Es macht demzufolge Sinn, sich zunächst dieses Konzept anzuschauen.

Ausgangspunkt ist wieder die Tabelle „ITPSQL.ORT“:

RSN	ORTSNAME	PLZ
2	Berlin	48455
3	Bremen	55116
8	Düsseldorf	28357
10	Düsseldorf	40211
6	Leverkusen	07387
7	Leverkusen	51373
9	Leverkusen	28307
15	Maisach	82216

Ausgangssituation

Wenn Sie einen Bitmap-Index für die Spalte „Ortsname“ erstellen würden, dann sähe das Ergebnis für die dargestellte Ausgangssituation folgendermaßen aus:

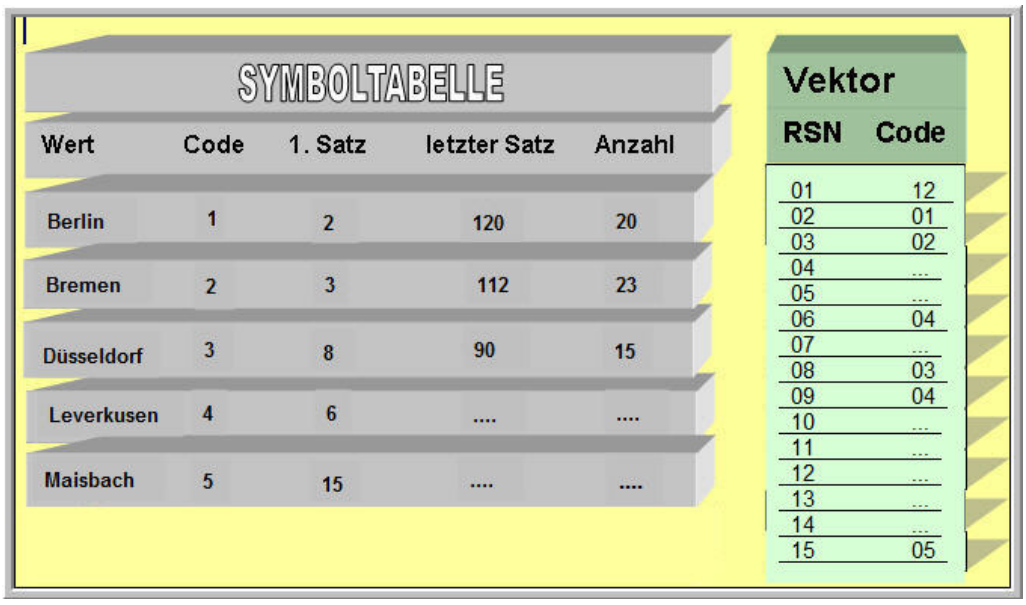
Bitmap	Wert
0000 0000 0000 0010	Berlin
0000 0000 0000 0100	Bremen
0000 0010 1000 0000	Düsseldorf
0000 0001 0110 0000	Leverkusen
0100 0000 0000 0000	Maisach

Die erste Zeile des Bitmaps repräsentiert die Vorkommen des Wertes „Berlin“. Das erste Bit ist „0“. Der Wert „0“ bedeutet in diesem Zusammenhang Wert „Berlin“ nicht in Zeile 1 vorhanden“. Das zweite Bit hat dagegen den Wert „1“. Der Satz mit der relativen Satznummer „2“ enthält den Wert „Berlin“. Wenn Sie in einer SQL-Abfrage alle Kunden aus Berlin benötigen, dann kennt die SQL Engine aufgrund des Bitmaps die relativen Satznummern des gesuchten Wertes. Außerdem lassen sich die Bitmaps durch logisches „AND“ oder „OR“ einfach kombinieren. Stellen Sie sich vor, Sie suchen Kunden aus „Düsseldorf“ oder aus „Leverkusen“:

7.3.5
Seite 14

Bitmap	Wert
0000 0010 1000 0000	Düsseldorf
0000 0001 0110 0000	Leverkusen
0000 0011 1110 0000	Ergebnis

Es werden die Sätze „6“, „7“, „8“, „9“ und „10“ benötigt. Das Problem mit den ursprünglichen Bitmap-Strukturen ist nur: Sie benötigen zu viel Platz. Nehmen wir an, die Tabelle verfügt über 200.000 Datensätze mit 50 unterschiedlichen Ortsangaben. Sie würden 200.000 * 50 also 1.000.000 Bits für den Bitmap-Index benötigen. Aus diesem Grund hat IBM die Bitmap-Technologie weiterentwickelt. Die patentierte Index-Technologie besteht aus einer Symboltabelle und einem Vector. Beide Komponenten werden gemeinsam als Datenstruktur gespeichert. In der Symboltabelle wird zunächst jedem eindeutigen Schlüsselwert ein binärer Code zugeordnet. So kann jeder Schlüsselwert – unabhängig von Typ und Länge – abgebildet werden. Zusätzlich werden in der Symboltabelle die erste und die letzte relative Satznummer sowie die Anzahl der Vorkommen (Kardinalität) hinterlegt. Im Vector ist wiederum jeder relativen Satznummer der zugehörige Binärcode entsprechend der Symboltabelle zugeordnet. Die nachfolgende Grafik verdeutlicht diese Strukturen:



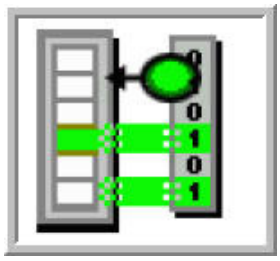
Encoded Vector Index

Die benötigte Byte-Anzahl zur Kodierung der vorhandenen Schlüsselwerte steht hierbei in direktem Bezug zur Anzahl der unterschiedlichen Vorkommen. Für die Codierung von bis zu 255 verschiedenen Vorkommen wird lediglich ein Byte benötigt. Wenn „256“ bis „65.535“ Werte kodiert werden müssen, dann sind zwei Byte notwendig. Falls die Spalte mehr als „65.536“ unterschiedliche Werte enthält, wird eine Vektorgröße von 4 Byte gewählt.

Von	Bis	Vektorgröße
1	255	1
256	65535	2
65536	4,2 Billionen	4

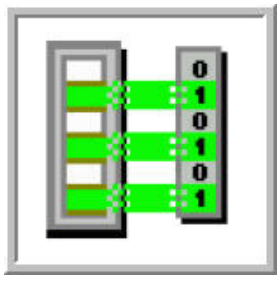
Der SQE Optimizer nutzt die Symboltabelle zur Informationsgewinnung während der Optimierungsphase. Für die Ausführung der Abfrage kann der EVI anschließend unterschiedlich eingesetzt werden:

EVI Index-Probe
EVIs können für den Tabellenzugriff nicht direkt genutzt werden. Vielmehr müssen – mit Hilfe des Vectors – zur Laufzeit temporäre Listen oder Bitmaps aufgebaut werden, die für den wahlfreien Tabellenzugriff genutzt werden. Der Vorteil liegt – im Gegensatz zum Einsatz eines Binärindizes – im asynchronen Datenzugriff. I/O-Operationen können auf diese Weise sehr viel effektiver ausgeführt werden, da große Teile der Tabelle übersprungen werden, wenn Sätze nicht den Suchkriterien entsprechen. Auch für JOIN-Abfragen können EVIs mittlerweile in Kombination mit Hash-Tabellen eingesetzt werden.



Visual Explain Icon Encoded Vector-Probe

EVI Table Scan
Wenn sich der Optimizer für die Methode „EVI Table Scan“ entscheidet, dann werden nur die Einträge der Symboltabelle sequentiell gelesen, ohne den Vector oder die zugrunde liegende Tabelle zu nutzen. Dieses Verfahren kann Abfragen, die die Schlüsselwörter „GROUP“, „DISTINCT“, „COUNT“ oder „COUNT DISTINCT“ enthalten, extrem beschleunigen. Entsprechende Beispiele enthält das Kapitel 4.



Visual Explain Icon EVI Symbol Table Scan

Ergänzung 8/2009

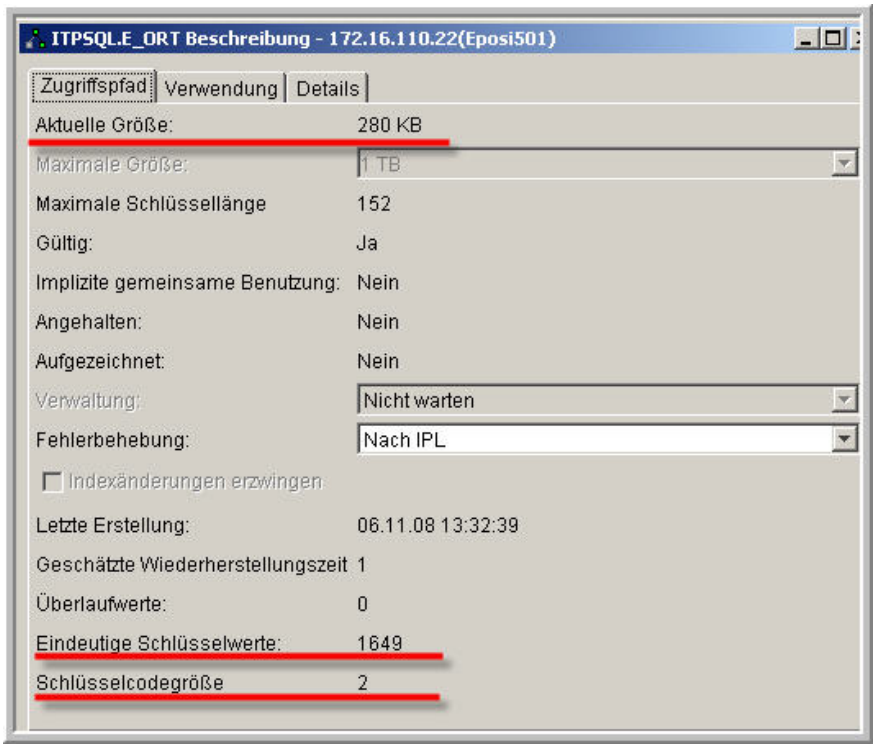
7.3.5.4 EVI-Index – Erstellung und Pflege

Die Klausel „ENCODED VECTOR“ im Statement „CREATE INDEX“ ist ausreichend, um den Typ des Index zu ändern.

```
CREATE ENCODED VECTOR INDEX E_ORT
ON ORT(ORTSNAME) WITH 3000 DISTINCT VALUES;
```

Der Zusatz „WITH 3000 DISTINCT VALUES“ bestimmt die Größe des Vectors. In meinem Beispiel sollten zwei Byte für die Kodierung der unterschiedlichen Schlüsselwerte ausreichend sein. Sie kennen bereits die Tabelle „ORT“ aus den vorangehenden Beispielen und wissen daher, dass die Spalte „ORTSNAME“ die Ortsbezeichnung enthält. Aktuell gibt es 1649 verschiedene Ortsnamen. Es ist nicht anzunehmen, dass wir irgendwann einmal mehr als 65535 unterschiedliche Werte abbilden müssen, so dass zwei Byte für die Kodierung der Schlüsselwerte ausreichend sein dürften.

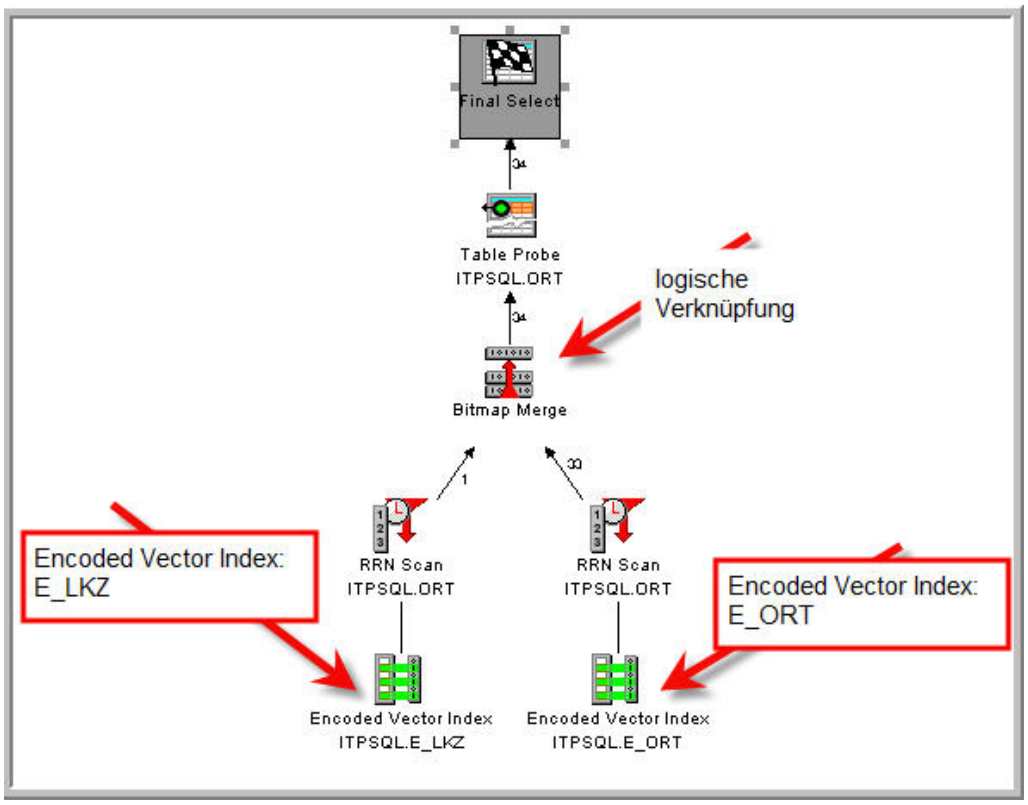
Schauen wir uns in einem weiteren Schritt die Eigenschaften der EVIs an. Hierzu öffnen Sie im System i Navigator das Schema und anschließend den Eintrag „Indizes“. Auf der rechten Bildschirmseite finden Sie alle SQL-Indizes. Ich markiere jetzt mit der rechten Maus den erstellten EVI und öffne im Kontextmenü den Dialog „Beschreibung“.



Encoded-Vector-Index-Eigenschaften

Die Schlüssel-Codegröße beträgt aufgrund der Klausel „WITH n DISTINCT VALUES“ wie erwartet 2 Byte. In meinem Beispiel hätte ich auf die Angabe der „WITH“-Klausel sogar verzichten können, da die Tabelle „ORT“ bereits alle Daten enthält und die Vectorgröße während der Indexerstellung automatisch anhand der Datenlage ermittelt werden kann. Ein Vorteil der EVIs ist die geringe Größe. Der eben erstellte Index ist nur 280Kb groß. Außerdem liefern EVIs aufgrund der Symboltabelle exaktere Statistikdaten als die Binärindizes; Sie können zudem logisch über „AND“ bzw. „OR“ verknüpft werden. Und das probieren wir jetzt aus. Ich habe zwischenzeitlich einen weiteren EVI für die Spalte „Länderkennzeichen“ der Tabelle „Ort“ erstellt und führe folgende Abfrage aus:

```
SELECT *
FROM ITPSQL
WHERE Ortsname = ,Bremen' OR LKZ = 10;
```



EVI-logische Verknüpfung

Der Ausführungsplan zeigt, dass die SQE beide EVI-Objekte für die Ausführung nutzt, indem zunächst für jedes lokale Kriterium ein temporäres Bitmap aufgebaut wird. Die Teilergebnisse werden zu einem Bitmap vereint. Über das neue temporäre Objekt erfolgt der wahlfreie Zugriff auf die Tabelle „ORT“. Aufgrund dieser Eigenschaft sollten Sie Encoded Vector Indizes vorrangig für einzelne oder zusammengesetzte Spalten erstellen, die häufig in unterschied-

lichen Kombinationen in den lokalen Selektionskriterien verwendet werden. Achten Sie bei der Auswahl der Spalten zusätzlich darauf, dass es sich um Spalten mit einer relativ geringen und konstanten Kardinalität handelt. Der Optimizer könnte natürlich auch einen Binärindex für die Ausführung nutzen. Der Ausführungsplan ist letztendlich von der selektierten Datenmenge abhängig. Werden wenig Datensätze (ca. bis zu 20 Prozent) benötigt, dann ist der Binärindex effektiver als der Einsatz der EVIs. Sobald aber größere Datenmengen (ca. 20 bis 70 Prozent) abgefragt werden, wird der Optimizer möglicherweise die EVIs verwenden, weil durch die asynchronen Lesevorgänge die I/O-Kosten in der Regel deutlich reduziert werden. Die Reduzierung der Plattenzugriffe rechtfertigt in diesen Fällen die zusätzlichen Kosten für den Aufbau temporärer Objekte.

Encoded Vector Indizes haben aber nicht nur Vorteile. Insbesondere in der Pflege gibt es einige beachtenswerte Eigenheiten. Wenn Sie häufig Inhalte der Schlüsselspalte verändern, neue Zeilen mit neuen Schlüsselwerten hinzufügen oder bestehende Sätze löschen, dann könnte es sein, dass Sie die Symboltabelle gegebenenfalls von Zeit zu Zeit neu dimensionieren müssen, damit Schlüsselwerte nicht im Overflow-Bereich des Index abgelegt werden. Der Overflow-Bereich wird genutzt, wenn die gewählte Schlüssel-Codegröße für die Darstellung der Schlüsselwerte nicht ausreichend ist. Deshalb sollten Sie den Zusatz „WITH n DISTINCT VALUES“ keinesfalls vergessen, wenn Ihre Tabelle zum Zeitpunkt der Indexerstellung leer oder nicht vollständig gefüllt ist, um eine entsprechende Anzahl Bytes für die Schlüsselkodierung zu reservieren. Falls Sie sich nicht sicher sind, wie hoch die Anzahl der eindeutigen Werte sein wird, verwenden Sie einfach den Maximalwert. Bei den Spalten „Ortsname“ und „Länderkennzeichen“ habe ich dieses Problem nicht. Wenn ich aber für eine Spalte „Kundennummer“ einen EVI erstellen müsste, könnte ich heute noch nicht sagen, wie sich der Inhalt dieser Spalte entwickelt wird. Deshalb würde ich den Index folgendermaßen erstellen:

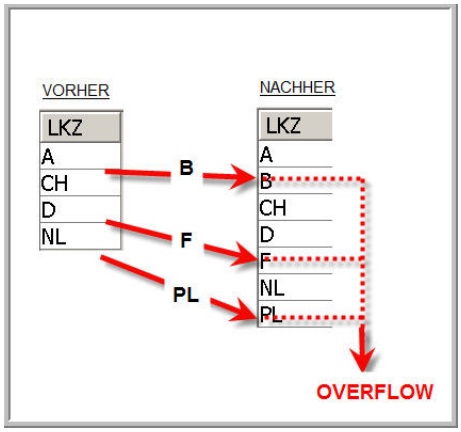
```
CREATE ENCODED VECTOR INDEX E_KDNR
ON KDTAB(KDNR) WITH 100000 DISTINCT VALUES;
```

Aber der Overflow-Bereich wird auch eingesetzt, wenn nachträglich in unsortierter Reihenfolge Schlüsselwerte eingefügt werden. Stellen wir uns folgendes Szenario vor:

Die Spalte „LKZ“ enthielt zum Zeitpunkt der Indexerstellung nur die Werte „A“, „CH“, „D“ und „NL“. In aufsteigender Sortierfolge wurde dem Wert „A“ der Binärcode „1“, dem Wert „CH“ der Binärcode „2“ usw. zugeordnet. Schauen Sie, was passiert, wenn ich die folgenden Sätze in die Tabelle „ORT“ einfüge:

```
INSERT INTO ITPSQL.ORT(LKZ, ONAME, PLZ, ORTINSERTDATE,
ORTINSERTUSER)
VALUES('F', 'St. Germain', '78100', NOW(), 'RASCHE' );

INSERT INTO ITPSQL.ORT(LKZ, ONAME, PLZ, ORTINSERTDATE,
ORTINSERTUSER)
VALUES('B', 'Brüssel', '1020', NOW(), 'RASCHE' );

INSERT INTO ITPSQL.ORT(LKZ, ONAME, PLZ, ORTINSERTDATE,
ORTINSERTUSER)
VALUES('PL', 'Warszawa', '00140', NOW(), 'RASCHE' );
```

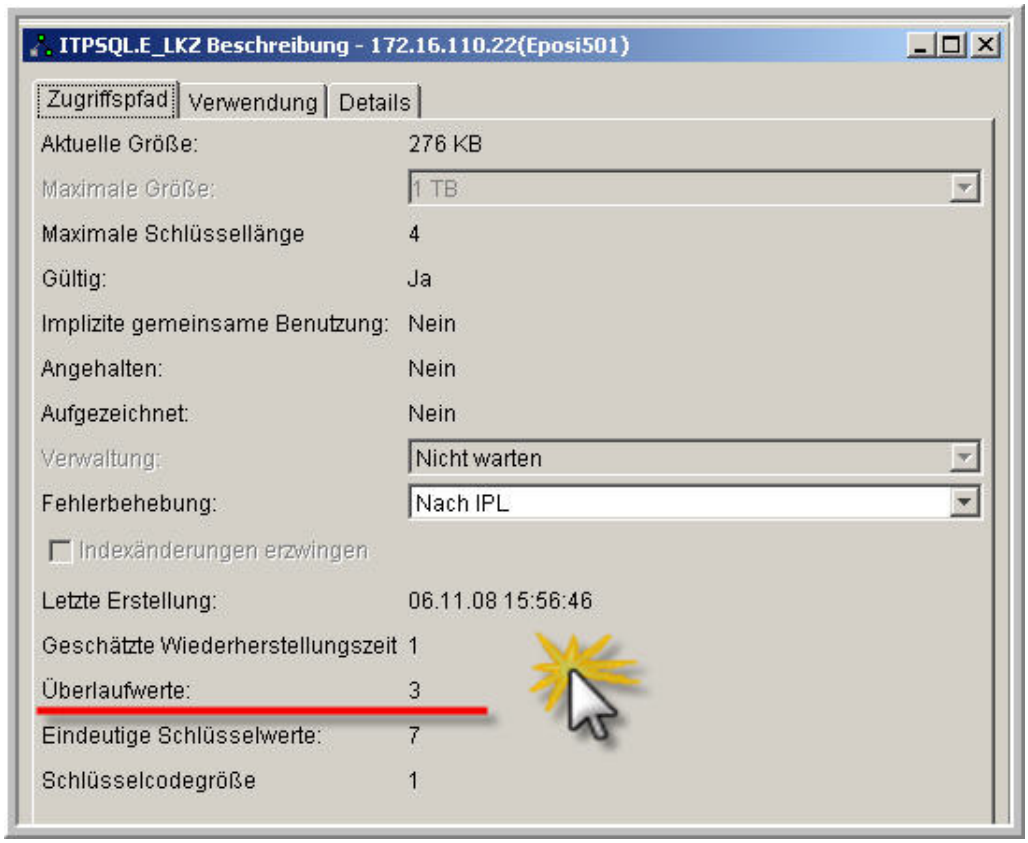
Die neuen Schlüsselwerte müssten innerhalb der Symboltabelle eingefügt werden. Dazu wären aber ein Neuaufbau der Symboltabelle, eine neue Binärkodierung und der anschließende Neuaufbau des Vectors erforderlich. Stattdessen werden diese Werte in den Overflow-Bereich verschoben. Die folgende Grafik veranschaulicht dieses Verhalten:



Overflow – EVI

Je mehr Werte aber in den Overflow-Bereich verlagert werden, je ineffektiver ist der Einsatz dieses Objektes. Ob im Overflow-Bereich des Indexes derzeit tatsächlich drei Einträge vorhanden sind, lässt sich schnell feststellen. Klicken Sie im System i Navigator mit der rechten Maus auf den entsprechenden Index und öffnen Sie im Kontextmenü den Eintrag „Beschreibung“.

7.3.5
Seite 20



EVI Eigenschaften – Overflow

Wie erwartet enthält der Overflow-Bereich Einträge. Die Bereinigung erfolgt, wenn Sie den Index neu erstellen. Sie können auch den Parameter „Force Rebuild Access Path“ (FRCRBDAP) des Befehls „Change Logical File“ (CHGLF) auf den Wert „*YES“ setzen. Diese Alternative sollten Sie in Betracht ziehen, wenn Möglichkeiten und Informationen zur Neuerstellung des Indexes fehlen.

Abschließend eine kurze Zusammenfassung dieser Eigenheiten:

	Situation	Wirkung
Sehr Effektiv – ohne Zusatz pflege	1. Schlüsselwert existiert bereits.	Statistiken der Symboltabelle werden modifiziert. Vector erhält neues Element mit neuer RSN und existierenden Schlüsselcode.
	2. Schlüsselwert ist neu. Vectorgröße ist für die Kodierung ausreichend. Wert kann angehangen werden.	Symboltabelle erhält neue Zeile mit Schlüssel-Codeeintrag und Statistiken. Vector erhält neues Element mit neuer RSN und neuem Schlüsselcode.
Mit Zusatz pflege	3. Schlüsselwert ist neu. Vectorgröße ist für die Kodierung ausreichend. Außerhalb bestehender Sortierfolge.	Symboltabelle erhält neue Zeile für Schlüssel-Codeeintrag und die zugehörigen Statistiken im Overflow-Bereich. Vector erhält neues Element mit neuer RSN und neuem Schlüsselcode. Überlegung: Neuerstellung des Index, damit die Overflow-Werte integriert werden!
wenig effektiv	4. Schlüsselwert ist neu. Vectorgröße ist nicht für die Kodierung ausreichend.	Überlegung: Neuerstellung des Index, damit die neuen Werte integriert werden!

Jetzt erklärt sich auch, warum Sie Encoded Vector Indizes vorwiegend für

- Read-Only-Tabellen oder
- Spalten mit einem relativ statischen Wertebereich

definieren sollten. Andernfalls müssen Sie überlegen, ob es möglich ist, den Index vor der Datenübernahme zu löschen und ihn anschließend neu zu erstellen.

Vielleicht fragen Sie sich jetzt, warum Sie einen Encoded Vektor Index verwenden sollten? Der EVI ist lediglich auf das Selektieren und nicht auf das Sortieren der Datensätze ausgerichtet. Alles, was der Encoded Vector Index kann, scheint auch mit dem Binärindex möglich zu sein.

Ergänzung 8/2009

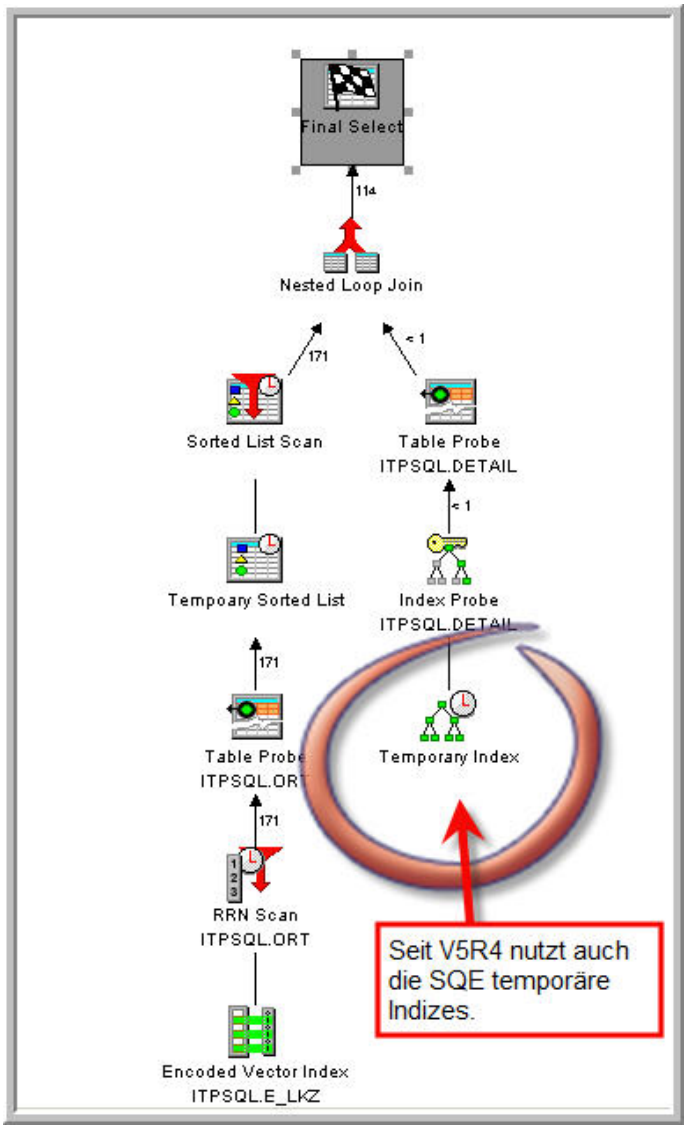
Die Erstellung eines Encoded Vector Index sollten Sie vor allem in Betracht ziehen, wenn mehr als 20 Prozent der Daten im Arbeitsspeicher benötigt werden und wenn ein vorhandener Binärindex hierfür nicht genutzt wird. EVIs sind kleiner als binäre Bäume. Oft brauchen sie nur ein Sechszehntel des Platzes, den binäre Bäume benötigen. Festplattenspeicherung und – was noch wichtiger ist – die Möglichkeit, den Index umfangreicher Tabellen mit wenigen Speicherseiten zu laden, kann die Performance steigern. Außerdem kann der Datenzugriff anders als beim Binärindex asynchron erfolgen. Ein weiterer Vorteil ist, dass mehrere EVIs logisch kombiniert werden können. Erstellen Sie für Spalten, die häufig unterschiedlich in den Abfragen kombiniert werden, EVIs. Der Optimizer kann dann verschiedene Kombinationen lokaler Selektionskriterien zur Laufzeit selbst aufbauen. Als „Informant" im Optimierungsprozess kann der Binärindex bestenfalls nur grobe Einschätzungen dafür liefern, wie viele Einträge die speziellen Selektionskriterien erfüllen. EVIs geben Aufschluss über die genaue Zahl der Einträge anhand der Symboltabelle. EVIs können die Performance einfach dadurch verbessern, dass sie detaillierte Informationen während des Optimierungsprozesses zur Verfügung stellen, wenn die Statistikdaten nicht ausreichend sind, weil sich die Datenlage schnell verändert. Wie so oft geht es also nicht darum, zu entscheiden: Encoded Vector Index oder Binärindex. Vielmehr stellen Sie mit dem Encoded Vector Index Werkzeuge bereit, die dem Optimizer zusätzliche Möglichkeiten bieten, die Abfrage effizient auszuführen.

7.3.5.5 Maintained Temporary Index (MTI)

Vor Release V5R4 wurden temporäre Indizes ausschließlich von der CQE erstellt. Die SQE nutzte bis Release V5R4 keine temporären Indizes; sie bevorzugte die Methode „Table Scan". Erst mit Release V5R4 wurden temporäre Indizes als neues Feature eingeführt. Der SQE Optimizer beobachtet – vor der Erstellung des temporären Indexes – die Ausführung mehrerer SQL-Abfragen und berücksichtigt die im Index Advisor hinterlegten Informationen, um zu entscheiden, ob die Erstellung eines temporären Indexes sinnvoll ist. Durch das Konzept der temporären Indizes kann sich die SQE ein Stück weit selbst tunen. Schauen wir uns die Verwendung an einem einfachen Beispiel an:

```
select lkz, bzkurz, bzlang, ortsname
from itpsql.detail join itpsql.ort on lkz = dekurz
WHERE LKZ IN (10, 20, 30)
order by ortsname;
```

Es existiert für die Spalte „DEKURZ" der Tabelle „DETAIL" kein permanenter Index, der verwendet werden könnte. Daher entscheidet sich der Optimizer einen temporären Index zu erstellen.



Erstellung eines temporären Index

Für die Erstellung und Nutzung des temporären Index benötigt die SQE zusätzliche Verarbeitungszeit. Dieser Zeitfaktor ist jedoch geringer als jede alternative Zugriffsmethode.

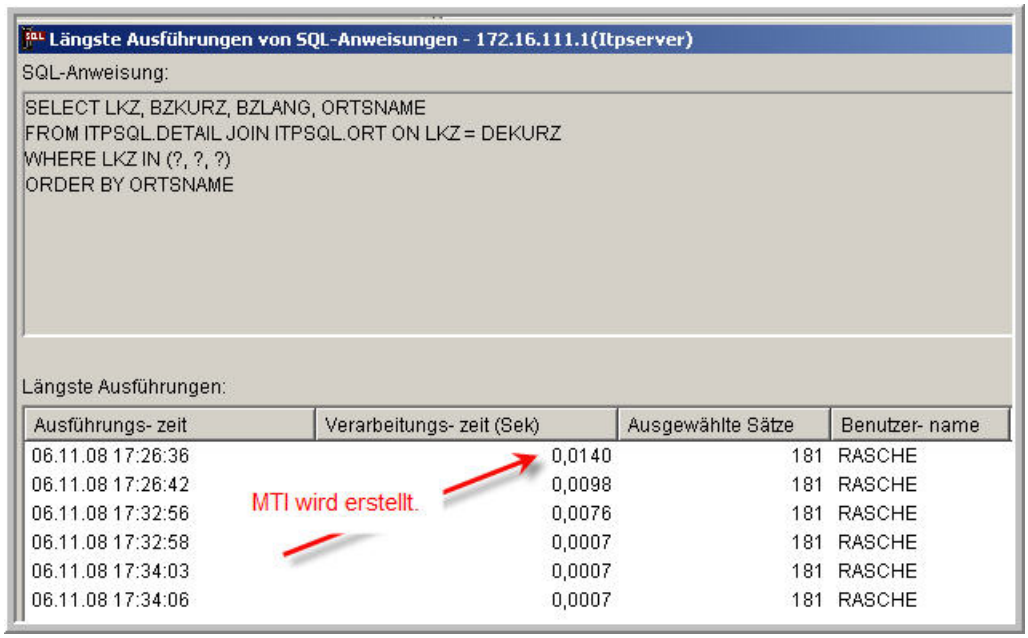
Temporäre Indizes nutzt die CQE innerhalb eines Jobs für eine einzige SQL-Abfrage. Die Mehrfachverwendung eines solchen Index – auch innerhalb des gleichen Jobs – ist ausgeschlossen.

Das Konzept der temporären SQE-Indizes wurde allerdings vollständig überarbeitet: Erstellung und Wiederverwendbarkeit von temporären Indizes wurde optimiert, indem die Informationen über die temporären Objekte im Plan Cache hinterlegt werden. Der Plan Cache enthält also nicht nur Ausführungspläne, sondern auch temporäre SQE-Indizes. Dadurch ist es möglich, einen temporären Index einmalig zu erstellen und anschließend beliebig vielen SQL-

Ergänzung 8/2009

Abfragen in verschiedenen Jobs zur Verfügung zu stellen. Die temporären SQE-Indizes bleiben solange im Plan Cache existent, bis alle Zugriffspläne aus dem Plan Cache entfernt sind, die das temporäre Objekt nutzen. Die neuen temporären Indizes, die ausschließlich von der SQE erstellt werden, nennt man übrigens „Autonomic Index“ (AI) oder „Maintained Temporary Index“ (MIT).

So erklären sich auch die unterschiedlichen Verarbeitungszeiten meiner SQL-Abfrage, die ich mir im Plan Cache anzeigen lasse. Ich öffne hierfür die Datenbank ➜ Rechtsklick auf „Momentaufnahmen für SQL Planungscache“ ➜ „SQL Planungscache“ ➜ „Anweisungen anzeigen…“. Daraufhin öffnet sich das Fenster „Anweisungen des SQL-Planungscache“. Jetzt selektiere ich die Ausführungspläne und wähle über das Kontextmenü des entsprechenden Ausführungsplanes den Eintrag „Längste Ausführungen“:

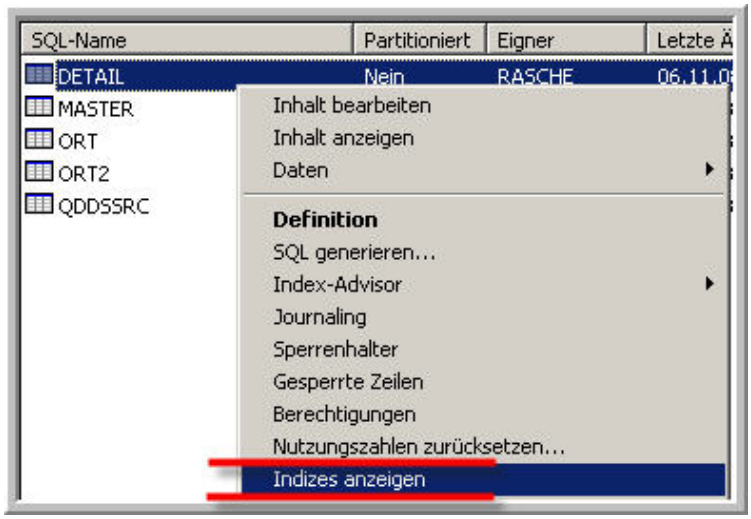


Unterschiedliche Ausführungszeiten mit MIT

In der Abbildung ist deutlich zu erkennen, dass die erste Verarbeitungszeit um ein Vielfaches höher ist als die nachfolgenden. Den Grund kennen wir jetzt; der temporäre Index musste erstellt werden. Da der Plan Cache aber beim IPL geleert wird, sind diese quasi-permanenten Indizes nach einem IPL nicht mehr existent und müssen bei der nächsten Ausführung einer SQL-Abfrage, die einen solchen Index erfordert, neu erstellt werden. Es kann daher sinnvoll sein, die temporären Objekte permanent zu erstellen.

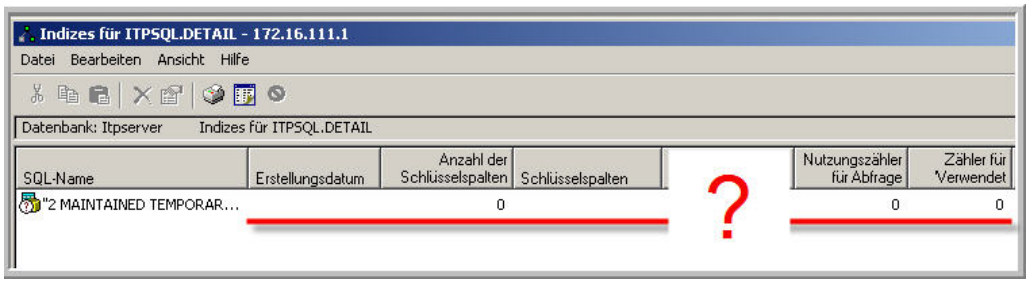
Doch zuvor stellt sich die Frage „Wie finden wir die temporär erstellten Indizes?“ System i Navigator bietet gleich mehrere Möglichkeiten, diese Objekte aufzuspüren:

Die Funktion „Indizes anzeigen für…“
Sie können sich mit System i Navigator die Indizes anzeigen lassen, die für eine Tabelle erstellt wurden. Ich öffne hierfür zunächst die Datenbank und das Schema, das die Tabelle „DETAIL“ enthält. Anschließend wähle ich – wie in der folgenden Abbildung dargestellt – im Kontextmenü der Tabelle den Eintrag „Indizes anzeigen“.



System i Navigator – Indizes anzeigen

Für die folgende Bildschirmausgabe verändere ich zunächst die Reihenfolge der Spalten, weil die benötigten Informationen ansonsten sehr weit rechts stehen. Hierfür können Sie das Kontextmenü oder die Funktionstaste „F12“ nutzen.



Indizes anzeigen – MTI

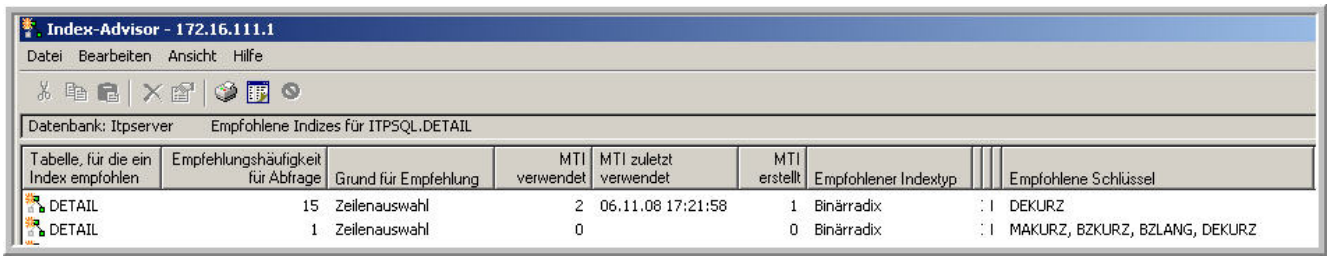
Tatsächlich wird der temporäre Index für die Tabelle „DETAIL“ angezeigt. Allerdings sind die Informationen eher dürftig. Während für permanente Indizes Nutzungszähler existieren und die Schlüsselspalten der Indizes angezeigt werden, erhalten Sie in diesem Fall keine Informationen. Man kann nicht einmal erkennen, wann der Index erstellt und wie häufig er genutzt wurde. Für mich ist diese Anzeige enttäuschend. Vielleicht ändert sich an dieser Darstellung ja noch etwas im Laufe der nächsten Zeit.

7.3.5
Seite 26

Index-Advisor

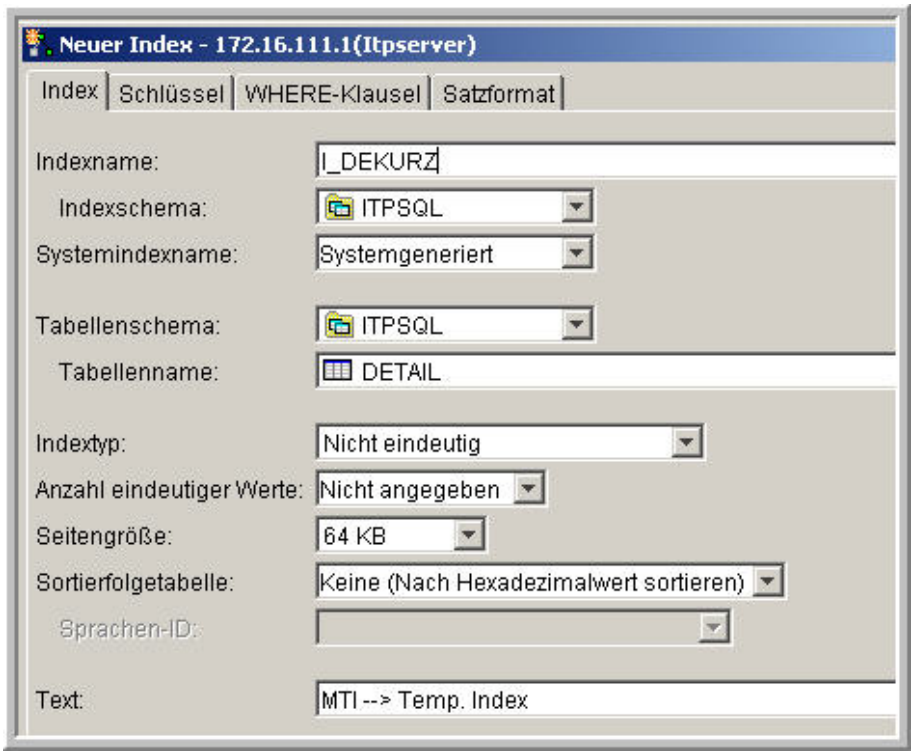
Etwas besser sind die Informationen im Index Advisor. Hierfür wählen Sie im Kontextmenü der Tabelle den Eintrag „Index Advisor“.

Ich werde zu einem späteren Zeitpunkt auf den Index Advisor zurück kommen. Jetzt beschränke ich mich lediglich auf die Darstellung der MTI-Informationen. Wir suchen alle Indexvorschläge mit entsprechenden Einträgen in den Spalten „MTI erstellt“ und „MTI verwendet“. Die benötigten Informationen stehen weit rechts, wenn Sie nicht die Spaltenreihenfolge der Bildschirmausgabe ändern – wie ich es getan habe:



Index Advisor – MIT

Die Informationen sind deutlich besser als in der vorangegangenen Bildschirmausgabe. Die Empfehlung für den Binärindex mit dem Schlüsselfeld „DEKURZ“ erfolgte insgesamt fünfzehn Mal aufgrund lokaler Selektionskriterien. Ein Mal wurde der temporäre Index mit dieser Spalte bisher aufgebaut und insgesamt zweimal genutzt. Zuletzt wurde der MTI am „06.11.2008“ verwendet. Wir hätten jetzt – aufgrund dieser Informationen – die Möglichkeit zu entscheiden, den Index permanent zu erstellen. Sie müssen hierfür den Dialog nicht verlassen sondern könnten im Kontextmenü des Indexvorschlages den Eintrag „Index erstellen“ wählen. Daraufhin erscheint ein neuer Dialog:



Index Advisor – MTI erstellen

Alle benötigten Parameter sind voreingestellt – entsprechend der Indexempfehlung. Es fehlt nur noch ein Indexname und eventuell ein beschreibender Text. Im Release V6R1 wurden die Register „WHERE-Klausel“ und „Satzformat“ hinzugefügt. Eine Erläuterung dieser Register finden Sie im nächsten Abschnitt dieses Kapitels. Auf diese Weise lassen sich sehr komfortabel Indizes erstellen. Die Anzeige verleitet dazu, wahllos Indizes aufgrund der Indexempfehlungen zu erstellen. Sie sollten stets sorgfältig prüfen, ob der Index wirklich ein guter Indexkandidat ist oder ob es nicht wesentlich bessere Spaltenkombinationen gibt. Denken Sie daran, Indexempfehlungen nach der Bearbeitung aus der Liste zu entfernen, damit nicht zu einem späteren Zeitpunkt der Eindruck entsteht, es sei ein weiterer temporärer Index erstellt worden.

Die Nutzung des Index Advisor ist für die Untersuchung der MTIs besser geeignet als die zuvor vorgestellte Möglichkeit. Ein Nachteil bleibt allerdings bestehen: Sie müssen wissen, für welche Tabelle temporäre Indizes erstellt wurden. Natürlich könnten Sie den systemweiten Index Advisor nutzen und die MTIs darüber ermitteln. Allerdings stehen dort nicht selten mehr als 10.000 Indexempfehlungen – alte und aktuelle. Hinzu kommt, dass ich die Handhabung der grafischen Darstellung sehr umständlich finde. Mir sind daher kompaktere Informationen lieber.

Ergänzung 8/2009

7.3.5
Seite 28

Die Tabelle: System Wide Index Advised (SYSIXADV)

Alle Informationen der Indexratgeber werden in der Systemtabelle QSYS2. SYSIXADV hinterlegt. Sie können mit Hilfe von SQL auf die Informationen zugreifen und dabei beliebige Filterkriterien verwenden. Diese Metatabelle benutze ich jetzt, um mir die notwendigen Informationen über die MTIs zu beschaffen:

```
SELECT TABLE_NAME as TAble, KEY_COLUMNS_ADVISED AS KEYS,
char(date( LAST_ADVISED) , EUR), TIMES_ADVISED as Adviced,
MTI_USED AS USED, MTI_CREATED AS CREATED,
char(date( LAST_ADVISED) , EUR) AS „LAST USED“

FROM QSYS2.SYSIXADV

WHERE TABLE_SCHEMA = ‘ITPSQL’ AND MTI_USED > 0
ORDER BY MTI_USED DESC;
```

Im Result Set erscheinen anschließend – kompakt dargestellt – die gewünschten Informationen:

```
SELECT TABLE_NAME as TAble, KEY_COLUMNS_ADVISED AS KEYS,
char(date( LAST_ADVISED) , EUR), TIMES_ADVISED as Adviced,
MTI_USED AS USED, MTI_CREATED AS CREATED,
char(date( LAST_ADVISED) , EUR) AS "LAST USED"

FROM QSYS2.SYSIXADV

WHERE TABLE_SCHEMA = 'ITPSQL' AND MTI_USED > 0
ORDER BY MTI_USED DESC;
```

TABLE	KEYS	00003	ADVICED	USED	CREATED	LAST USED
DETAIL	DEKURZ	06.11.2008	15	2	1	06.11.2008
ORT	LKZ	06.11.2008	12	2	1	06.11.2008

Index Advisor – SYSIXADV

Haben Sie Bedenken, dass die temporären Indizes Ihr System unnötig belasten? Ihre Sorge ist grundlos, da der SQE Optimizer – anders als die CQE – nicht für einfache lokale Selektionskriterien temporäre Indizes erstellt. Außerdem werden mögliche Ausführungsalternativen sorgfältig geprüft. Wenn Sie dennoch die Erstellung temporärer Indizes einschränken wollen, können Sie das Attribut „ALLOW_TEMPORARY_INDEXES“ in der Attributdatei QAQQINI ändern. Mit dieser Option legen Sie fest, inwieweit temporäre Indizes vom Optimierungsprogramm für die Ausführung der Abfrage berücksichtigt werden. Die Einstellung „*YES“ wird als Unterlassungswert verwendet; sie erlaubt dem Optimizer temporäre Indizes einzusetzen. Wenn Sie den Wert „*ONLY_REQUIRED“ angegeben haben, werden temporäre Indizes für den Zugriffsplan nicht berücksichtigt. Der Optimizer wählt eine andere Implementierung unabhängig von den Kosten, um keinen temporären Index zu erstellen. Nur

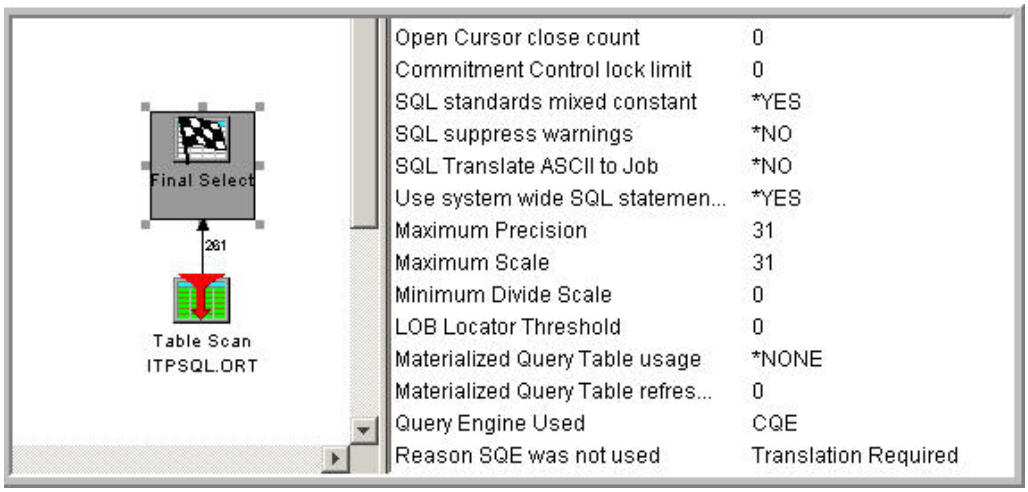
wenn kein funktionsfähiger Plan gefunden wird, wird ein temporärer Index benutzt. Wenn Sie hingegen verhindern wollen, dass die MTIs im Plan Cache gespeichert werden, weil Sie den Cache-Bereich für die Ausführungspläne benötigen, bewirkt die Änderung der Option „CACHE_RESULTS“ in der Attributdatei „QAQQINI“, dass die MTIs gelöscht werden, wenn der Datenpfad der Abfrage geschlossen wird. Dies entspricht dem Verhalten der CQE.

7.3.5.6 Abgeleitete Indizes Release V6R1

Traditionell konnten in SQL-Indizes nur einzelne Spalten oder Spaltenkombinationen verwendet werden. Seit Release V6R1 ist es auch möglich, Ausdrücke oder SQL-Funktionen zu verwenden. Schauen Sie sich das folgende SQL-Statement an:

```
SELECT *
FROM ITPSQL.ORT
WHERE UPPER(ORTSNAME) = ,BRÜSSEL‘;
```

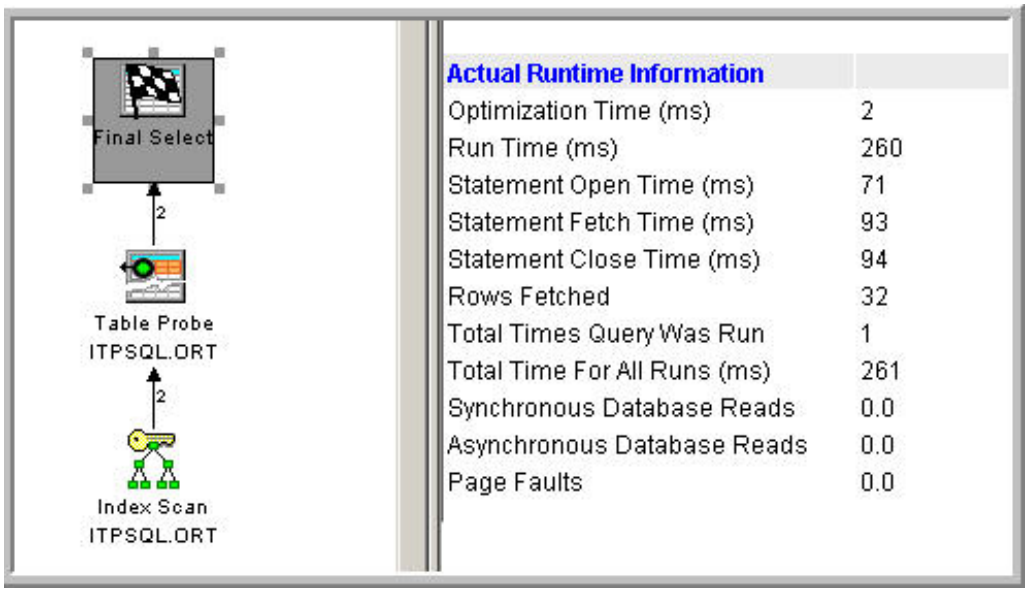
Die Funktionen „UPPER“, „LOWER“ und „TRANSLATE“ werden im Release V5R4 von der CQE ausgeführt. Als Grund finden Sie stets den Hinweis „Translation required“. Doch obwohl ein Binärindex für die Spalte „Ortsname“ existiert, kann die CQE den Index für den Datenzugriff nicht nutzen.



NLSS-Konvertierung V5R4

7.3.5
Seite 30

Seit Release V6R1 kann die SQE solche Abfragen übernehmen und nutzt hierfür auch die bestehenden Binärindizes – wie folgende Abbildung beweist:



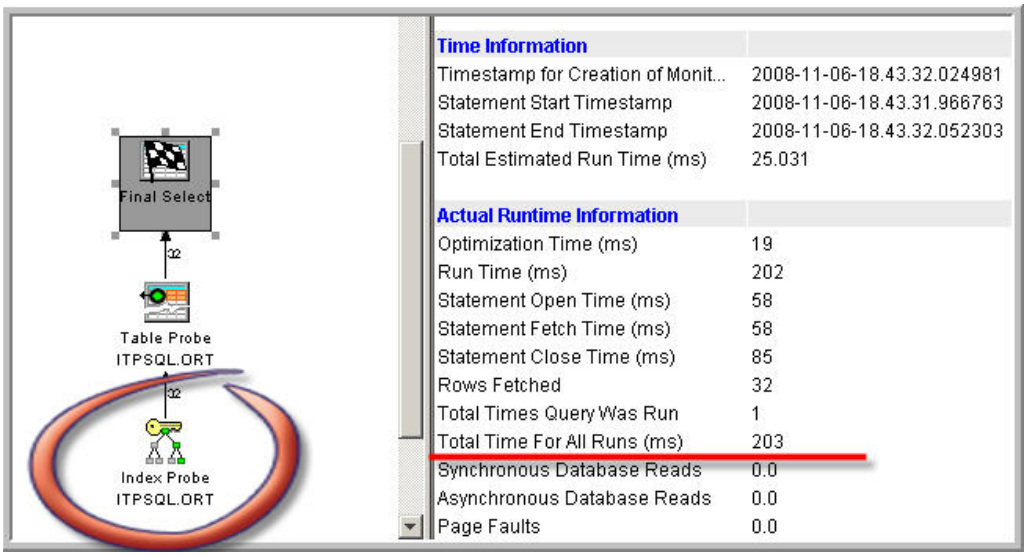
NLSS-Konvertierung V6R1

Allerdings muss der Index noch seitenweise gelesen werden, da aufgrund der Funktion „UPPER“ ein direkter Vergleich der Schlüsselwerte nicht möglich ist. Der Vorteil der Methode „Table Scan“ liegt darin, dass über den Index sehr viel mehr Sätze mit einer einzigen I/O-Operation gelesen werden können. Die gelesene Seite enthält lediglich die relative Satznummer und den Schlüsselwert, während die I/O-Seite beim „Table Scan“ alle Spalten der Tabelle beinhaltet. So erklärt sich die Zeitersparnis, wenn die SQE die Abfrage ausführt.

Seit Release V6R1 geht es aber noch besser. Erstellen Sie einfach einen abgeleiteten Index, damit der direkte Zugriff über Schlüsselwerte möglich wird:

```
CREATE INDEX I_Upper_Ortsname
ON itpsql.ort(UPPER(ortsname))
```

Die Syntax ist einfach. Sie können einfach Funktionen oder Ausdrücke in der Spaltenbezeichnung verwenden. Nach der Erstellung des abgeleiteten Index ändert sich der Ausführungsplan erneut.



Einsatz abgeleiteter Indizes V6R1

Die SQE wählt jetzt die Methode „Table Probe“. Während des Zugriffs auf die Daten können die Schlüsselwerte bereits geprüft werden, so dass nur noch die tatsächlich benötigten Sätze gelesen werden. Die Datenmenge und die benötigte CPU-Zeit reduzieren sich erneut. Natürlich können Sie auch andere skalare SQL-Funktionen nutzen:

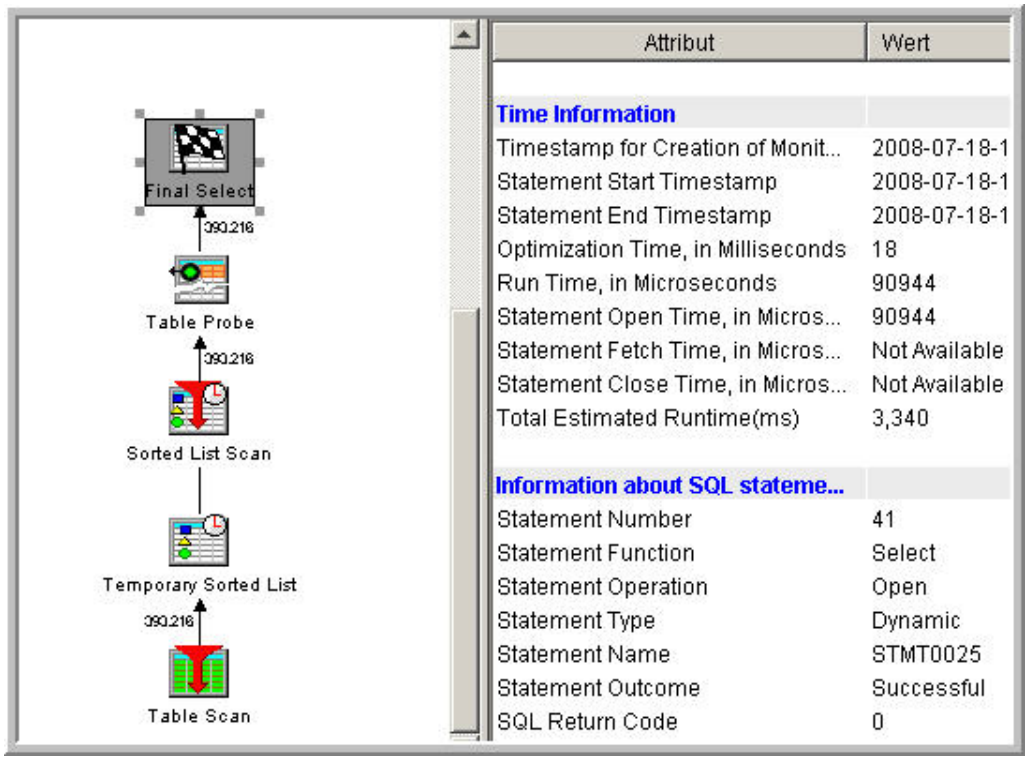
Eine ähnlich gute Wirkung lässt sich mit diesen Indizes erzielen, wenn Sie mathematische Ausdrücke z. B. in der „ORDER-BY“-Klausel nutzen.

Im nächsten Beispiel benötige ich eine Liste, die nach Bestandswerten (PRDQTY) sortiert ist. Hierbei soll aber auch die aktuelle Bestellmenge (PRDBSTQTY) berücksichtigt werden.

```
SELECT *
FROM PRDKT00
ORDER BY (PRDQTY+ PRDBSTQTY);
```

7.3.5
Seite 32

Die folgende Abbildung zeigt, dass die Daten zunächst vollständig gelesen werden. Erst im Arbeitsspeicher wird der Ausdruck aufgelöst, um die Sortierfolge zu bestimmen.

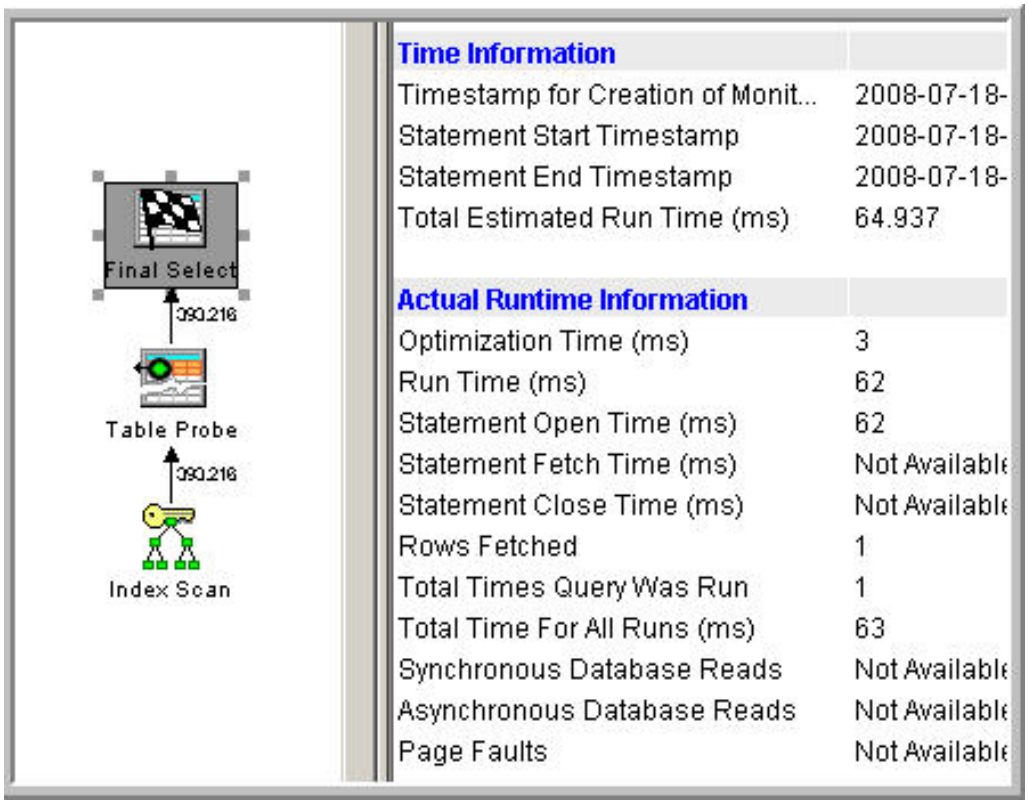


Mathematische Ausdrücke „ORDER BY“ V5R4

Angesichts der hohen Laufzeiten kann man schnell auf die Idee kommen, Summenspalten in den Tabellen zu nutzen, um derartige Ausführungspläne zu umgehen. V6R1 löst das Problem durch abgeleitete Indizes.

```
CREATE INDEX I_PLUS_BSTQTY_QTY
ON PRDKT00(PRDBSTQTY+PRDQTY);
```

Die SQE liest die Daten jetzt anhand des abgeleiteten Index „I_PLUS_BSTQTY_QTY“ in sortierter Folge.



Mathematische Ausdrücke „ORDER BY“ V5R4

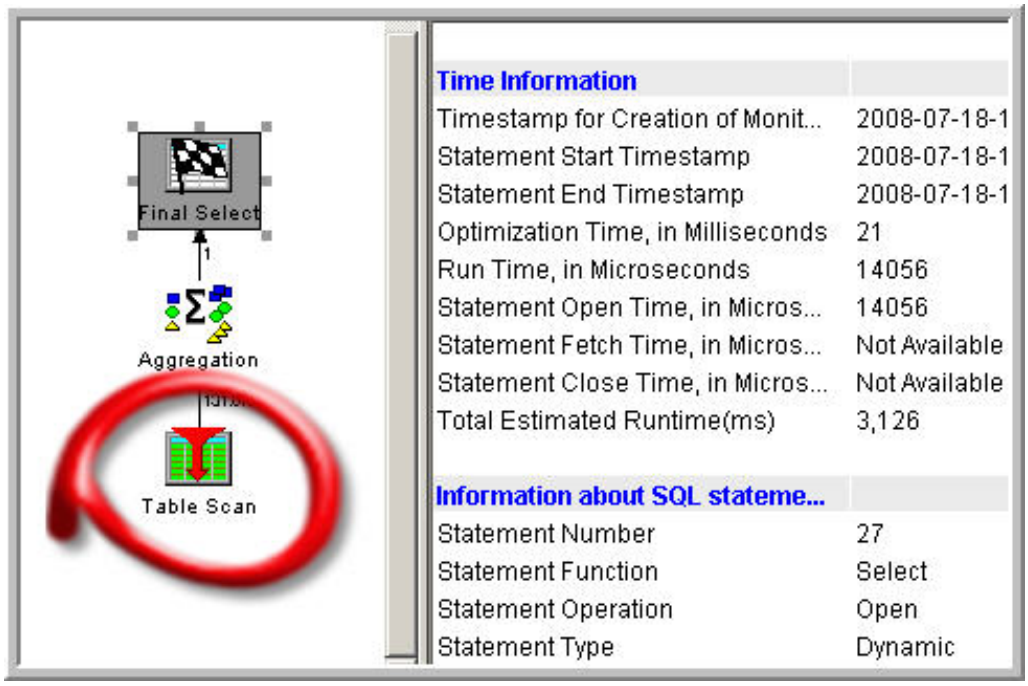
Alle temporären Sortieralgorithmen entfallen. Die Abfrage benötigt daraufhin anstatt 90,944 ms nur noch 0,0063 ms.

Aber auch wenn Sie Aggregatfunktionen einsetzen bringen die neuen Indizes Vorteile: Nehmen wir an, wir möchten wissen, wie viele Artikel mehr als 100.000,00 Euro Bestandswert haben. Hierbei sollen wieder die aktuellen Bestellmengen berücksichtigt werden. Also müssten wir folgende SQL-Abfrage schreiben:

```
SELECT COUNT(*) AS “Anzahl”
FROM PRDKT00
WHERE (PRDQTY + PRDBSTQTY) > 100000;
```

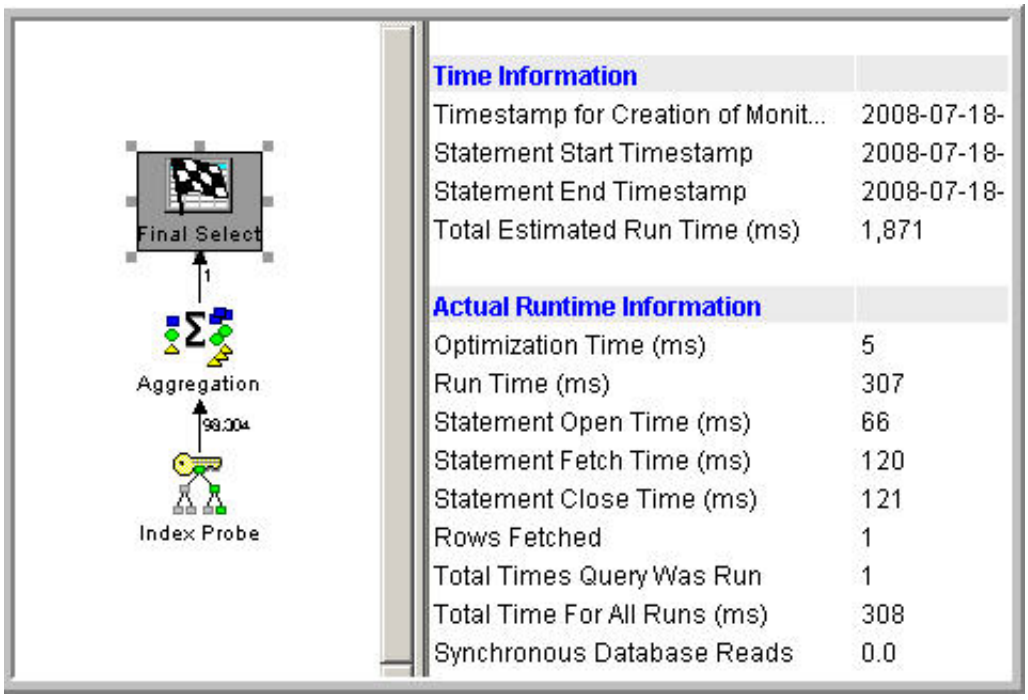
Meine Abfrage enthält in diesem Fall in der „WHERE“-Klausel einen mathematischen Ausdruck. Die SQE kann zwar das SQL-Statement ausführen, aber sie muss – wie der folgende Ausführungsplan zeigt – aufgrund der Aggregatfunktion „COUNT“ die Tabelle sequentiell lesen, obwohl zwei eigenständige Indizes für die Abfrage vorhanden sind. Die Auflösung des Ausdruckes erfolgt wieder zur Laufzeit.

7.3.5
Seite 34



Verwendung von Ausdrücken in lokalen Selektionskriterien V5R4

Auch hier ist der zuvor erstellte Index „I_PLUS_BSTQTY_QTY“ nützlich. Der Index enthält die notwendigen Informationen, so dass ein Zugriff auf die Tabelle „PRDKT00“ unnötig wird.



Einsatz abgeleiteter Index mit mathematischer Funktion V6R1

Abgeleitete Indizes können von der SQE auch genutzt werden, wenn Sie die Ausdrücke in „WHERE“-Klauseln ohne Aggregatfunktion verwenden. Hier habe ich aber bisher die Erfahrung gemacht, dass die Indizes für die Spalten „PrdQty“ und „PrdBstQty“ vorher gelöscht werden müssen, damit der neue Index verwendet wird. Grundsätzlich sollten Sie am Anfang sehr genau prüfen, ob die abgeleiteten Indizes den erhofften Nutzen bringen, und zunächst sehr einfache Ausdrücke verwenden. Je komplexer die Ausdrücke aufgebaut sind, je unwahrscheinlicher wird die Verwendung des Index. Laut IBM spielt in diesen Fällen auch die Spaltenfolge in der „WHERE“-Bedingung eine entscheidende Rolle. Wurde der Index – wie bei mir – mit der Spaltenfolge „PRDQTY“ und „PRDBSTQTY“ erstellt, müssen die Abfragen die Indexspaltenfolge beachten. Diese Erfahrung habe ich bislang nicht machen können. Das kann auf Ihrem Server mit anderer Laufzeitumgebung und anderem Datenbestand anders sein. Probieren Sie es aus! Sie finden sicherlich sehr schnell weitere sinnvolle Einsatzmöglichkeiten für den Einsatz abgeleiteter Indizes und lösen auf diesem Weg das eine oder andere Laufzeitproblem.

IBM hat mit Einführung der abgeleiteten Indizes im Übrigen auch die Möglichkeit geschaffen, abgeleitete logische DDS-Dateien zu ersetzen.

```
***** Datenanfang ***********************************
    R DETAILFMT                 PFILE(ITPSQL/DETAIL)
      DEKURZ
      MAKURZ
      BZKURZ
      BZLANG
    K DEKURZ
```

Logische DDS-Datei mit Satzformatnamen

Dieses DDS-Objekt konnte bislang nicht durch einen SQL-Index ersetzt werden. Hinderlich war einerseits der Satzformatname. Bislang war es lediglich möglich, den Tabellen eigenständige Satzformatnamen zu zuweisen. Seit Release V6R1 ist die Klausel RCDFMT auch für SQL Indizes vorhanden.

```
CREATE INDEX itpsql.i_DEKURZ on itpsql.DETAIL(DEKURZ)
rcdfmt DETAILFMT;
```

7.3.5
Seite 36

Des Weiteren besteht mit DDS die Möglichkeit, alle Spalten der physischen Datei in das Satzformat der logischen Datei zu übernehmen oder nur bestimmte Spalten im Satzformat zu benennen. Auch diese Möglichkeit kannte der SQL-Index bislang nicht. Mit Release V6R1 kommt daher eine weitere Klausel hinzu:

```
CREATE INDEX ITPSQL.I_DE_C01 ON ITPSQL.DETAIL (DEKURZ ASC)
RCDFMT DEKURZFMT ADD ALL COLUMNS;
```

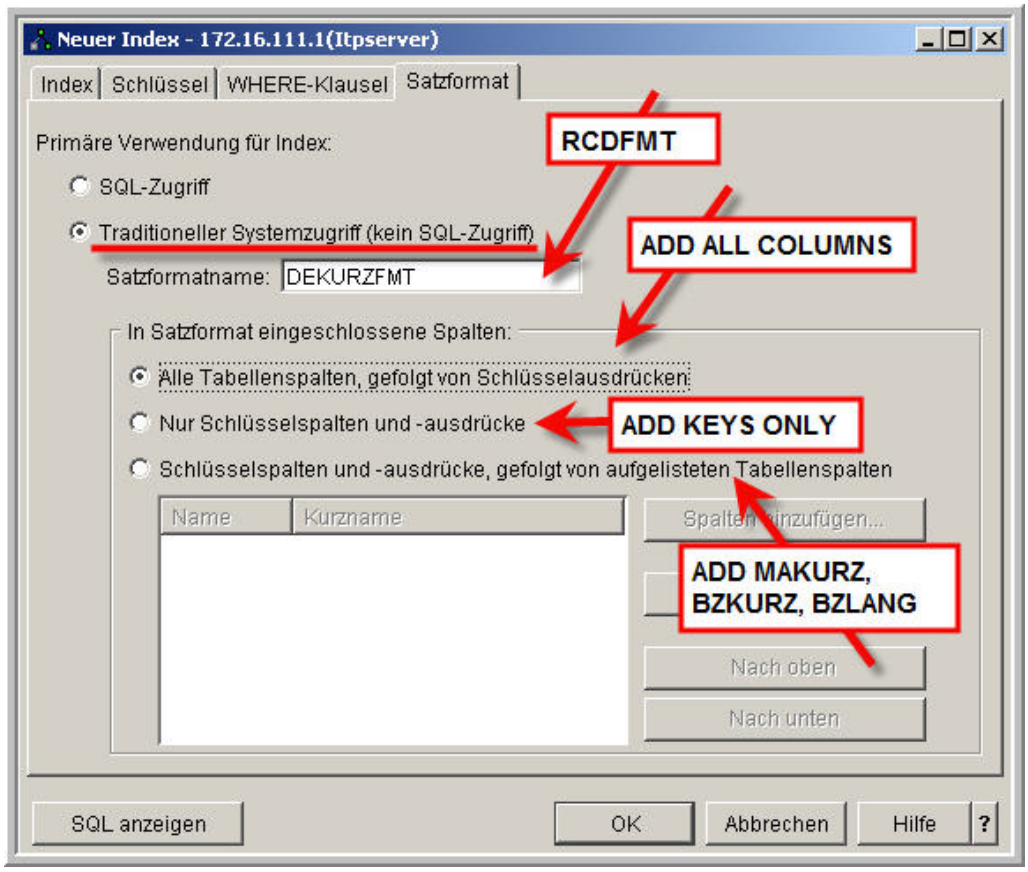
Um alle Spalten ins Satzformat zu übernehmen, müsste die Klausel ADD ALL COLUMS verwendet werden. „ADD ALL COLUMS“ ist im Übrigen der Unterlassungswert. Sollen hingegen lediglich die Schlüsselfelder im Satzformat vorhanden sein, müsste die Klausel ADD KEYS ONLY verwendet werden.

```
CREATE INDEX ITPSQL. I_DE_C02 ON ITPSQL.DETAIL (DEKURZ ASC)
RCDFMT DEKURZFMT ADD KEYS ONLY;
```

Und letztendlich können Sie die Felder des Satzformates jetzt auch mit SQL individuell bestimmen, wie im Ausgangsbeispiel mit DDS gezeigt:

```
CREATE INDEX ITPSQL. I_DE_C03 ON ITPSQL.DETAIL (DEKURZ ASC)
RCDFMT DEKURZFMT ADD DEKURZ, MAKURZ, BZKURZ, BZLANG;
```

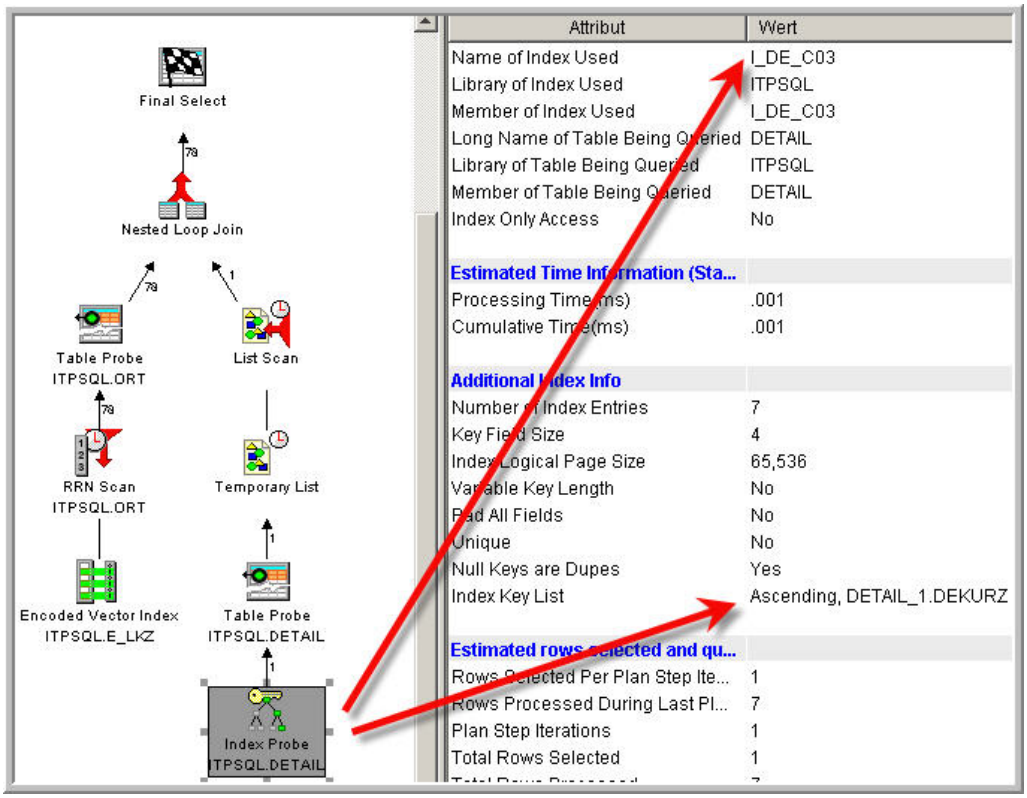
Aufgrund dieser neuen Klauseln wurde in System i Navigator der Dialog „Neuer Index“ um das Register „Satzformat“ erweitert:



Dialog „Neuer Index“ V6R1

7.3.5
Seite 38

Sie können den Satzformatnamen benennen, alle Spalten, nur die Schlüsselspalten oder die Schlüsselspalten und individuelle benannte Spalten im Satzformat aufnehmen. Außerdem legen Sie die primäre Verwendung des Index fest. Hier könnte schnell der Eindruck entstehen, dass Indizes, die primär für den traditionellen Systemzugriff erstellt werden, nicht vom SQL Optimizer während des Optimierungsprozesses berücksichtigt werden. Dies ist jedoch nicht der Fall wie der folgende Ausführungsplan schnell beweist:



SQL Index – Traditioneller Systemzugriff

Im Optimierungsprozess nicht berücksichtigt werden lediglich Indizes, die abgeleitete logische DDS-Dateien ersetzen. Schauen wir uns noch einmal eine DDS-Ausgangssituation an:

```
.A.Name++++++.LängeDDsF.....Funktionen++++++++++
*** Datenanfang ********************************
  R DETAILFMT_                PFILE(ITPSQL/DETAIL)
  K DEKURZ
  S DEKURZ                    RANGE(10 30)
```

Logische DDS-Datei abgeleitet

Dieses Mal werden alle Felder der physischen Datei übernommen, aber die logische DDS-Datei enthält Selektionskriterien. Es handelt sich um eine „abgeleitete“ DDS-Datei. Derartige Selektionen können im Release V6R1 ebenfalls mit SQL abgebildet werden.

```
CREATE INDEX ITPSQL.I_DETAIL02 ON ITPSQL.DETAIL (DEKURZ ASC)
WHERE DEKURZ BETWEEN 10 and 30
RCDFMT DETAILFMT ADD ALL COLUMNS;
```

Der Index erhält jetzt eine zusätzliche „WHERE“-Bedingung. Dieser abgeleitete Index wird **nicht** im Optimierungsprozess berücksichtigt; er soll ausschließlich helfen, DDS-Objekte vollständig abzulösen. Bisher waren abgeleitete logische DDS-Dateien ein Handicap bei der Abschaffung der DDS-Dateien. Dieses Handicap hat IBM im Release V6R1 beseitigt! Es bleibt abzuwarten, ob und wann IBM sich von DDS-Syntaxen im Datenbankumfeld vollständig trennen wird. Sie sollten nicht so lange mit der Umstellung Ihrer Dateibeschreibungen warten, sondern bereits jetzt beginnen sukzessive Ihre DDS-Quellen durch SQL-Syntax ersetzen.

7.3.5
Seite 40

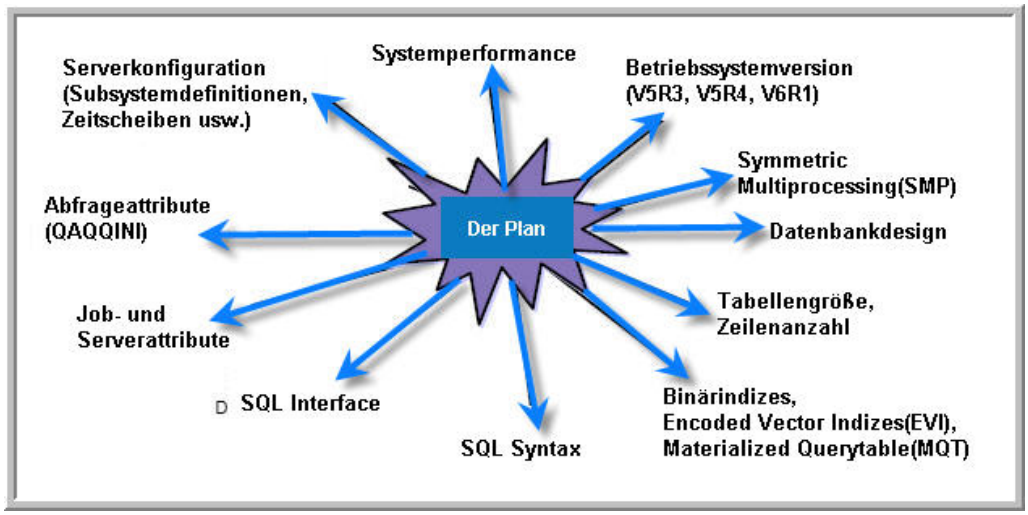
7.3.6 SQL-INDIZES – Einsatz und Wirkung

Sicherlich konnten Sie bereits erste Erfahrungen mit SQL-Abfragen sammeln. Dabei haben Sie bestimmt festgestellt, dass die SQL Engine einige SQL-Abfragen zufrieden stellend, andere nur sehr langsam ausführt. Vielleicht ist sogar das Laufzeitverhalten schlecht, obwohl Indizes eingesetzt werden? Mit den SQL-Abfragen ist es manchmal wie verhext – mal sind sie trotz großer Komplexität schnell, dann wieder erstaunlich langsam. Manch einer ist sogar bereit, nach derartigen Erfahrungen auf SQL zu verzichten. Häufig wird das stark variierende Laufzeitverhalten aber nicht von SQL verursacht – Schuld sind fehlende Indizes oder die verwendete SQL-Syntax. Hinzu kommt, dass IBM mit jedem Release neue Features und Zugriffsmethoden für den SQL-Optimizer implementiert, die Ihnen aber auch dem Optimizer neue Möglichkeiten eröffnet. In diesem Kapitel möchte ich Ihnen die Wirkungsweise der Indizes anhand verschiedener SQL-Beispiele erläutern, so dass Sie zukünftig besser entscheiden können, welche Indizes für Ihre Applikation erforderlich sind.

Doch die Wahl des Ausführungsplanes ist nicht nur von vorhandenen Indizes abhängig, sie wird auch durch die Laufzeitumgebung Ihrer SQL-Abfragen bestimmt. SQL-Abfragen werden auf unterschiedlichen System i Servern in unterschiedlichen Umgebungen ausgeführt und nutzen heute verschiedene Interfaces:

- Web/J2EE, DB2 for i5/OS via JDBC
- Client Server-Applikationen: JDBC, ODBC, CLI
- Client Server-Applikationen via ADO/OLEDB Provider
- HLL-Programme mit Embedded SQL Statements

Bei der Erstellung des Ausführungsplans wird der Optimizer von der Konfiguration des Servers aber auch von vielen weiteren Komponenten beeinflusst:



Faktoren für die Planerstellung

Natürlich hat die verwendete Hard- und Software Einfluss auf die Wahl des Ausführungsplanes. Der Optimizer berücksichtigt bei seinen Entscheidungen die Größe des Hauptspeichers, die Anzahl der Prozessoren und auch das installierte Release ist von Bedeutung, aber auch Systemwerteinstellungen (u. a. QPFRADJ, QQRYDEGREE) beeinflussen den Optimizer. Hinzu kommen das Datenbankdesign und die Tabellengrößen und natürlich auch die Attributeinstellungen der Optionsdatei „QAQQINI“ sowie das verwendete Optimierungsziel, um nur einige Komponenten zu nennen. Ändern Sie eine der Komponenten, dann ändert sich sehr wahrscheinlich auch der Ausführungsplan. Deutlich wird dieses Verhalten, wenn wir uns zunächst einmal die Komponente „Optimierungsziel“ anschauen.

7.3.6.1 Das Optimierungsziel

Es ist heute selbstverständlich, dass SQL-Abfragen interaktiv ausgeführt werden. Der Anwender betrachtet die erste Ergebnisseite und bricht anschließend die Abfrage ab. In diesem Szenario sollte der Benutzer nicht all zu lange auf das Ergebnis der Abfrage warten müssen. In anderen Fällen werden SQL-Abfragen ausgeführt, um Reports zu generieren. Normalerweise weiß der Anwender, dass die Erstellung eines solchen Berichtes einige Zeit dauern kann. Für beide Szenarien gibt es unterschiedliche Optimierungsziele.

Wie ist dieses Optimierungsziel zu verstehen? Stellen Sie sich vor, Sie wollen eine Strecke von A nach B zurücklegen. Dafür stehen Ihnen unterschiedliche Transportmittel zur Verfügung: Flugzeug, Auto, Bus, Fahrrad. Das Flugzeug ist nicht grundsätzlich die beste Wahl, denn wenn Sie eine Wegstrecke von 500 Metern überwinden müssen, ist das Flugzeug noch nicht einmal gestartet während Sie mit dem Fahrrad bereits Ihr Ziel erreicht haben. Die Wahl des Transportmittels ist somit abhängig von der Länge der Wegstrecke, der Verfügbarkeit des Transportmittels und weiterer Faktoren. Und genau so verhält es sich auch mit der Wahl des jeweiligen Ausführungsplanes durch den Optimizer. Grundsätzlich unterscheidet der Optimizer hierbei zwei unterschiedliche Optimierungsziele:

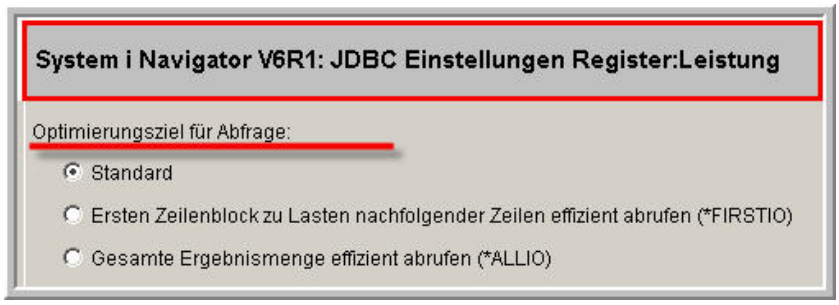
- *FIRSTIO
  Die Abfragen werden dahingehend optimiert, dass die erste Seite des Result Set möglichst schnell angezeigt werden kann zu Lasten der nachfolgenden Seiten. Das Optimierungsziel sollten Sie wählen, wenn die Abfrage durch einen Anwender kontrolliert wird, der die Daten vermutlich nicht vollständig benötigt.

- *ALLIO
  Wenn hingegen Reports generiert werden, die Ergebnismengen in Tabellen speichern oder Summen und Durchschnittswerte bilden, dann müssen die Abfragen dahingehend optimiert werden, alle Tabellenzeilen vollständig zu verarbeiten. Die Abfragen werden mit dem Ziel optimiert, die komplette Abfrage in der kürzestmöglichen Zeit zum Abschluss zu bringen.

Da stellt sich natürlich die Frage wie das Optimierungsziel vom Optimizer bestimmt wird. Ihre Schnittstellenkonfiguration, Einstellungen der Optionsdatei „QAQQINI“ und/oder entsprechende Compiler-Direktiven bestimmen das grundlegende Optimierungsziel. In der Prozedurumgebung vom System i Navigator V5R4 ist beispielsweise das Attribut „OPTIMIZATION_GOAL“ der Attributdatei „QAQQINI“ maßgebend. Das Attribut gibt das Ziel vor, das das Optimierungsprogramm für Abfragen bei Entscheidungen zur Kostenrechnung berücksichtigen soll. Die entsprechende Defaulteinstellung lautet *DEFAULT. Das Optimierungsziel wird dadurch von der verwendeten Schnittstelle (ODBC, SQL Pre-Compileroptionen, OPTIMIZE FOR nnn ROWS-Klausel) bestimmt.

System i Navigator bietet seit Release V6R1 eine weitere Möglichkeit, das Optimierungsziel zu bestimmen:

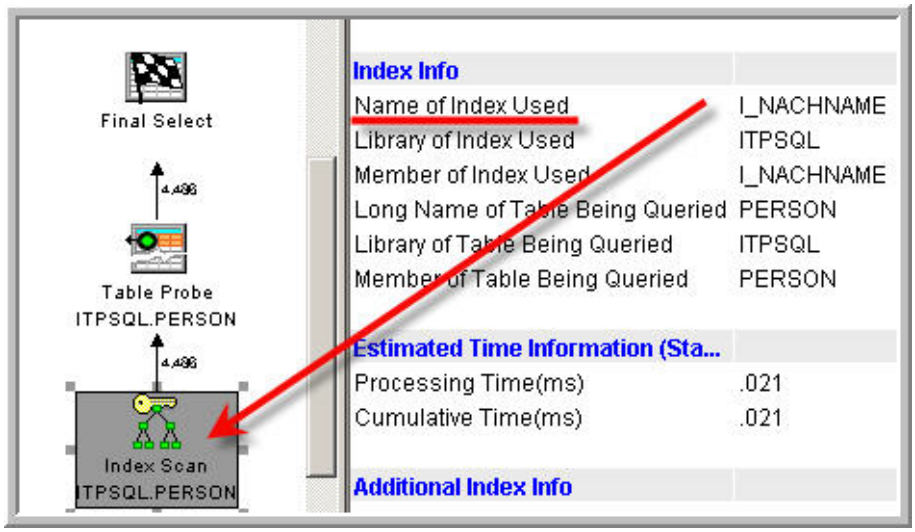
Öffnen Sie hierfür den Menüpunkt „Verbindungen“ und wählen Sie „JDBC-Setup“. Klicken Sie anschließend auf das Register „Leistung“:



JDBC Einstellung System i Navigator V6R1 – Optimierungsziel

7.3.6
Seite 4

Im Register „Leistung“ hinterlegen Sie für die aktuelle JDBC-Verbindung das grundlegende Optimierungsziel. Aufgrund der Defaulteinstellungen wird der Optimizer versuchen, die ersten Zeilen der Abfrage möglichst effizient abzurufen, d. h. der Optimizer bevorzugt für die Ausführung einen Index. Schauen Sie sich das nachfolgende Beispiel an:



*Optimierungsziel *First I/O*

Ausgeführt wurde das folgende SQL Statement:

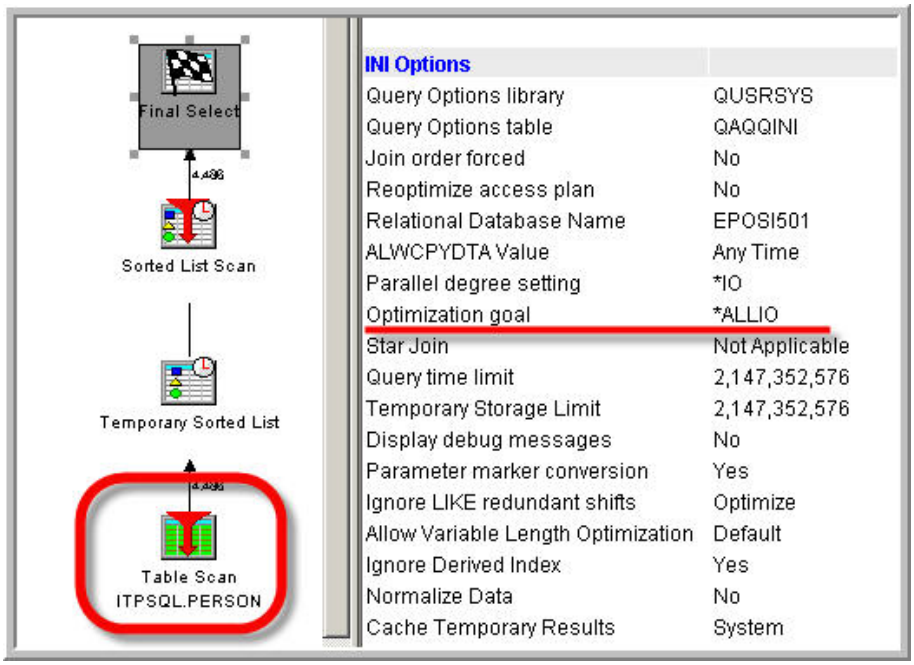
```
select * from person order by nachname;
```

Eine sortierte Kundenliste wird benötigt. Da ein Index für die Spalte „Nachname“ existiert, ist es nicht weiter verwunderlich, dass der Optimizer sich entscheidet zur Ausführung der Abfrage den Index „I_Nachname“ zu verwenden, um effizient die ersten Zeilen des Result Sets anzuzeigen. Ändere ich das Optimierungsziel im JDBC-Setup meiner aktuellen Sitzung, würde sich auch der Ausführungsplan ändern.

Es gibt aber eine andere Möglichkeit das Optimierungsziel zu beeinflussen. Sie können in der SQL Abfrage die Klausel „OPTIMIZE FOR … ROWS“ verwenden. Sie müssen nicht einmal die genaue Satzanzahl kennen. Es ist ausreichend, wenn Sie eine Vorstellung von der zu erwartenden Datenmenge haben. Schauen Sie sich die Wirkung an, wenn ich die vorherige Abfrage ein zweites Mal ausführe:

```
select * from person order by nachname
OPTIMIZE FOR ALL ROWS;
```

Durch die Klausel ändert sich das Optimierungsziel und dadurch auch der Ausführungsplan:



*Optimierungsziel *All I/O*

Der Optimizer verzichtet auf den Index, lädt stattdessen die Tabelle vollständig in den Hauptspeicher und nutzt zur Sortierung der Daten temporäre Listen. Die Ausführung des Statements ohne Index ist unter Berücksichtigung der gesamten Datenmenge im Übrigen deutlich schneller als ein vergleichbarer Zugriff über Index.

7.3.6.2 Single Table Optimierung

Schon dieses erste Beispiel zeigt, dass der Optimizer Indizes sehr unterschiedlich einsetzt und vielleicht mussten auch Sie bereits das ein oder andere Mal feststellen, dass die SQL Engine Ihre Indizes gar nicht nutzt oder sogar eigene Indizes erstellt? Welche Indizes werden für die unterschiedlichen Abfragen benötigt? Welche Indizes sind unnötig? Um diese Fragen zu beantworten, ist es einerseits hilfreich, den Entscheidungsprozess des Optimizers zu verstehen, andererseits müssen Sie die Datenbank und die Applikation kennen, um eine geschickte Indexstrategie aufzubauen. Und ein weiterer Gesichtspunkt darf nicht unberücksichtigt bleiben:

Optimieren Sie Ihre Anwendung oder das Programm zum ersten Mal und legen somit die Grundlagen (**Proaktive Indexstrategie**)?

Oder optimieren Sie einzelne SQL Statements oder eine Gruppe von SQL Statements, deren Laufzeitverhalten schlecht ist (**Reaktive Indexstrategie**)?

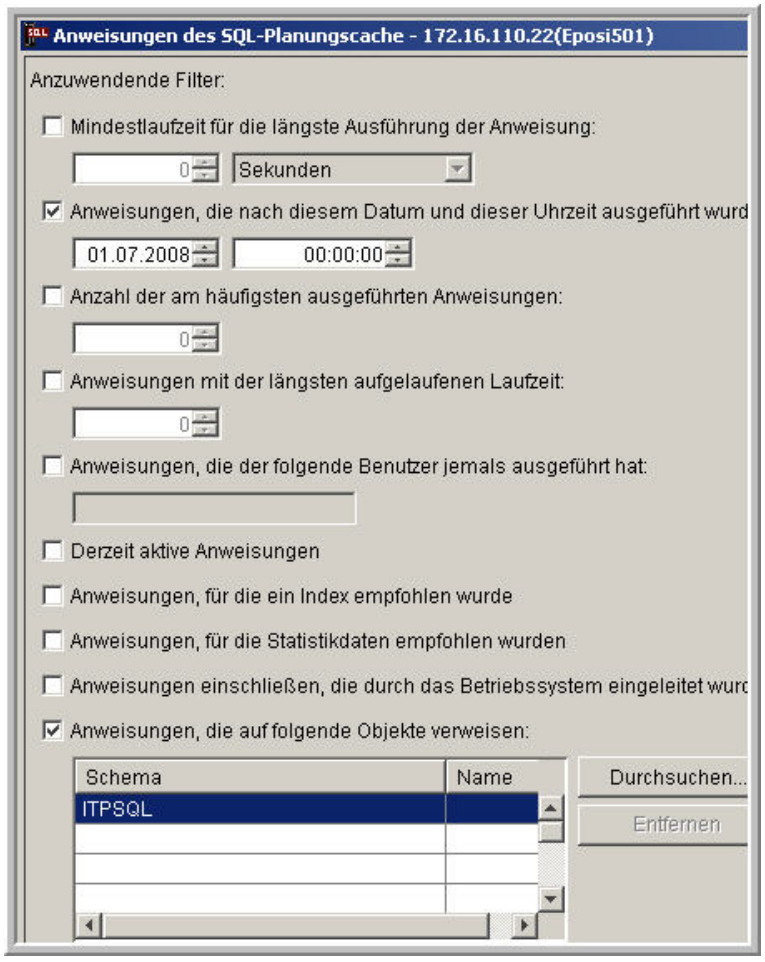
Sie sollten sich zunächst nicht auf das Laufzeitverhalten einzelner SQL Abfragen konzentrieren, sondern die richtige Basis legen. Doch bevor Sie damit beginnen, gilt es folgende Fragen zu klären:

- Welche Informationen benötigen Ihre Anwender?
- Welche davon häufig und welche eher selten?
- Welche Abfragen werden interaktiv ausgeführt?
- Welche Berichte werden generiert?
- Welche Abfragen laufen nachts?
- Welches Interface wird verwendet?
- Welche Tabellen sind von entscheidender Bedeutung?
- Welche Relationen existieren?
- Welche Laufzeitumgebung steht zur Verfügung?
- Welches Release ist installiert?
- usw.

Ohne ausreichende Kenntnis der Laufzeitumgebung, der SQL Abfragen und der betroffenen Tabellen, ist es nicht möglich geeignete Indizes zu erstellen. Sind Sie sich nicht sicher, welche Informationen abgefragt werden und wie häufig die Informationen benötigt werden, kann der Plan Cache helfen.

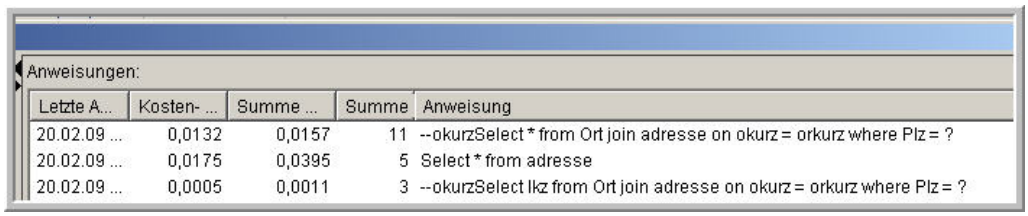
Hierfür öffnen Sie in System i Navigator die „Datenbank“ ➜ „Momentaufnahmen für SQL Planungscache“ ➜ „Rechtsklick“ ➜ „SQL Planungscache“ ➜ „Anweisungen anzeigen…“. Daraufhin erscheint das bekannte Selektionsfenster.

Ich lasse mir alle Abfragen anzeigen, die seit dem 01.07.2008 für Tabellen im Schema „ITPSQL“ ausgeführt wurden.



Selektion häufig ausgeführter SQL-Abfragen (SQE)

Anhand des Ergebnisses erkenne ich schnell, welche Informationen benötigt wurden und welche Tabellen hierbei von besonderem Interesse waren. Berücksichtigen muss ich natürlich, dass nur Ausführungspläne der SQE angezeigt werden, die seit dem letzten IPL erstellt wurden.



Plan Cache Ergebnisse

Die Ausgabe sortiere ich nach der Anzahl der Ausführungen. Hierfür muss ich nur auf die Spalte „Summe der Ausführungen“ klicken. Ist die Anzeige zu umfangreich ist, können Sie die Filterkriterien verfeinern, indem Sie z. B. nur Pläne anzeigen, die für die Ausführung mehr als 10 Sekunden benötigten.

In einem weiteren Schritt verdeutliche ich mir außerdem die Tabellenrelationen und Tabellengrößen. Die aktuelle Satzanzahl, Vorkommen der Spaltenwerte usw. entnehme ich hierfür den Statistikdaten. Ausgangspunkt der anschließenden Betrachtung sollte im Anschluss eine einzelne Tabelle sein. Natürlich sind die komplexen Abfragen mit vielen Tabellen die interessanteren, aber auch der Optimizer optimiert komplexe Abfragen nicht zusammenhängend. Warum sollten wir es also tun?

Konzentrieren Sie sich also als erstes auf Ihre wichtigste Tabelle – die die meisten Datensätze enthält und an vielen Abfragen beteiligt ist. Um zu erläutern, anhand welcher Kriterien die Indizes für die Tabelle erstellt werden, nutze ich zwar im weiteren Verlauf beispielhaft einige SQL Abfragen, aber erklärtes Ziel ist es, Indizes zu erstellen, die vielfach zum Einsatz kommen und nicht nur einzelne Abfragen unterstützen.

Zunächst führe ich eine SQL Abfrage aus, ohne weitere Vorkehrungen zu treffen:

```
Select *
from adresse
where adrinsertuser = 'RASCHE';
```

Was ist passiert:

Während der Ausführung meiner Abfrage versucht der Optimizer, immer den bestmöglichen Ausführungsplan zu erstellen. Bester Ausführungsplan bedeutet in diesem Zusammenhang möglichst wenig I/O Operationen und effektive Verarbeitung der Datenmenge im Hauptspeicher. Hierzu durchläuft die Optimierung verschiedene Schritte:

1. Informationsbeschaffung

Die Satzanzahl für jedes Selektionskriterium wird anhand eines Index bestimmt, wenn die führenden Schlüsselspalten des Index mit den lokalen Selektionskriterien übereinstimmen. Diese Methode nennt man „Schlüsselanfrage“ (Key Range Estimate). Andernfalls nutzt der Optimizer entsprechende Spaltenstatistiken. Für meine Abfrage könnte ein Index mit der Spalte „ADRINSERTUSER“ diesen Prozess unterstützen. Ich vermeide – wenn möglich – die Erstellung von Indizes zu statischen Zwecken. Prüfe dagegen, ob für die Spalten der Tabelle „ADRESSE“ entsprechende Spaltenstatistiken vorliegen und erstelle diese notfalls auch schon mal manuell, um zu verhindern, dass die Anfragen des Optimizers zunächst mit Defaultwerten beantwortet werden.

2. Kostenberechnung der unterschiedlichen Zugriffspläne

Nachdem das Datenvolumen ermittelt ist, werden die Kosten für die zur Verfügung stehenden Ausführungsmethoden berechnet:

- Kosten für die Methode „Table Scan“
- Kosten für den Aufbau eines temporären Index und dessen anschließende Nutzung
- Kosten für eine temporäre Sortierroutine
- Kosten für Methoden, die Hashtabellen verwenden
- Kosten für die Verwendung permanenter Indizes.
  Die SQE untersucht immer alle für die Abfrage relevanten Indizes. Die Information hierfür erhält Sie vom Statistik Manager. Die relevanten Indizes werden vor der Untersuchung anhand der Größe des Zugriffspfades sortiert. Als erstes werden die Implementierungskosten für Indizes mit dem kleinsten Zugriffspfad, d. h. einem Schlüsselfeld geschätzt, danach Indizes mit dem nächst größerem Schlüsselpfad untersucht. Sind die Implementierungskosten für die aktuell untersuchte Indexgruppe größer als die zuvor ermittelten Kosten, wird die Suche beendet. Der Index mit dem kleinsten Kostenfaktor wird im weiteren Entscheidungsprozess berücksichtigt.
  Die CQE geht anders vor. Sie untersucht als erstes die zu letzt erstellten Indizes. Die Untersuchung ist zeitlich durch das Attribut „QUERY_TIME_LIMIT in der Abfrageoptionsdatei QAQQINI begrenzt. Nach Ablauf der Zeit entscheidet sich die CQE für den Index mit dem kleinsten Kostenfaktor. Aufgrund der zeitlichen Limitierung kann es passieren, dass einige Indizes nicht berücksichtigt werden.

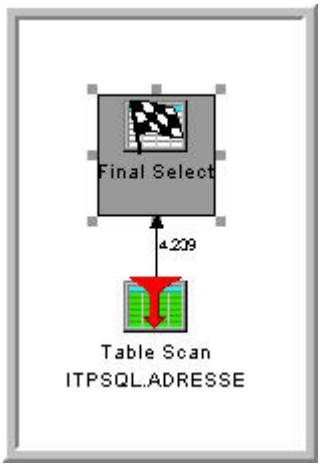
Ergänzung 9/2009

7.3.6
Seite 10

3. Die Entscheidung

Erst jetzt, wenn die Kosten aller Ausführungsmethoden ermittelt sind, entscheidet sich der Optimizer für den Plan mit dem geringsten Kostenfaktor und führt die SQL Abfrage aus.

Da in meiner Testumgebung keine Indizes existieren, erscheint das Symbol „Table Scan“ im Ausführungsplan.



Ausführungsplan der Beispielabfrage

In meiner Testumgebung hat der Optimizer zunächst keine Wahl. Er muss sich für die Methode „Table Scan“ entscheiden, da derzeit keine Indizes zur Verfügung stehen. Erscheint – wie in meinem Beispiel – das Symbol „Table Scan“, schauen viele erst einmal nach, „Was sagt der Index Advisor“? Der Index Advisor erhält vom Optimizer Rückmeldungen über den Optimierungsprozess und erstellt anhand dieser Informationen mehr oder weniger geeignete Indexvorschläge. In diesem Fall hat der Index Advisor folgenden Vorschlag für uns:

Indexvorschlag für die Abfrage

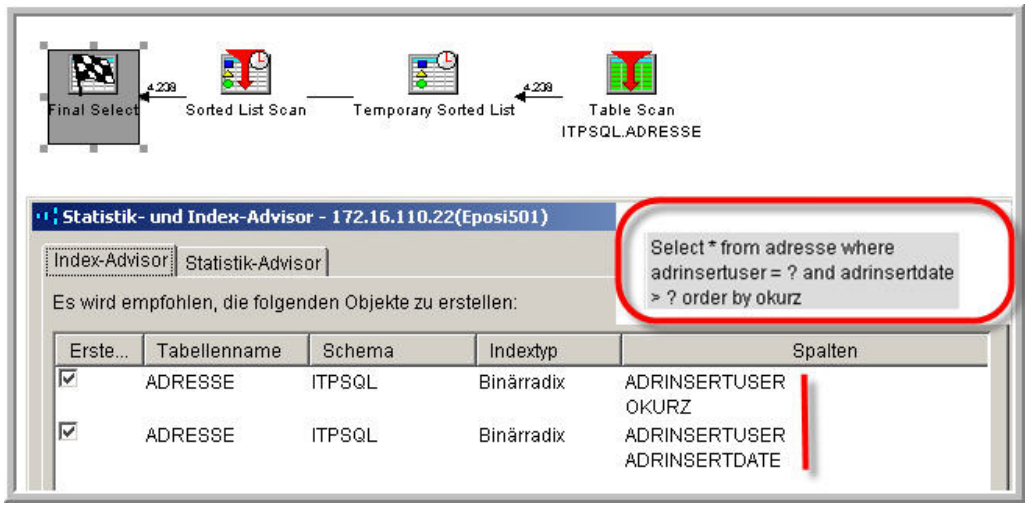
Passen Sie auf, was passiert, wenn ich die Abfrage ein klein wenig verändere und alle Adressdaten selektiere, die heute vom Benutzer „Rasche“ erfasst wurden:

```
Select *
from adresse
where adrinsertuser = 'RASCHE'
and adrinsertdate > '2009-02-20 00:00:00.000000';
```

Der Ausführungsplan bleibt der gleiche, aber der Index Advisor hat jetzt plötzlich eine neue Idee:

Der neue Indexvorschlag

Es wird ein zusammengesetzter Index für die Spalte „ADRINSERTUSER“ und „ADRINSERTDATE“ vorgeschlagen. Ändere ich aber die Abfrage erneut und lasse die Daten sortieren, erhalte ich neue Indexvorschläge!



Die nächsten Indexvorschläge

Natürlich ändert sich dieses Mal auch der Ausführungsplan, da die Daten sortiert werden. Aber auch der Index Advisor ändert seine Meinung und schlägt mir jetzt gleich zwei Indizes vor. Welches ist die bessere Wahl?

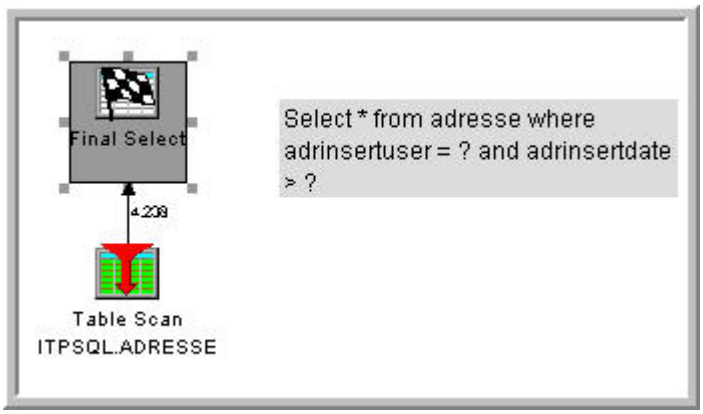
7.3.6
Seite 12

Die wenigen Beispiele zeigen, dass wir zu diesem Zeitpunkt nicht den Vorschlägen der Index Advisors folgen können. Ich würde für jedes SQL Statement individuelle Indizes erstellen, denn der Index Advisor bezieht seine Vorschläge stets auf die einzelne Abfrage und das verwendete Optimierungsziel ohne Berücksichtigung der aktuellen Datenlage und Arbeitsumgebung.

Wir brauchen eine Strategie! Betrachten Sie nicht die soeben ausgeführte Abfrage, sondern untersuchen Sie die gesamte Tabelle. Als Minimum benötigen Sie Binärindizes für wichtige lokale Selektionskriterien. Das heißt aber nicht, dass Sie für jede Spalte Ihrer Tabellen einen Index erstellen müssen. Gute Indexkandidaten sind häufig verwendete Spalten, die eine hohe Selektivität besitzen. In der Tabelle „ADRESSE“ sind dies u. a. die Spalten „AdressArt“, „AdrInsertUser“ und „AdrInsertDate“.

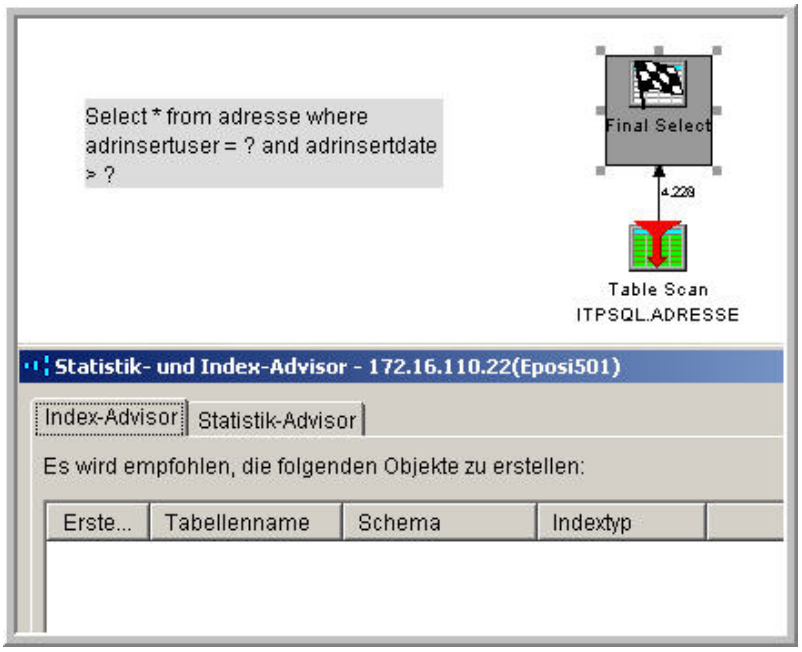
```
Create Index I_AdrInsertDate on Adresse(AdrInsertDate);

Create Index I_AdrInserUser on Adresse(AdrInsertUser);

Create Index I_AdressArt on Adresse(AdressArt);

CREATE INDEX I_ADKURZ ON ADRESSE(ADKURZ);
```

Nachdem die Indizes erstellt sind, führe ich die Abfrage ein weiteres Mal aus:



Der Optimizer nutzt die erstellten Indizes nicht

Obwohl sowohl für die Spalte „AdrInsertUser“ als auch für die Spalte „AdrInsertDate“ ein Index erstellt wurde, nutzt der Optimizer diese Indizes nicht. Stattdessen beharrt er auf seinem Vorschlag für einen zusammengesetzten Index. Spaßeshalber erstelle ich den zusammengesetzten Index, aber auch jetzt verwendet der Optimizer weiterhin die Methode „Table Scan“. Lediglich der Indexvorschlag entfällt.



Der Optimizer ignoriert auch den zusammengesetzten Index

Wie ist dieses Verhalten zu erklären? Eines ist klar, es liegt nicht am zusammengesetzten Index. Den zusammengesetzten Index kann ich daher für den Moment wieder löschen:

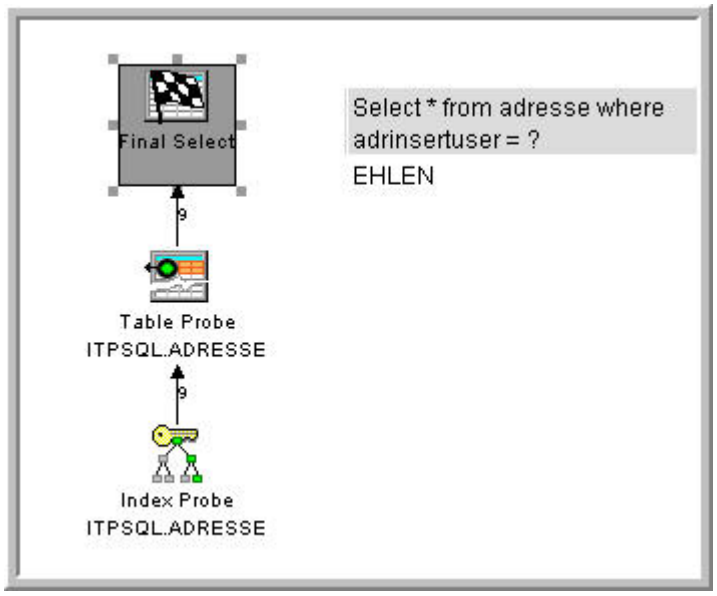
```
DROP Index I_AdrInsertUser_AdrInsertDate;
```

Der Optimizer ignoriert nämlich alle vorhandenen Indizes, weil während des Optimierungsprozesses festgestellt wird, dass mehr als 80 % der Daten abgerufen werden. Binärindizes sind aber oft nur nützlich, wenn weniger als 20 % der Datensätze benötigt werden. Diese Behauptung lässt sich schnell beweisen. Ich ändere einfach den Hostvariablenwert:

```
Select *
from adresse
where adrinsertuser = 'EHLEN'
and adrinsertdate > '2009-02-20 00:00:00.000000';
```

7.3.6
Seite 14

Jetzt nutzt der Optimizer sofort den vorhandenen Index, wie die nachfolgende Abbildung zeigt.



Geänderte Hostvariablenwerte

Sie sehen, wenn die Bedingungen für den Einsatz von Indizes erfüllt sind, können auch einfache Indizes nützlich sein, obwohl der Index Advisor weiterhin auf seinen Vorschlag für einen zusammengesetzten Index besteht.

In der Regel enthalten unsere SQL Abfragen komplexe Kombinationen verschiedener Selektionskriterien. Sie können für **häufige Spaltenkombinationen** durchaus zusammengesetzte Binärindizes erstellen und das weiß auch der Optimizer. Er kann allerdings nicht beurteilen, wie häufig die jeweiligen Spaltenkombinationen genutzt werden, und schlägt Ihnen daher grundsätzlich bei mehreren Selektionskriterien zusammengesetzte Schlüssel vor. Wenn Sie zusammengesetzte Indizes erstellen, ist die Reihenfolge der Schlüsselspalten entscheidend:

Verwenden Sie zunächst Spalten, die den Gleich-Operator nutzen. Gibt es mehrere Spalten die diese Bedingung erfüllen, wird die Spalte mit der höchsten Selektivität vorangestellt. Lokale Selektionskriterien mit anderen Vergleichsoperatoren fügen Sie an.

Schauen Sie sich das nächste Beispiel an. Es verdeutlicht was gemeint ist:

```
Select *
from adresse
where adrinsertuser = ,EHLEN‘
and adrinsertdate > ,2009-02-20 00:00:00.000000‘;
```

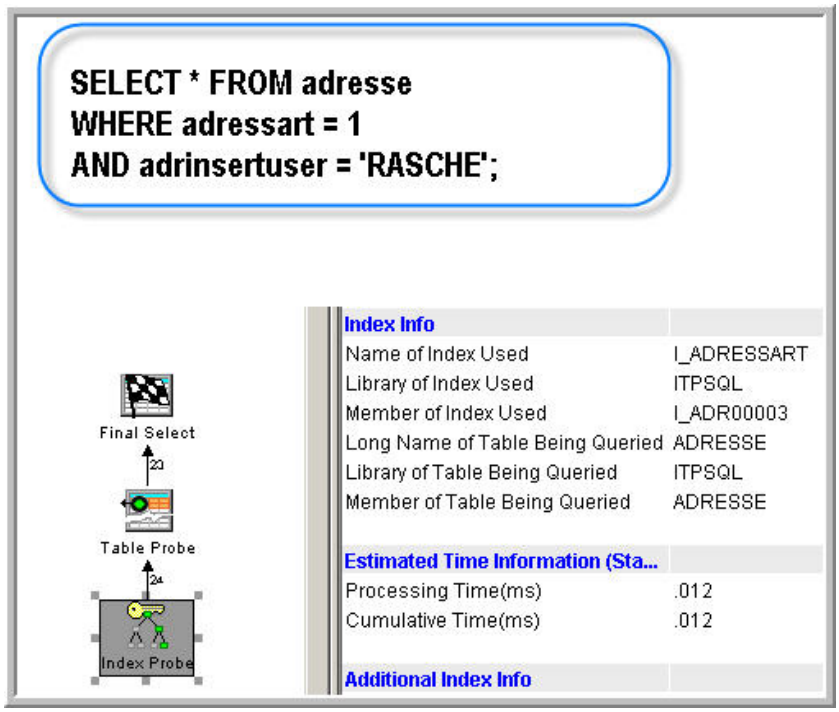
Der Index Advisor schlägt uns für diese Abfrage vor, folgenden Index zu erstellen:

Zusammengesetzter Indexvorschlag

Der Vorschlag als solcher ist nachzuvollziehen. Es handelt sich um die Kombination der lokalen Selektionskriterien. Die Reihenfolge der Spalten wird durch die verwendeten Operatoren bestimmt. Aus diesem Grund wird die Spalte „AdrInsertDate“ angefügt. Der Index Advisor weiß nicht, dass ich in anderen Abfragen durchaus auch für die Spalte „AdrInsertDate“ den Gleich-Operator verwenden könnte. Er kann auch nicht beurteilen, welche der beiden Spalten die höhere Selektivität besitzt. Es muss also nicht zwingend die vorgeschlagene Reihenfolge des Index Advisors sein. Manchmal ist es besser, selbst ein wenig über die Verwendung der Spalten und deren Selektivität nachzudenken und eigene Entscheidungen zu treffen.

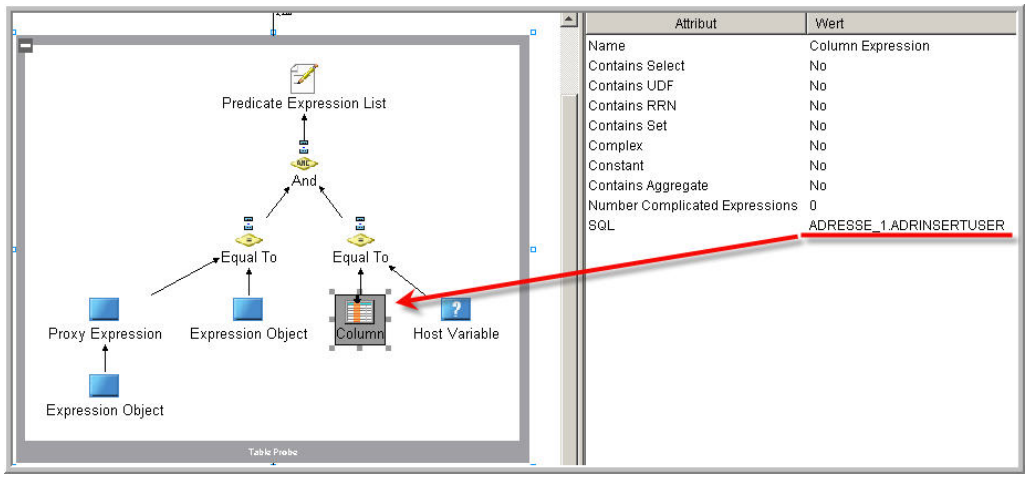
7.3.6.3 Encoded Vector Index (EVI)

Sie können aber nicht für alle Spalten und Spaltenkombinationen Binärindizes erstellen. Zudem haben Sie gesehen, dass der Optimizer Binärindizes meidet, wenn die Datenmenge zu groß ist. So kann es immer wieder passieren, dass der Optimizer für die Ausführung einer Abfrage nur die Methode „Table Scan“ verwendet wie in unserem ersten Beispiel.



Fehlende Schlüsselspalten

Der Ausführungsplan sieht auf den ersten Blick perfekt aus! Ist er aber nicht! Da kein zusammengesetzter Binärindex für die Spalte „AdressArt“ und „AdrInsertUser“ erstellt wurde, kann eine der Bedingungen erst geprüft werden, wenn die Daten der Tabelle „ADRESSE“ in den Hauptspeicher übertragen werden. Sie glauben mir nicht? Lassen Sie sich die vollständigen Diagrammdetails in Visual Explain anzeigen. Hierfür wählen Sie in Visual Explain das Menü „Optionen“ ➜ „Diagrammdetail“ ➜ „Vollständig“ und erweitern anschließend die Anzeige der Methode „Table Probe“:



Erweiterte Anzeige der Methode „Table Probe“

Jetzt können Sie sehen, welche Bedingungen der Optimizer während der Untersuchung der Tabelle prüft. Was kann ich tun?

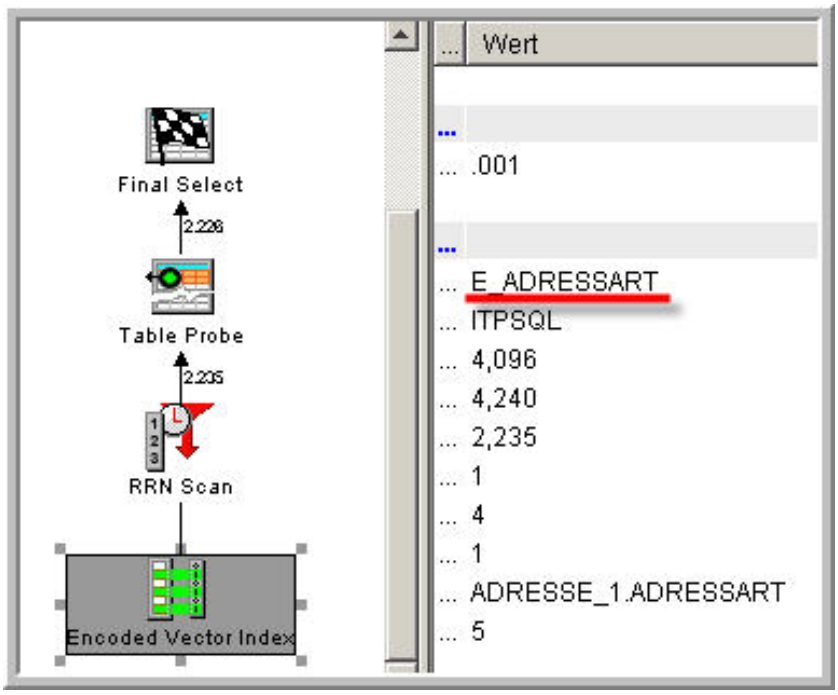
Eine Kombination der Spalten „AdressArt“ und „AdrInsertUser“ ist eine Möglichkeit.

Eine Alternative zum Binärindex wären auch „Encoded Vector Indizes“ (EVI).

```
CREATE ENCODED VECTOR INDEX E_adrinsertuser
ON adresse(adrinsertuser);


CREATE ENCODED VECTOR INDEX E_adressart
ON adresse(adressart);
```

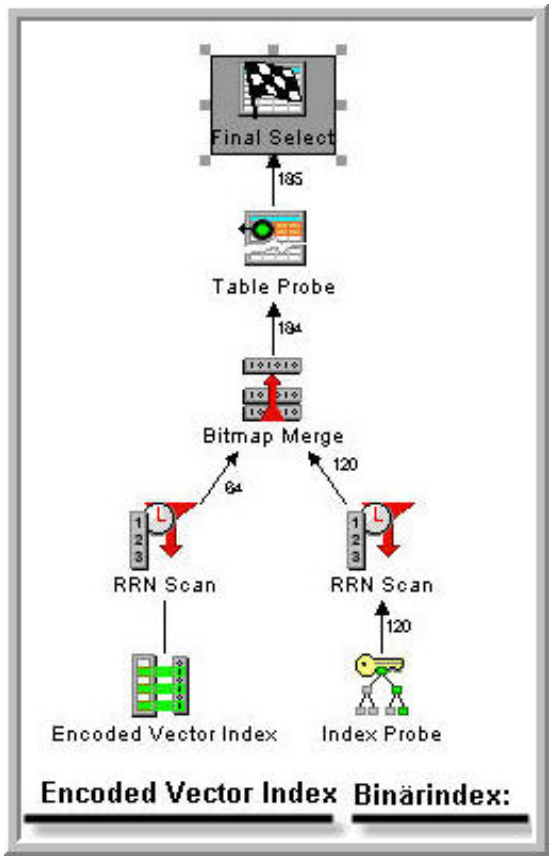
Der Zugriff über „Encoded Vector Index“ (EVI) ist häufig schneller, weil der Index kleiner ist und daher schneller in den Arbeitsspeicher verlagert werden kann, und da die SQL Engine auch Ergebnisse der EVI- und Binärindizes kombiniert, verwendet der Optimizer jetzt den EVI an Stelle meines Binärindex:

7.3.6
Seite 18



Vorteile des EVIs

Der Optimizer entscheidet sich für den EVI und gegen meinen Binärindex. Der Optimizer kann sogar Binär- und EVI Indizes kombinieren, falls die Datenlage dies erfordert.



Logische Verknüpfung EVI und Binärindex

Schnell stellt sich jetzt die Frage, wie viel schneller der Zugriff mittels EVI ist? Diese Frage kann nicht allgemein beantwortet werden, da viele Faktoren hierbei eine Rolle spielen. Häufig ist die Frage zu diesem Zeitpunkt ohnehin unerheblich, weil lediglich Werkzeuge bereitgestellt werden sollen, die dem Optimizer alternative Ausführungsstrategien ermöglichen. Aus diesem Grund können Sie zusammengesetzte Binärindizes um entsprechende „Encoded Vector Indizes“ ergänzen. Beachten sollten Sie natürlich die Spezifika der EVIs. Detaillierte Informationen hierzu finden Sie im Kapitel 7.3.5.

Kommen wir aber noch einmal auf die Frage zurück: „Wie viel schneller wird die Abfrage?“ Es ist unerheblich, ob die Abfrage 1.000 Millisekunden oder 2.000 Millisekunden schneller ausgeführt werden kann. Sie optimieren hoffentlich keine SQL Abfragen im Millisekundenbereich! Wichtig ist, dass Sie später, wenn reaktiv auf einzelne Abfragen reagiert werden muss, noch Alternativen bereit stehen. Sollte sich z. B. herausstellen, dass die eben ausgeführte Abfrage strategisch wichtig ist und die Ausführung nicht den Erwartungen entspricht, kann beispielsweise mit einem zusammengesetzten Index für die Spaltenkombinationen reagiert werden. Vielleicht wird dieser Zugriffspfad aber auch nie erforderlich. Warum also jetzt etwas tun, was überflüssig ist und nur unnötig das System belastet?

7.3.6.4 Index Access Only

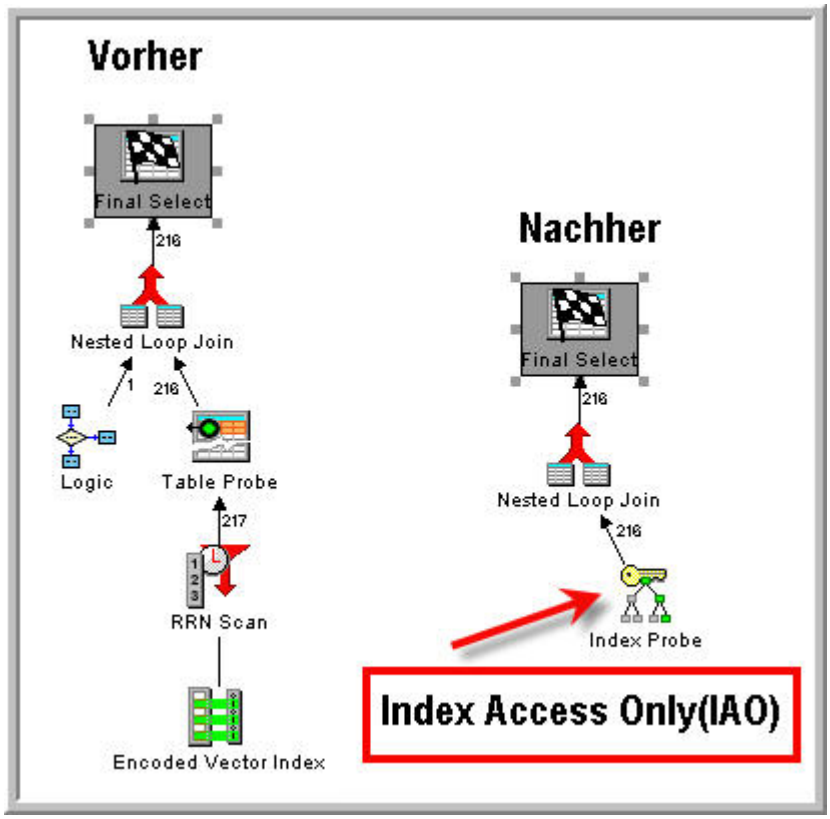
Wenn Sie in einem HLL Programm einen Satz lesen, müssen stets alle Spalten des Satzes in den Hauptspeicher übertragen werden. Hat eine Tabelle viele Spalten, Sie benötigen aber nur die Informationen weniger Spalten, werden viele unnütze Informationen übertragen. Mit einer SQL Anweisung übertragen Sie aber nicht einzelne, sondern viele Sätze in den Hauptspeicher. Ein Faktor für die Performance eines SQL Statements ist daher die Liste der selektierten Felder. Je kleiner die Liste der selektierten Spalten ist, desto kleiner ist die zu verarbeitende Datenmenge im Hauptspeicher und desto schneller ist die Ausführung der SQL Abfrage. Deshalb sollten Sie der Versuchung widerstehen, das Asterix-Zeichen(*) zu verwenden, um alle Spalten einer Tabelle zu selektieren. Stellen Sie sich vor, ich benötige die heute erfassten Rechnungsadressen. Auf dem Bildschirm will ich lediglich den Benutzer und die Adressart anzeigen, der die Daten erfasst hat. Die Abfrage ist schnell geschrieben.

```
SELECT *
FROM Adresse
WHERE AdrInsertDate between '2009-02-20 00:00:00.000000'
and = '2009-02-20 23:59:59.999999'
AND AdressArt = 1;
```

Aber das Problem kann in der Struktur der Tabelle liegen. Die Tabellenzeile verfügt vielleicht über mehr als 100 Spalten. Jeder Datensatz beansprucht mehr als 1 KB Speicherplatz. Bei 15.000 Datensätzen ist das eine Menge von 15 MB Daten, die in den Arbeitsspeicher und später zum Client übertragen werden. Besser Sie konkretisieren, welche Spalten benötigt werden:

```
SELECT adressart, adrInsertUser
FROM Adresse
WHERE AdrInsertDate between '2009-02-20 00:00:00.000000'
and = '2009-02-20 23:59:59.999999'
AND AdressArt = 1;
```

Und es gibt einen weiteren entscheidenden Vorteil: Spezifizieren Sie die Spaltenliste, ist der Optimizer vielleicht in der Lage, nur mit Hilfe eines Index die Abfrage auszuführen. Voraussetzung für die Methode „Index Access Only“ (IAO) ist jedoch, dass der Index alle benötigten Tabellenspalten enthält.



Die Methode „Index Access Only“ (IAO)

Der Zugriff auf den Index ist ausreichend für die Realisierung der SQL Abfrage. Ein Zugriff auf die Tabelle „Adresse“ entfällt. Die Abfrage benötigt in meiner Testumgebung nur die halbe Ausführungszeit. Manchmal muss man sich halt auch die Frage stellen, „Wer soll optimiert werden?“ Der Anwender, der möglichst schnell ein Statement kodiert, oder die SQL Engine, die die Daten bereitstellt?

7.3.6.5 Tabellenverknüpfungen und Indizes

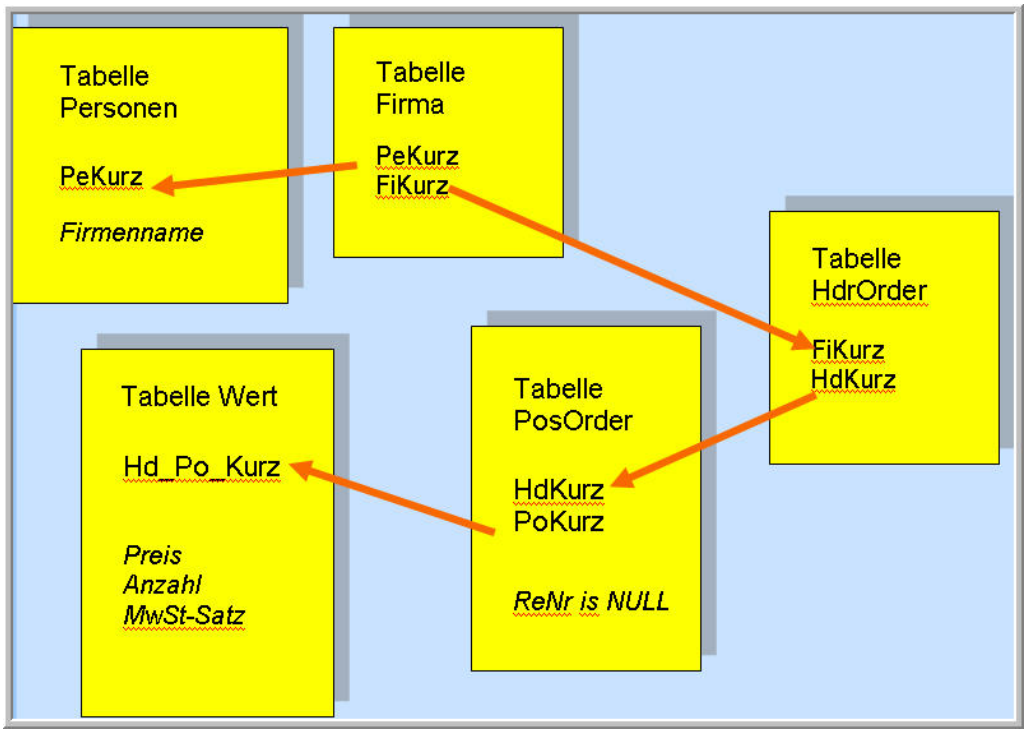
Eher selten verwendet man im SQL Umfeld Abfragen für einzelne Tabellen. Die Regel sind Tabellenverknüpfungen. Nicht selten werden drei, vier oder auch mehr Tabellen miteinander verbunden. Diese komplexen Join-Abfragen benötigen viel Aufmerksamkeit. Sie müssen Ihre Indizes und die Abfragen möglichst geschickt aufbauen und nicht selten kann man so die Laufzeiten von mehreren Minuten oder Stunden auf wenige Sekunden reduzieren.

Schauen wir uns auch hier ein Beispiel an, um zu verstehen, was gemeint ist: In einer weiteren Abfrage selektiere ich alle offenen Auftragspositionen, die im aktuellen Monat fertig gestellt wurden und jetzt fakturiert werden sollen. Folgende Informationen sind gefordert:

ANREDE	NACHNAME		Fertig gestellt am	ANZAHL	BETRAG	NETTOBETRAG	BRUTTOBETRAG
Firma	Müler + Gruber	...	2009-05-10	4	652.00	2608.00	3103.51
Firma	Müler + Gruber	...	2009-05-10	2	600.00	1200.00	1428.00
Firma	Huber GmbH	...	2009-05-18	3	715.00	2145.00	2552.55
Firma	Huber GmbH	...	2009-05-23	3	720.00	2160.00	2570.40

Die Anforderung

Die zugehörige Datenbasis sieht folgendermaßen aus:



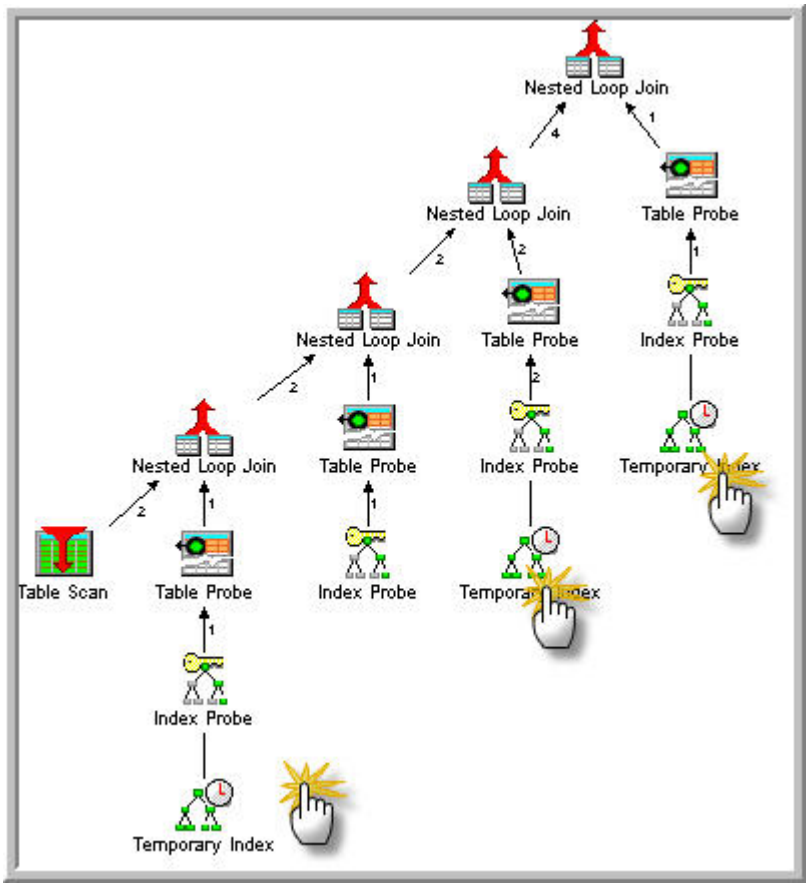
Die Datenbasis

Um die Anforderung aufgrund der vorliegenden Datenbasis zu realisieren, muss ich diese fünf Tabellen miteinander verknüpfen und einige kleinere Berechnungen durchführen. Desweiteren benötige ich nur die Aufträge, die im Mai fertig gestellt (IstDatum) und bisher nicht fakturiert wurden (ReNr is Null). Das zugehörige SQL Statement sieht dann folgendermaßen aus:

```
select
anrede, Nachname, pos.istdatum as "Fertig gestellt am",
anzahl, betrag, betrag* anzahl As "Nettobetrag",
dec(round(betrag * anzahl *(1+mwst),2),11,2) as "Bruttobe-
trag"


from person join firma using(Pekurz)
 join hdrorder using(fikurz)
 join posorder pos Using(hdkurz)
 join wert on pokurz = hd_po_kurz


where pos.istdatum between '01.05.2009' and '31.05.2009' and
ReNr is NULL


order by pos.istdatum;;
```

7.3.6
Seite 24

Die Abfrage wird schleppend ausgeführt. Schauen wir uns den Ausführungsplan an. Er zeigt den Grund für die schleppende Verarbeitung:

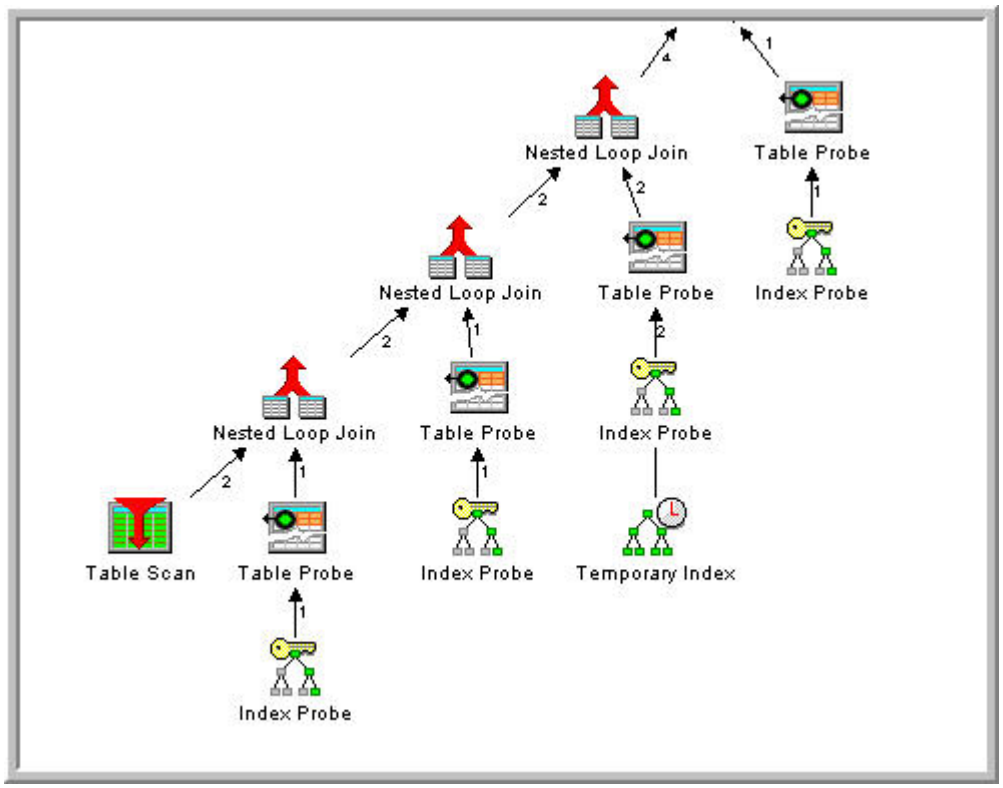


„Join"-Abfrage mit temporären Indizes

Der Optimizer baut für die Verarbeitung verschiedene temporäre Indizes auf, da im Moment in meiner Testumgebung nicht einmal die Minimalanforderungen für „JOIN"-Abfragen erfüllt sind. Sie benötigen mindestens für alle „JOIN"-Spalten Binärindizes!

```
create Index I_PeKurz_Firma on Firma(PeKurz);

create Index I_FiKurz_hdororder on hdrorder(fiKurz);

create Index I_HdKurz_Posorder on posorder(hdkurz);

create Index I_hdpoKurz_wert on wert(hd_po_Kurz);
```

Bauen wir also die Binärindizes für die entsprechenden Spalten auf:



Minimalanforderungen mit Binärindizes

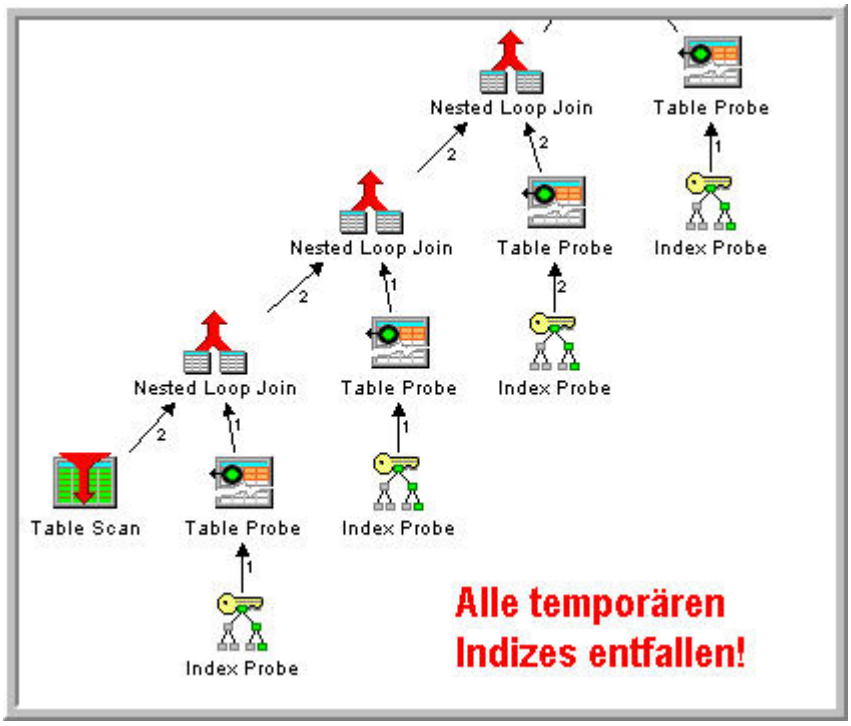
Die erstellten Binärindizes werden sofort verwendet und die Laufzeit der Abfrage halbiert sich. Allerdings wird nach wie vor ein Temporärindex vom Optimizer für die Abfrage verwendet. Warum? Nun, dies möchte ich gern zu einem späteren Zeitpunkt klären.

7.3.6
Seite 26

Zunächst möchte ich mit einem Vorurteil aufräumen: Falls Sie glauben, dass es keine Unterschiede im Laufzeitverhalten zwischen der Verwendung einfacher Binärindizes und Datenbankbedingungen gibt, werden wir einfach einen Vergleich durchführen und unsere zuvor erstellten Binärindizes durch entsprechende referentielle Datenbankbedingungen ersetzen:

```
ALTER TABLE ITPSQL.FIRMA
       ADD CONSTRAINT ITPSQL.Q_ITPSQL_FIRMA_PEKURZ_00001
       FOREIGN KEY( PEKURZ )
       REFERENCES ITPSQL.PERSON ( PEKURZ )
       ON DELETE NO ACTION
       ON UPDATE NO ACTION ;

ALTER TABLE ITPSQL.HDRORDER
       ADD CONSTRAINT ITPSQL.Q_ITPSQL_HDRORDER_FIKURZ_00001
       FOREIGN KEY( FIKURZ )
       REFERENCES ITPSQL.FIRMA ( FIKURZ )
       ON DELETE NO ACTION
       ON UPDATE NO ACTION ;

ALTER TABLE ITPSQL.POSORDER
       ADD CONSTRAINT ITPSQL.Q_ITPSQL_POSORDER_HDKURZ_00001
       FOREIGN KEY( HDKURZ )
       REFERENCES ITPSQL.HDRORDER ( HDKURZ )
       ON DELETE NO ACTION
       ON UPDATE NO ACTION ;

ALTER TABLE ITPSQL.WERT
       ADD CONSTRAINT ITPSQL.Q_ITPSQL_WERT_HD_PO_KURZ_00001
       FOREIGN KEY( HD_PO_KURZ )
       REFERENCES ITPSQL.POSORDER ( POKURZ )
       ON DELETE NO ACTION
       ON UPDATE NO ACTION ;
```

Anschließend führe ich die Abfrage ein weiteres Mal aus. Schauen wir uns nun den Ausführungsplan dieser Abfrage an, wenn referentielle Integritätsbedingungen anstatt der einfachen Binärindizes verwendet werden:



Minimalanforderungen mit referentiellen Integritätsbedingungen

Der Ausführungsplan ändert sich und auch die Ausführungszeiten werden nochmals um ca. 10 % reduziert. Selbst wenn Sie Abfragen vergleichen, deren Ausführungspläne völlig identisch sind, werden Sie feststellen, dass die referentiellen Integritätsbedingungen die bessere Wahl sind. Vielleicht denken Sie doch einmal darüber nach referentielle Integritätsbedingungen in Ihrer Datenbank zu implementieren? Die Minimalanforderungen für Ihre Tabellenverknüpfungen hätten Sie auf diesem Weg zumindest sofort erfüllt!

In meiner Testumgebung sind die Ausführungszeiten dieser Abfrage gut – unabhängig davon, ob ich Binärindizes oder referentielle Integritätsbedingungen einsetze. Wäre ich allerdings nicht zufrieden, könnte ich für die Abfrage zusätzliche Indizes aufbauen:

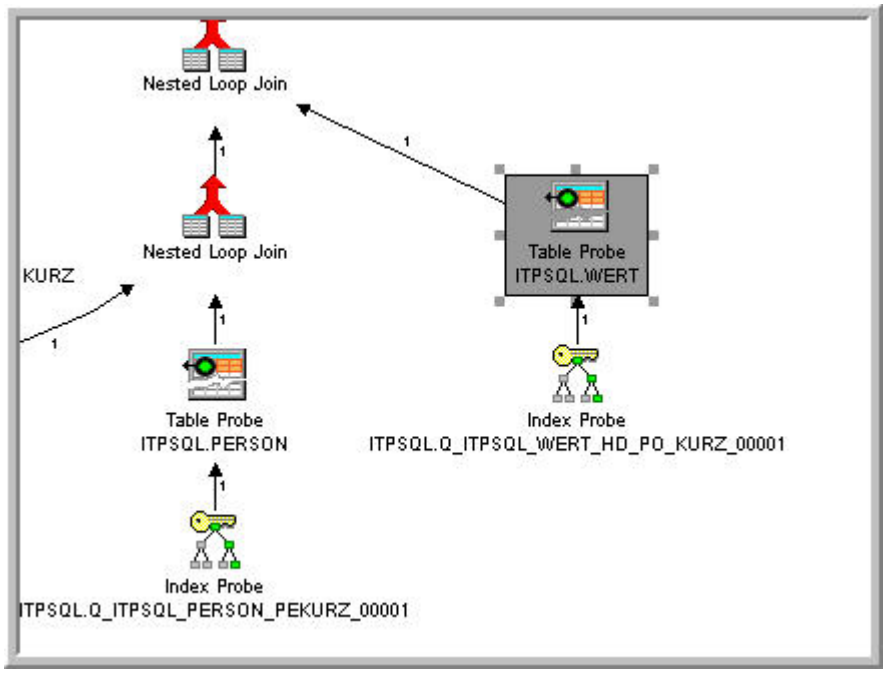
7.3.6
Seite 28

Die Methode „Multi Key Row Positioning V5R4“

Seit Release V5R4 ist die SQL Engine nämlich in der Lage, die Methode „Multi Key Row Positioning“ zu nutzen. Soll heißen, die SQL Engine kann die lokalen Selektionskriterien und die „Join“-Bedingung gleichzeitig auflösen. Zu diesem Zweck werden aber Indizes benötigt, deren führende Schlüsselspalten den lokalen Selektionskriterien entsprechen. Die „Join“-Spalte wird angefügt. Für meine Abfrage bedeutet dies, dass ich zusätzlich zu den referentiellen Integritätsbedingungen einen Index benötige, der die Spalten RENR, ISTDATUM und HDKURZ enthält:

```
CREATE INDEX ITPSQL.I_RENR_ISTDATUM_HDKURZ
ON ITPSQL.POSORDER ( RENR ASC , ISTDATUM ASC , HDKURZ ASC ) ;
```

Tatsächlich nutzt der Optimizer den eben erstellten Index für den Datenzugriff, wie nachfolgende Grafik zeigt:



Die Methode „Multi Key Row Positioning“

Warum entscheidet sich der Optimizer für den Index? Der Vorteil ist in der reduzierten Datenmenge zu sehen, da die lokalen Selektionskriterien jetzt bereits vor dem Datenzugriff geprüft werden können, müssen weniger Daten in den Arbeitsspeicher verlagert werden und die Anzahl der I/O Counts reduziert sich deutlich. Dies war im Übrigen auch der Grund, warum ein Temporärindex weiterverwendet wurde, obwohl entsprechende Binärindizes vorhanden waren.

An dieser Stelle nochmals eine kleine Anmerkung zum Indexadvisor. Der Advisor schlägt für die ausgeführte Abfrage viele Indizes vor, nur der eben erstellte Index fehlt in der Vorschlagsliste!

Vorschläge des Indexadvisors

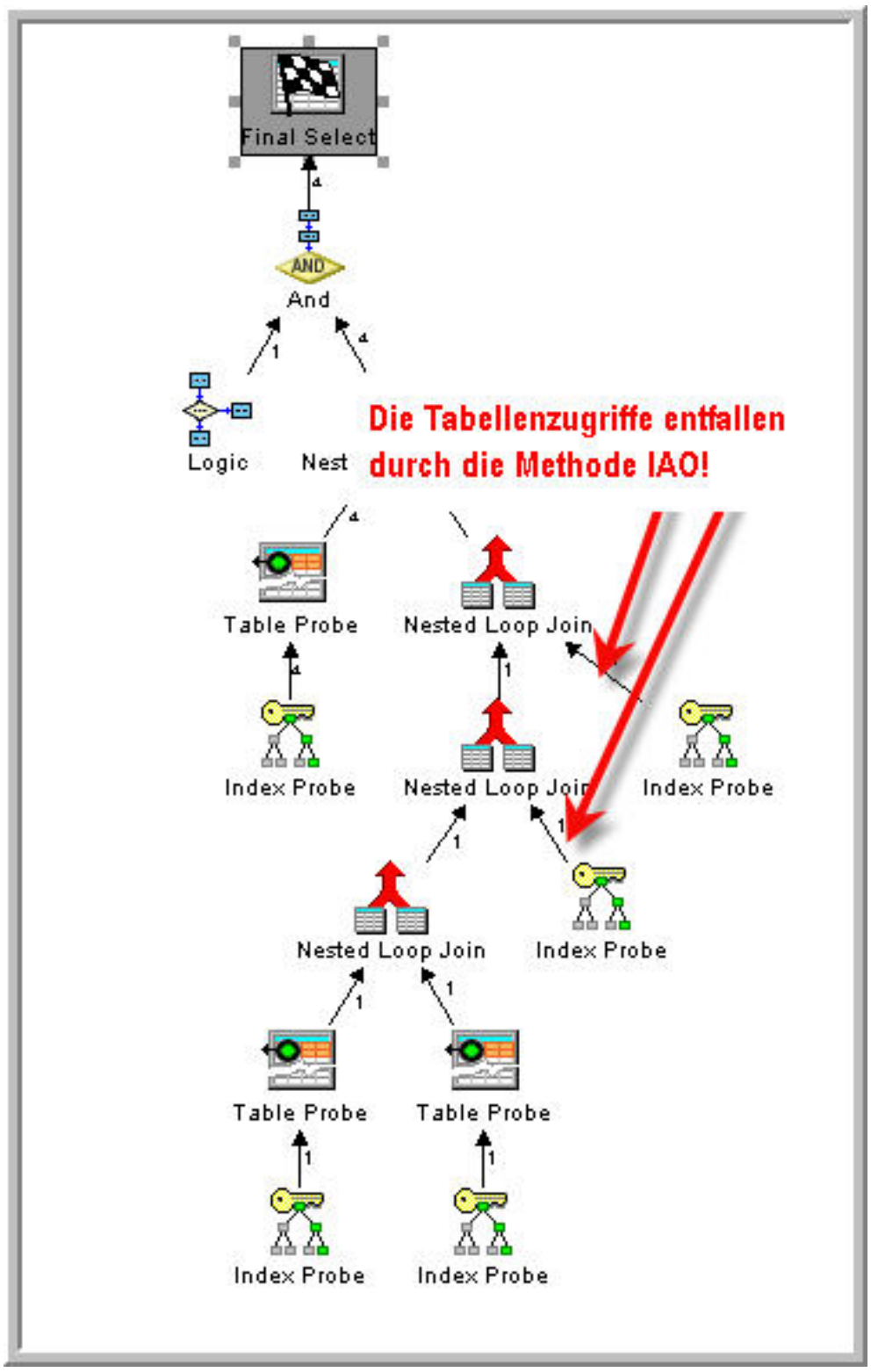
Ich folge eher selten den Empfehlungen des Index Advisors, sondern nutze ihn vielmehr als „Ideenlieferant". Zu häufig passiert es, dass die Vorschläge unbrauchbar oder nicht vollständig durchdacht sind. Auch in diesem Fall gibt es eine weitere Möglichkeit der Optimierung, die der Advisor nicht berücksichtigt. Erinnern Sie sich noch an die Methode „Index Access Only (IAO)" im Kapitel 7.6.3.4?

Sind weitere Optimierungen notwendig, könnte ich für die Tabellen „PERSON" und „WERT" Indizes anlegen, die dem Optimizer die Methode „Index Access Only" erlauben:

```
create index I_Pekurz_Anrede_Nachname
on Person(pekurz, nachname, anrede);

create Index I_hdpoKurz_ANZ_Betrag_Mwst
on wert(hd_po_Kurz, Anzahl, Betrag, Mwst);
```

7.3.6
Seite 30

Der Vorteil dieser Indizes? Der Optimizer kann die lokalen Selektionskriterien prüfen, braucht aber nicht auf die physischen Dateien zuzugreifen, da die Indizes alle notwendigen Spalten enthalten, und so sieht dann der optimierte Ausführungsplan aus:



Die Kombination der Methoden IAO und „Multi Key Row Positioning“

Die Laufzeiten verbessern sich durch den Einsatz der neuen Indizes natürlich erneut, da der Zugriff auf die physischen Tabellen entfällt. Derartige Indizes machen aber selbstverständlich nur dann Sinn, wenn die benötige Spaltenanzahl im ResultSet relativ gering ist und vor allem deutlich kleiner ist als die Spaltenanzahl der Tabelle. Auch müssen Sie abwägen, ob der Geschwindigkeitsvorteil tatsächlich den Aufbau weiterer Indizes rechtfertigt. Bedenken Sie, jeder zusätzliche Index belastet natürlich die Performance Ihrer Insert-, Update- und Delete-Statements. Andererseits sollte dieses Beispiel deutlich gezeigt haben, dass Sie durchaus Möglichkeiten besitzen komplexe „JOIN"-Abfragen deutlich zu beschleunigen – vorausgesetzt man nutzt die richtige Indexstrategie!

7.3.6.6 Gruppierungen und Indizes

Für die Implementierung von Gruppierungen stehen dem Optimizer derzeit generell zwei Implementierungsmethoden zur Verfügung:

- Indeximplementierung
- Hashimplementierung

Die Wahl der Implementierungsmethode ist wieder abhängig von der Datenlage, der Abfrage und den Systemressourcen. Der Optimizer ermittelt zunächst die Anzahl der Gruppen und die Anzahl Sätze je Gruppe.

Entscheidet sich der Optimizer einen Index für die Implementierung zu verwenden, greift der Datenbankmanager über den Index auf die einzelnen Gruppen zu und führt die angeforderten Summenfunktionen aus. Da im Index per Definition bereits alle Schlüsselwerte zusammengefasst sind, kann das erste Gruppenergebnis in kürzerer Zeit als bei der Hashmethode zurückgegeben werden. Wird die Gruppierung mittels Binärindizes implementiert, spricht man von der Methode „Ordered Grouping". Für die Ausführung wird allerdings ein Index benötigt, der alle Gruppierungsspalten enthält.

Entscheidet sich der Optimizer hingegen aufgrund fehlender Indizes für die Hashimplementierung, verwendet er Statistikwerte, um die Größe der Hashtabelle zu bestimmen. Die Anzahl der Gruppen bestimmt die Anzahl der Hashpartitionen. Das Hashverfahren als Zugriffsmethode ist effizient, solange die Hashtabelle gut ausgewogen ist und das Konsolidierungsverhältnis relativ hoch ist. Das Konsolidierungsverhältnis ist das Verhältnis der ausgewählten Tabellenzeilen zu den berechneten Gruppierungsergebnissen. Wenn für jede Zeile der Datenbanktabelle ein eigener, eindeutiger Gruppierungswert vorhanden ist, wird die Hashtabelle zu umfangreich und verlangsamt die Zugriffsmethode. Außerdem dauert die Anzeige erster Ergebnisse in der Regel immer länger als bei vergleichbaren Methoden, da die Tabelle erst vollständig gefüllt werden muss, bevor die Ergebnisse ausgegeben werden.

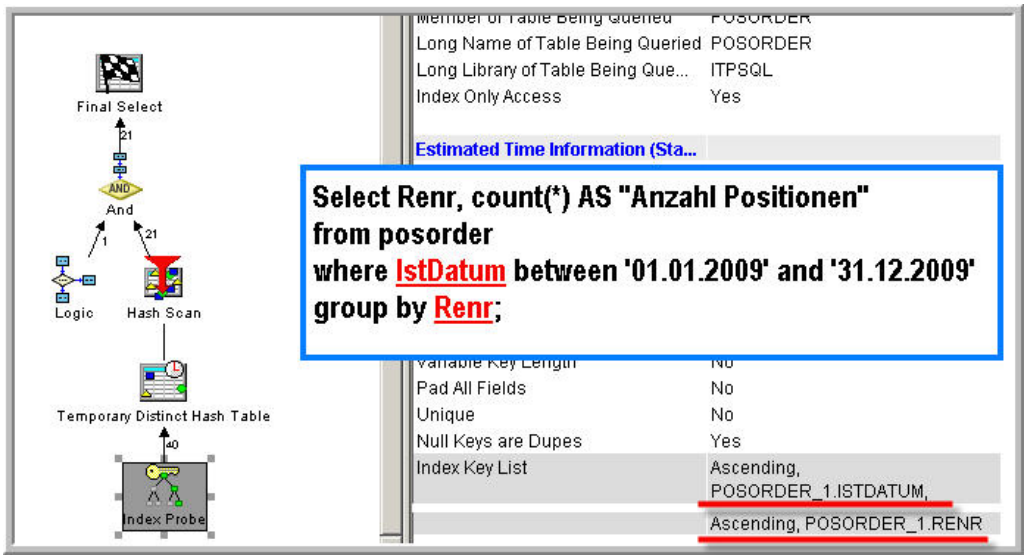
Ergänzung 10/2009

7.3.6
Seite 32

Doch auch für Gruppierungen sind Indizes, die lediglich die Gruppierungsspalten enthalten, häufig nicht ausreichend. Werden die Gruppierungsabfragen komplexer und enthalten komplexe lokale Selektionskriterien, sollten Sie auch hier in Erwägung ziehen, zusammengesetzte Binärindizes zu erstellen. Die lokalen Selektionskriterien sind wiederum den Gruppierungsspalten voranzustellen. Schauen wir uns ein Beispiel an:

```
SELECT RENR, COUNT(*) AS "ANZAHL POSITIONEN"
FROM POSORDER
WHERE ISTDATUM BETWEEN '01.01.2009' AND '31.12.2009'
GROUP BY RENR;
```

Ich möchte mit der Abfrage die Anzahl offener Rechnungspositionen im Geschäftsjahr 2009 ermitteln. Der Optimizer entscheidet sich, den Index „I_Istdatum_ReNr_PosOrder“ zu verwenden und implementiert die Abfrage folgendermaßen:



Binärindex für Gruppen mit lokalen Selektionskriterien

Nach wie vor wird allerdings eine Hashtabelle aufgebaut, um die Gruppierungen zu implementieren. Vielleicht sehen Ihre Ausführungspläne ähnlich aus? Zeigen diese nur anstelle der Methode „Index Probe“ die Methode „Index Scan“? Dann ist dies ein Zeichen, Ihre Indexstrategie zu prüfen. Die Methode „Index Probe“ hat den Vorteil, dass für die Verlagerung der Sätze in den Hauptspeicher deutlich weniger I/O Operationen und weniger Arbeitsspeicher notwendig sind.

Estimated Cost Information Abo	Methode Index Scan
Processing Time(ms)	
I/O Or CPU Bound	I/O Bound
CPU Cost(ms)	1,690
I/O Cost(ms)	6,400
I/O Count	256
PreLoad Relation	No
Memory Used(bytes)	29,938,304
Share of Memory Available(bytes)	29,938,304
Memory Constrained	Yes
Cumulative Memory Constrained	No

Estimated Cost Information Abo	Methode Index Probe
Processing Time(ms)	
I/O Or CPU Bound	I/O Bound
CPU Cost(ms)	175.817
I/O Cost(ms)	1,785
I/O Count	71.408
PreLoad Relation	No
Memory Used(bytes)	5,111,808
Share of Memory Available(bytes)	35,585,200
Memory Constrained	No
Cumulative Memory Constrained	No

Vergleich der notwendigen I/O Operationen

Die Methode „Symbol Table Scan"

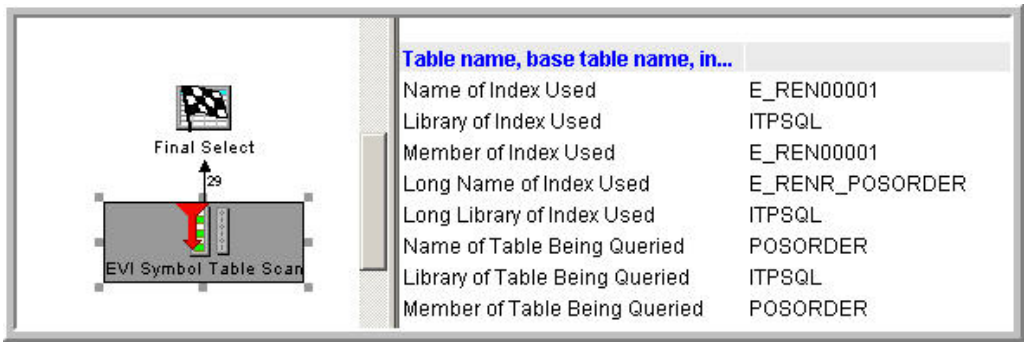
Die SQE kann aber auch Encoded Vector Indizes für Gruppierungen nutzen oder besser gesagt, werden die benötigten Informationen der Symboltabelle entnommen. Die Symboltabelle eines EVIs enthält bereits die Gruppierungsspalten und deren Vorkommen. Der EVI ist somit schon vorgruppiert.

Schauen Sie sich an, was passiert, wenn ich für die Spalte RENR einen Encoded Vector Index erstelle. Die Spalte RENR ist ein guter Vector-Index-Kandidat, da die Werte fortlaufend vergeben werden. Nur muss ich berücksichtigen, dass der Wertebereich noch wachsen wird und ich daher bereits jetzt einen 4-Byte großen Vectorindex erstellen sollte.

```
CREATE ENCODED VECTOR INDEX E_RENR_POSORDER
ON POSORDER(RENR) WITH 1000000 DISTINCT VALUES;
```

7.3.6
Seite 34

Welche Auswirkungen der neue Vector-Index auf die Abfrage hat, zeigt die folgende Grafik:

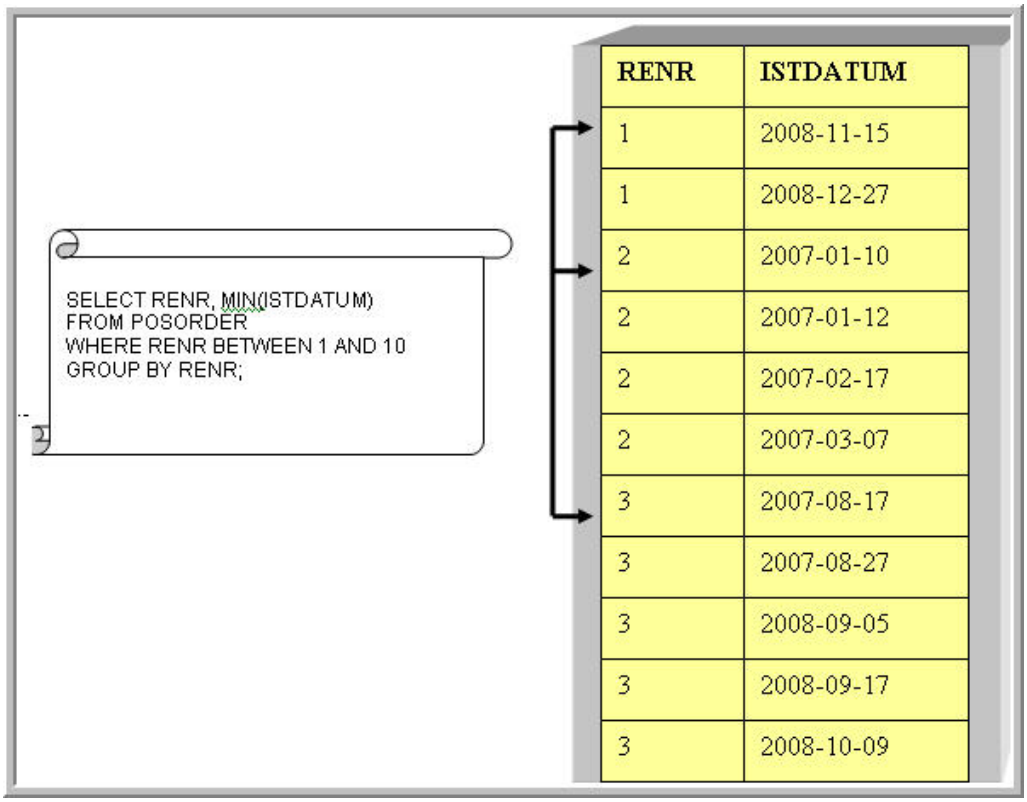


Die Methode „EVI-Symbol Table Scan“

Die verwendete Methode nennt man „Symbol Table Scan“ oder „Presummarized processing“. Sie ist für einzelne Tabellenabfragen geeignet, die einfache Aggregate mit der Funktion COUNT() bilden. Die Laufzeiten solcher Abfragen können sich durch EVIs drastisch verkürzen. Aber auch wenn lokale Selektionsbedingungen vorhanden sind, kann die SQL Engine den EVI für die Ausführung nutzen. Es muss in diesen Fällen aber zusätzlich eine temporäre Liste mit relativen Satznummern für den Zugriff aufgebaut werden.

Die Methode „Early Exit“
In den Gruppierungen wird aber nicht nur die Funktion COUNT(*) verwendet, häufig soll auch der kleinste oder der größte Wert einer Spalte ermittelt werden. Bestehende EVIs und einfache Binärindizes helfen an dieser Stelle oftmals nicht weiter und liefern häufig die Methode „Table Scan“ oder veranlassen die SQE, temporäre Indizes zu erstellen.

Überlegen Sie sich einfach, welche Sortierung benötigt wird, um möglichst schnell den größten und den kleinsten Wert einer Gruppe zu ermitteln!



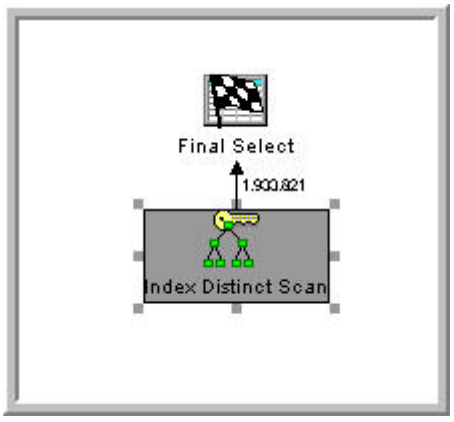
RENR	ISTDATUM
1	2008-11-15
1	2008-12-27
2	2007-01-10
2	2007-01-12
2	2007-02-17
2	2007-03-07
3	2007-08-17
3	2007-08-27
3	2008-09-05
3	2008-09-17
3	2008-10-09

Sortierfolge der Methode „Early Exit“

Wenn die Abfrage ausgeführt wird, muss ein Index genutzt werden, um ihn auf den ersten Satz einer Gruppe zu positionieren. Entspricht der Spaltenwert dem lokalen Selektionskriterium, wird die Zeile selektiert. Die Verarbeitung der Gruppe ist beendet. Es folgt ein Sprung zur nächsten Gruppe. Dieser Prozess wird fortgesetzt bis alle Gruppen geprüft wurden. Ich benötige also einen Index für die Spalten RENR und ISTDATUM. Dieses Mal muss allerdings das lokale Selektionskriterium nachgestellt werden:

```
CREATE INDEX I_RENR_ISTDATUM_POSORDER ON POSORDER(RENR,
ISTDATUM);
```

Und so sieht die perfekte Ausführung aus:



Die Methode „Early Exit“

7.3.6
Seite 36

Der Ausführungsplan ist deutlich reduziert. Es werden keine temporären Objekte erstellt und auch auf die Methode „Table Scan“ kann der Optimizer verzichten. Die Ausführungszeit beträgt nur noch ein Bruchteil dessen, was zuvor an Zeit benötigt wurde.

Die Methode „Early Exit“ kann der Optimizer allerdings nur einsetzen, wenn

- einzelne Tabellen abgefragt werden und
- keine anderen Spaltenfunktionen außer MIN() oder MAX() verwendet werden

Falls Sie jetzt glauben für die MAX() Funktion absteigend sortierte Indizes bereitstellen zu müssen, irren Sie sich. Seit Release V5R4 kann die SQE Indizes rückwärts lesen, so dass es nicht erforderlich ist zusätzliche Indizes für die MAX() Funktion zu erstellen.

7.3.6.7 Sortierungen und Indizes

Die Betrachtung der Klausel ORDER BY für die Erstellung geeigneter Indizes muss in den meisten Fällen nachgelagert werden, denn vorrangig werden Indizes benötigt, um Daten zu selektieren, zu verknüpfen und zu gruppieren. Diese Betrachtung ist für viele HLL Programmierer zunächst befremdlich, brauchten wir doch in unseren RPG Programmen die logischen DDS Dateien, um die Daten in sortierter Folge zu lesen und weit weniger oft für die Auswahl der Sätze. In einem HLL Programm habe ich aber auch immer nur einen Satz zur Verarbeitung im Speicher. Wenn Sie nicht dafür sorgen, dass die SQL Engine Daten von der Verarbeitung ausschließen kann, werden vielleicht 1.000.000 Sätze in den Arbeitsspeicher verlagert.

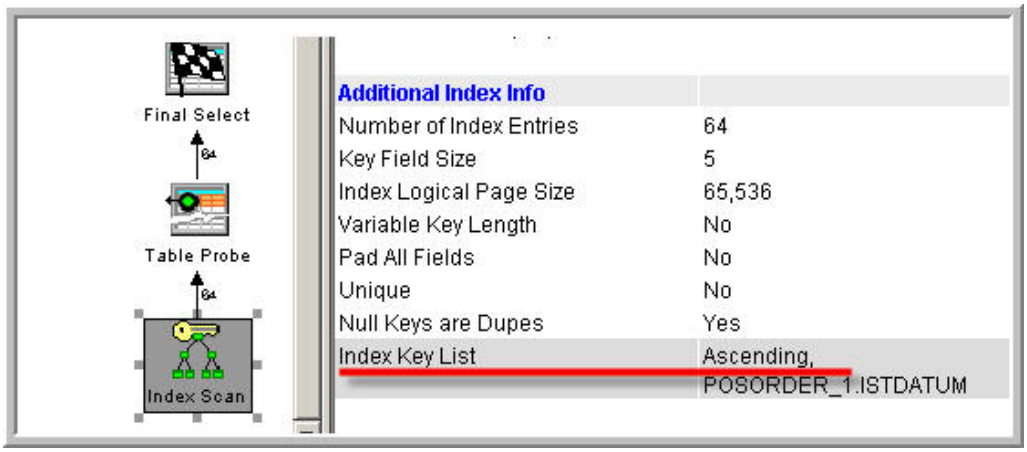
SQL kann die Methode „Index Ordering“ für die Implementierung der Sortierung oder die Methode „Sort Ordering“ einsetzen. Hierfür werden die Daten in einen Cache Bereich verlagert, sortiert und anschließend als Ergebnis angezeigt. Die Methode „Index Ordering“ benötigt natürlich – wie der Name schon sagt – einen Index. Die führenden Schlüsselspalten des Index müssen die Sortierspalten vollständig abdecken. So kann die SQL Engine die benötigten Daten bereits in sortierter Folge in den Arbeitsspeicher verlagern und in das Result Set übernehmen, ohne weitere Sortieralgorithmen implementieren zu müssen. Diese Strategie kann vorteilhaft sein, wenn für die Abfrage nicht alle Daten benötigt werden oder wenn bereits ein Index existiert, der die Anforderung realisieren könnte. Für unsere Indexstrategie bedeutet das, dass Sie auch hier wieder Indizes erstellen können, die eine Kombination der lokalen Selektionskriterien gefolgt von den Sortierspalten darstellen:

- Lokale Selektionskriterien und „Order By“-Spalten

Sie sollten aber hierauf nicht so viel Aufmerksamkeit verwenden. Viele Indizes, die für die Sortierung genutzt werden, ergeben sich durch die Untersuchung der „Where"- „Join"- und „Group By"-Bedingungen und wenn nicht, werden Sie häufig feststellen müssen, dass die Engine eigens für die Sortierung erstellte Indizes nicht nutzt, weil sie vorrangig das Ziel verfolgt Daten von der Verarbeitung aus- bzw. einzuschließen. Eines ist in diesem Zusammenhang noch wichtig zu wissen, die SQL Engine kann seit Release V5R4 Indizes rückwärts lesen. Das heißt, ein aufsteigend sortierter Index kann für absteigende Sortierfolgen verwendet werden. Sie benötigen also keine zusätzlichen Indizes, um folgende Abfrage auszuführen:

```
select *
from posorder
order by Istdatum desc;
```

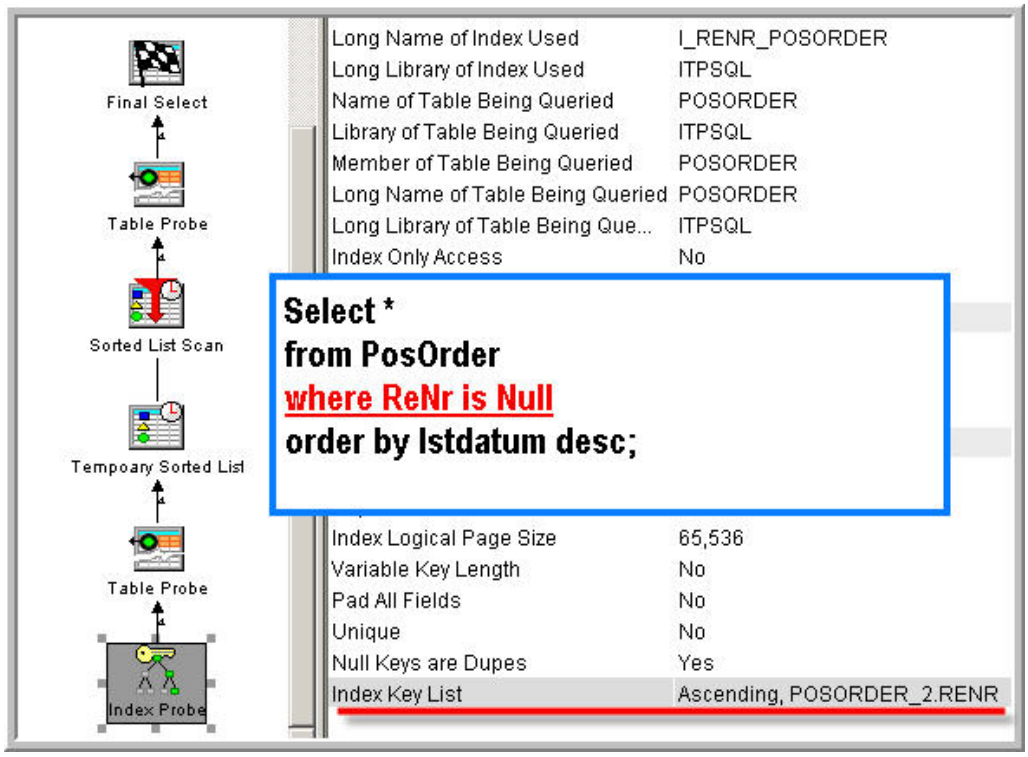
Der Ausführungsplan zeigt deutlich, dass ein aufsteigend sortierter Index für die Abfrage genutzt wird:



Aufsteigende Indizes für absteigende Sortierfolgen

7.3.6
Seite 38

Ändere ich die Abfrage, indem ich lokale Selektionskriterien hinzufüge, ändert sich sofort der Ausführungsplan:

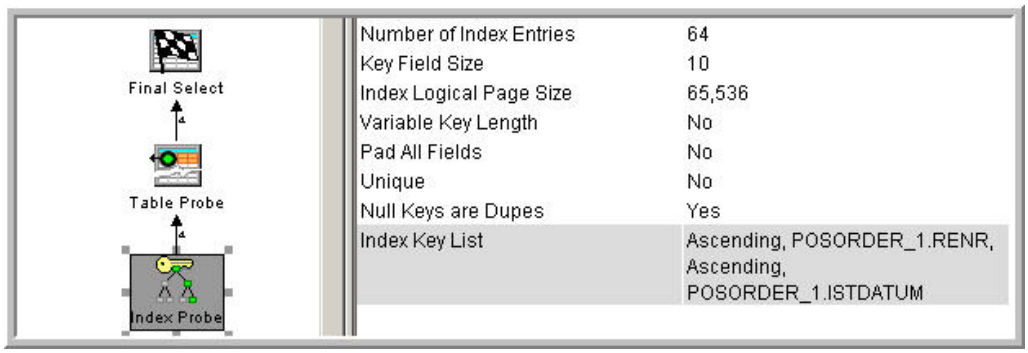


Selektive Selektionskriterien haben Vorrang

Das Beispiel zeigt deutlich, dass der Optimizer Indizes für den Datenausschluss bevorzugt, wenn die Ergebnisse möglichst schnell zur Verfügung stehen müssen. In diesen Fällen können Sie wiederum nur versuchen mit einem geeigneten Index Selektionskriterien und Sortierspalten zu verbinden:

```
CREATE INDEX I_RENR_ISTDATUM_POSORDER ON POSORDER(RENR,
ISTDATUM);
```

Der neue Ausführungsplan ist natürlich deutlich besser, da die temporären Sortieralgorithmen entfallen:



Der optimierte Ausführungsplan

Je komplexer Ihre Abfragen allerdings werden, desto unwahrscheinlicher wird die Verwendung dieses Index. Der Optimizer hat bislang keine Möglichkeiten Gruppierung, Verknüpfungen und Sortierungen in einem Arbeitsschritt auszuführen. Er muss sich stets entscheiden, welche Aufgabe er als erstes erfüllen will.

7.3.6.8 Warum werden meine Indizes ignoriert?

Auch wenn Sie Ihre Indizes sehr sorgfältig planen, werden Sie von Zeit zu Zeit feststellen können, dass der Optimizer in einzelnen Fällen keinen Index für die Ausführung der SQL-Abfragen nutzt. Dies kann verschiedene Gründe haben:

„Table Scan Cost is better"
Es gibt einfach Situationen, in denen Indizes von der Engine nicht verwendet werden, weil die Kosten für den Einsatz zu hoch sind. Schauen Sie sich das folgende Beispiel an:

```
CREATE INDEX I_INFINSUSR ON INFO(INFINSUSR);

SELECT FI_AP_KURZ , INFO FROM INFO
WHERE INFOINSERTUSER = 'RASCHE';
```

Jeder, der diese Abfrage liest, würde vermuten, dass der Optimizer den erstellten Index „I_INFINSUSR" zur Ausführung der Abfrage verwendet. Doch der Optimizer ignoriert den Index, wie nachfolgender Ausführungsplan zeigt:

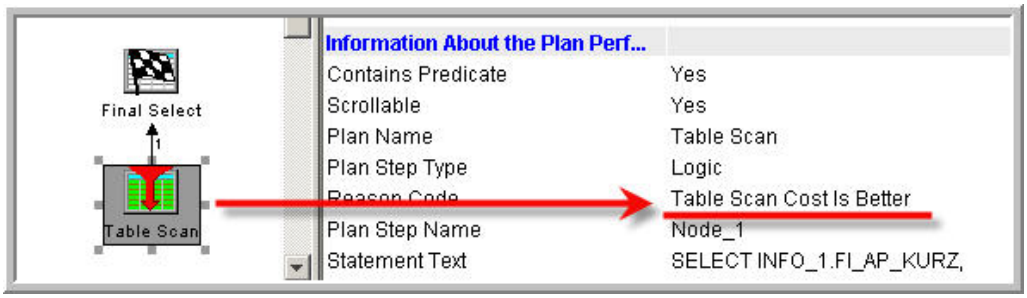
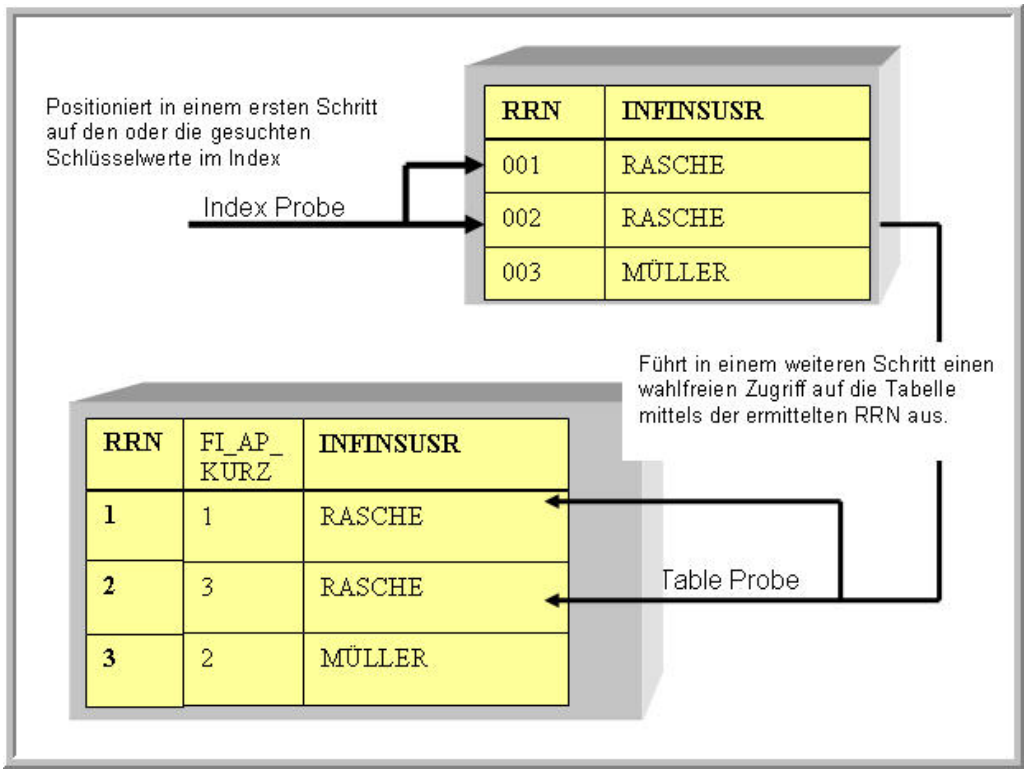


Table Scan Cost is Better

Der Grund? In der Tabelle „INFO" finden wir nur wenige Einträge. Durch die Verwendung des Index wäre der Optimizer gezwungen zunächst die Methode „Table Probe" zu verwenden. Der Index würde durchsucht werden und erst in einem zweiten Schritt könnte auf die Daten der Tabelle zugegriffen werden.

7.3.6
Seite 40



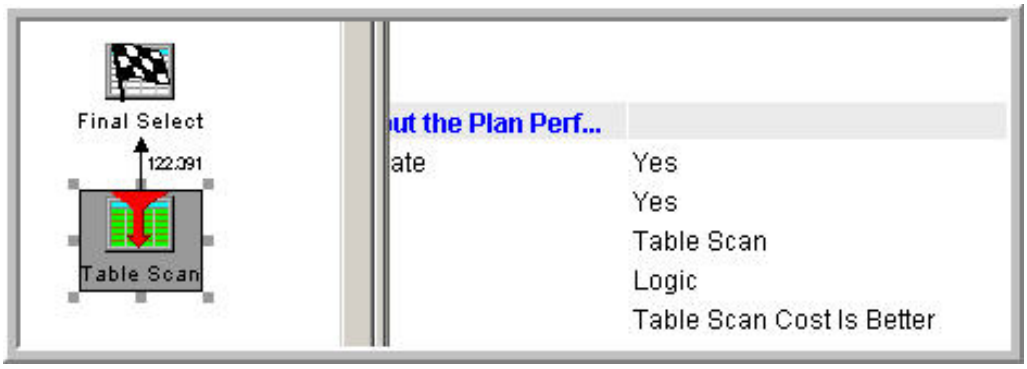
Die Methoden Index Probe und Table Probe

Zu viel Aufwand, um nur wenige Zeilen zu verarbeiten. Die einzige Möglichkeit könnte darin bestehen die Methode „Index Access Only“ zu ermöglichen, damit der Zugriff auf den Index ausreichend ist.

Aber es gibt weitere Situationen, die einen Indexeinsatz verhindern. Auch hierzu ein einfaches Beispiel:

```
SELECT *
FROM INFO
WHERE INFINSDAT > INFUPDDAT;
```

In der Tabelle wird das Datum der Erfassung und das Datum der letzten Zeilenveränderung gespeichert. Selektiert werden sollen alle Zeilen, deren Erfassungsdatum hinter dem Änderungsdatum liegt, um ggf. fehlerhafte Daten zu korrigieren. Dieses Beispiel steht für eine Reihe von SQL Statements, die den Einsatz eines Index verhindern. Das Problem dieser Abfrage besteht darin, dass in den lokalen Selektionskriterien zwei Spalten einer Tabellenzeile miteinander verglichen werden. In der Abfrage wird die SQL-Engine daher keinen Index einsetzen können, selbst wenn sowohl ein Index für „INFINSDAT“ als auch ein Index für „INFUPDDAT“ existieren würde. Um diese Abfrage aufzulösen, muss jeder einzelne Satz geprüft werden. Ein Zugriff über Index bringt hier keine Vorteile.



„*Table Scan Cost is Better*"

Unvollständige SQL Abfragen

Aber auch unvollständig formulierte SQL Abfragen können den Einsatz eines Index verhindern. Die SQL-Engine wird hart daran arbeiten, existierende Indizes zu finden, die die Durchführung jedes SQL-Statements vereinfachen. Dieser Prozess kann aber durch einige gängige Praktiken behindert werden. Manchmal werden beim Datenbankentwurf z. B. Vorkehrungen getroffen, um sich für die Zukunft zu wappnen. Gängige Praktik ist es, in die Datenbank eine Spalte für die Werks- und/oder Firmennummer einzufügen und diese bei den meisten Indizes als führende Schlüsselspalte zu verwenden:

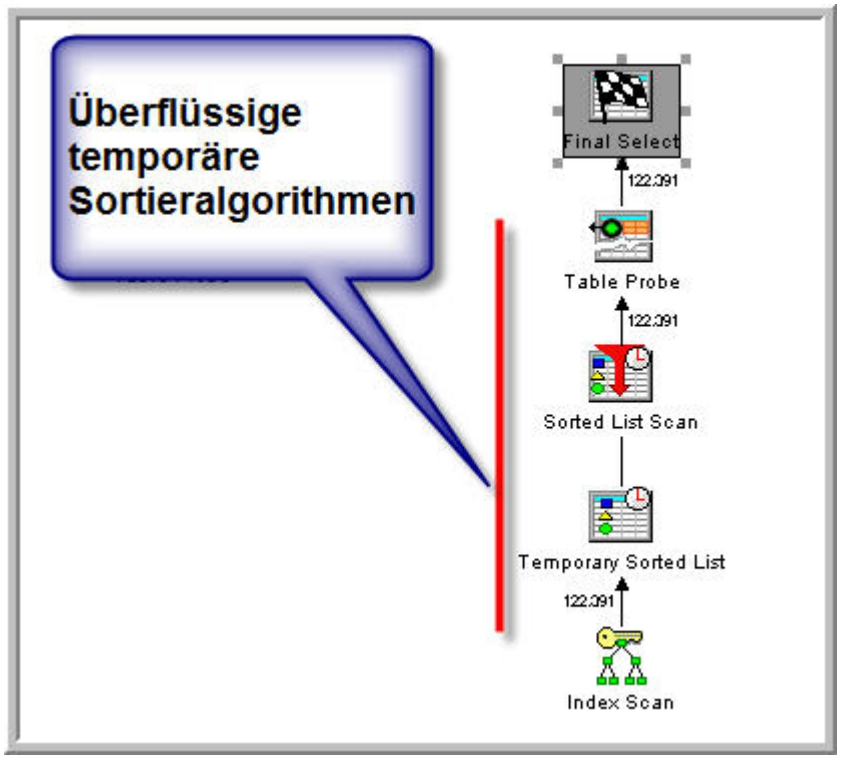
```
CREATE INDEX I_01 ON KDST99(KDFINR, KDKDNR);
```

Abgesehen davon, dass es bislang nur ein Unternehmen gibt, ist gegen diese Methode nichts einzuwenden. In der Datenbank hat jeder Kundendatensatz eine „00" als Unternehmensziffer. Schlaue Anwender wissen aber, dass die Firmennummer ohne Bedeutung ist und eliminieren sie aus ihrem SQL-Statement.

```
SELECT KDFINR, KDKDNR, KDNAM1
FROM KDST99
ORDER BY KDKDNR;
```

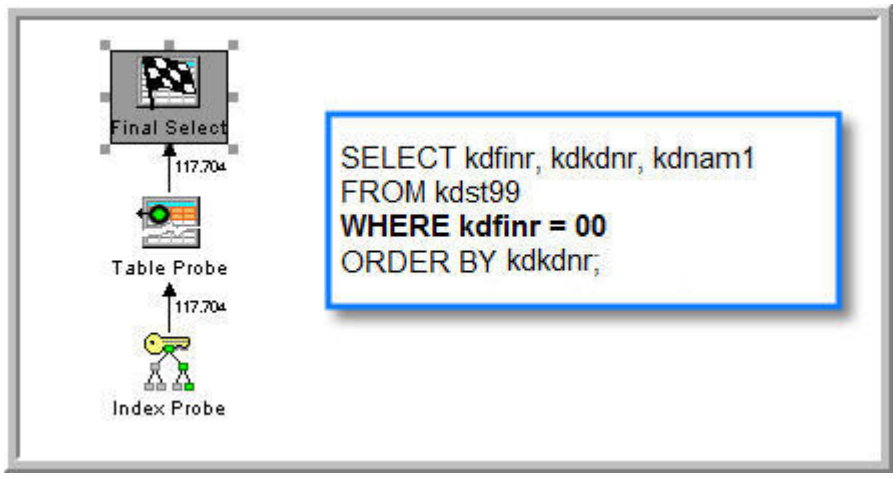
7.3.6
Seite 42

Für diese Abfrage ist der zuvor erstellte Index nur eingeschränkt nutzbar. Die SQL Engine weiß nicht, dass die Sortierfolge KDFINR, KDKDNR gleich bedeutend ist mit der Sortierfolge KDKDNR und wird im Arbeitsspeicher temporäre Objekte nutzen, um die Sortierfolge „KDKDNR“ zu realisieren:



Überflüssige Sortieralgorithmen

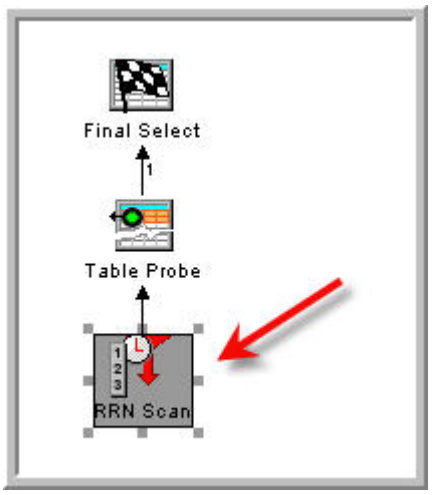
Würde die Unternehmensziffer in der „ORDER BY“-Klausel aufgeführt oder würde eine WHERE-Klausel verwendet werden, um nur das Unternehmen „00” zu selektieren, wüsste die SQL-Engine, wie der existierende Index zu verwenden ist.



Die korrigierte Abfrage

Vermeiden Sie relative Satznummern

Für die meisten Tabellen ist es sehr einfach eindeutige Schlüsselspalten zu erstellen, aber Transaktionstabellen haben oftmals keine eindeutigen Spaltenkombinationen, die sich als Index eignen. In einem HLL Programm spielt das auch keine Rolle, da wir in diesen Fällen einfach über die relative Satznummer (RSN) auf die Sätze der Tabelle zugreifen. Auch SQL ermöglicht die Nutzung der relativen Satznummer, wenn Sie die SQL Funktion RRN() verwenden. Die relative Satznummer ist aber leider keine Spalte der Tabelle, so dass Sie hierfür keinen Index erstellen können. Wenn Sie nun in Ihrer SQL Abfrage eine Zeile anhand der RSN lesen, nutzt der Optimizer die Methode „Relative Record Number Scan".



Die Methode „RRN Scan"

Diese Methode ist vergleichbar mit einem „Table Scan" und ähnlich langsam in der Ausführung. SQL bietet aber die Möglichkeit autogenerierte Schlüsselfelder mittels Identity Columns (Identifikationsspalten) oder anderer Objekte zu definieren. Anstelle eines Programms liefert jetzt die Datenbank eindeutige Schlüsselwerte. Mittlerweile kennt die DB2 for i5/OS viele Möglichkeiten, Identifikationsspalten zu erzeugen. Sie können den Datentypen ROWID verwenden oder die Klausel GENERATED ALWAYS AS IDENTITY einsetzen.

ROWID-Spalten werden verwendet, um eindeutige Schlüsselwerte für jede Zeile in einer Tabelle zur Verfügung zu stellen. Sobald eine neue Zeile in die Tabelle eingefügt wird, erstellt die Datenbank einen gültigen Wert, der innerhalb der Tabelle eindeutig ist. Die Verwendung des Datentyps ist einfach:

```
CREATE TABLE TransAkt (
                ID ROWID GENERATED ALWAYS,
                …
                )
```

Sie sehen, dass ich die Klausel GENERATED ALWAYS für die ROWID-Spalte verwende. GENERATED ALWAYS ist nicht ausschließlich für ROWID-Spalten zu nutzen, sondern wird generell für IDENTITY-Spalten benutzt, um der Datenbank mitzuteilen, dass die Werte dieser Spalte ausschließlich von der Datenbank generiert werden. Das bedeutet im Umkehrschluss, dass Sie – auch nicht mit Hilfe eines Programms – die Werte der Spalte ID manipulieren können. Die Implementierung der ROWID-Spalten ist zwar sehr einfach, allerdings werden die Werte als 40-Byte großes Bit-Muster gespeichert, was die Abfrage deutlich erschwert.

Aus diesem Grund ist der Datentyp ROWID meiner Meinung nach nicht so nützlich wie die Klausel „GENERATED ALWAYS AS IDENTITY“. Für die Deklaration benötigen wir zwar etwas mehr Angaben, aber dafür werden die Spalteninhalte als deutlich lesbarer und einfach generierter Wert gespeichert.

```
CREATE TABLE TransAkt
(         ID SMALLINT NOT NULL
          GENERATED ALWAYS AS IDENTITY
          (START WITH 100 INCREMENT BY 1
          NO CYCLE),
          …
)
```

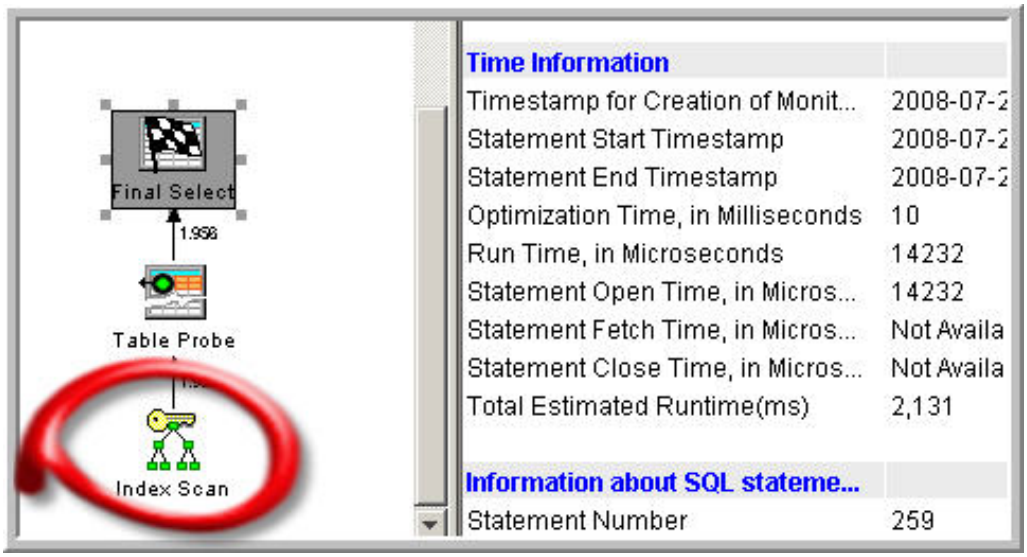
In diesem Beispiel verwende ich für die Definition der Spalte ID die Klausel GENERATED ALWAYS AS IDENTITY. Insgesamt darf nur eine Spalte innerhalb der Tabelle als IDENTITY-Spalte deklariert werden. Für bestehende Spalten können Sie das Attribut nicht nachträglich implementieren, da bereits Werte erfasst wurden. Der Vorteil der IDENTITY-Spalten besteht darin, dass Sie sowohl den Startwert als auch das Inkrement festlegen können. In meinem Beispiel wird die Spalte ID den Wertebereich 100 bis 32767 verwenden, da der Maximumwert in diesem Fall durch den Spaltentypen SMALLINT bestimmt wird. Aufgrund des Schlüsselwortes NO CYCLE wird der Startwert 100 nicht wieder verwendet, sobald der Maximalwert erreicht ist. Wird der Maximalwert erreicht, können keine weiteren Werte erfasst werden, da die zyklische Vergabe der Werte durch das Schlüsselwort NO CYCLE unterbunden ist. Weiterführende Informationen zu diesem Thema finden Sie im Kapitel 7.4.4.6. Sie sollten also auch bei Transaktionstabellen darauf achten entsprechende Schlüsselspalten zu generieren. Verwenden Sie in diesen Fällen doch einfach künstliche Schlüssel, wenn keine geeigneten Schlüsselspalten zur Verfügung stehen.

Skalare SQL Funktionen im Release V5R4

Ein weiterer Fehler ist es, vom Optimizer zu erwarten, dass er einen Index verwendet, wenn die lokalen Selektionskriterien Skalarfunktionen enthalten:

```
SELECT ISTDATUM, TEXT, RENR, REDAT
FROM POSORDER
WHERE YEAR(ISTDATUM) = '2009';
```

Obwohl ein Index für die Spalte „ISTDATUM“ vorhanden ist, wird der Optimizer nicht in der Lage sein, diesen Index zu verwenden. Hinderungsgrund ist die Funktion YEAR(). Zunächst muss die Funktion aufgelöst werden und erst in einem zweiten Schritt kann der Vergleich mit der Konstanten „2009“ ausgeführt werden. Der vorhandene Index „I_ISTDATUM“ wird nur verwendet, um die Tabelle seitenweise zu lesen ohne Prüfung der lokalen Selektionskriterien.

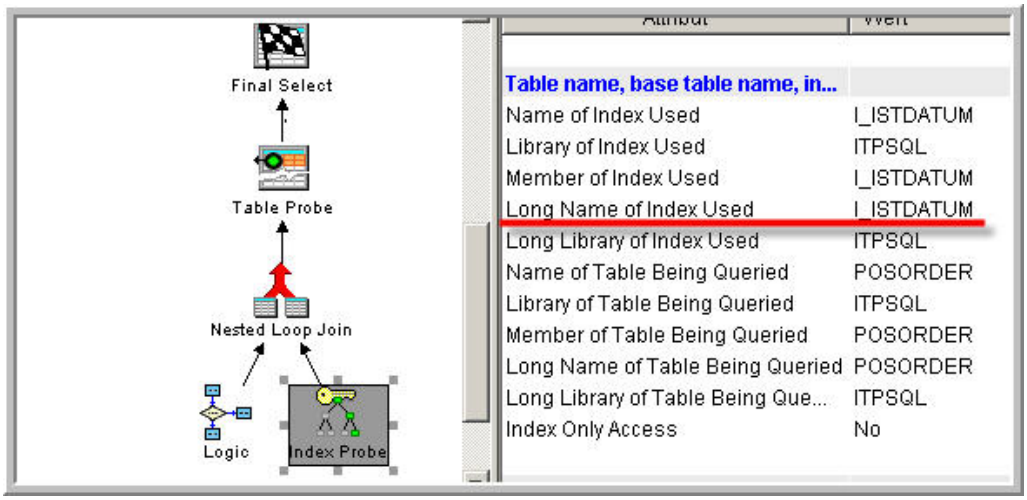


Funktionen in lokalen Selektionskriterien

Auch hier taucht wieder die Frage auf: „Wer soll optimiert werden?“ Der Programmierer, der mit möglichst kurzen Statements und wenig Schreibaufwand zum Erfolg kommen will? Oder die SQL Engine, die die Ausführung der Abfrage durch Reduzierung der I/O Kosten beschleunigen kann? Formulieren Sie die Abfrage besser folgendermaßen:

```
SELECT ISTDATUM, TEXT, RENR, REDAT
FROM POSORDER
WHERE ISTDATUM BETWEEN '01.01.2009' AND '31.12.2009';
```

7.3.6
Seite 46



Die korrigierte Abfrage

Der Schreibaufwand mag sich erhöhen, aber für die Ausführung benötigt die SQL Engine weniger Zeit.

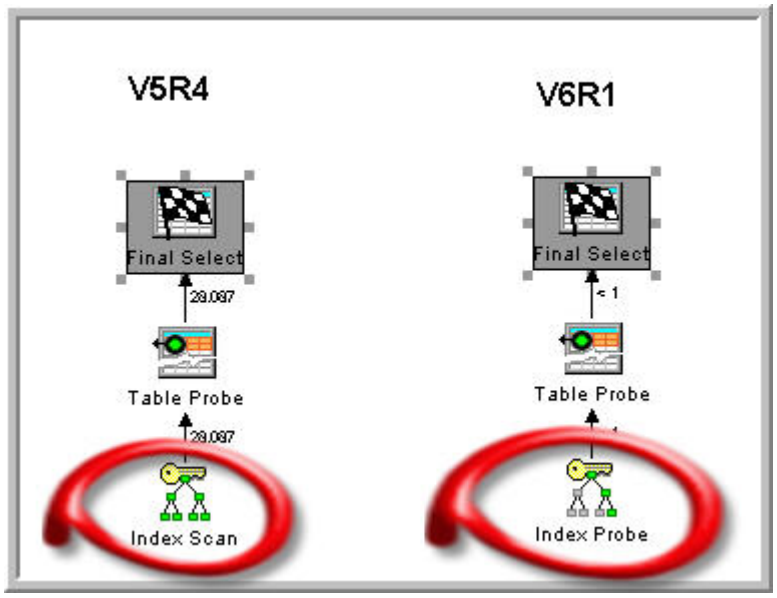
Es gibt allerdings Situationen, da lassen sich SQL Funktionen nicht vermeiden. Gern werden z. B. innerhalb eines Spaltenwertes Informationen verschlüsselt. Artikelnummern enthalten beispielsweise häufig Farbcodes oder Größenbezeichnungen. Um derartige Informationen zu selektieren, müssen Sie die SQL Funktion „SUBSTR“ verwenden:

```
SELECT PRDNAM, SUBSTR(PRDTYP, 3, 2) as PRDSIZE
FROM PRDKT00
WHERE SUBSTR(PRDTYP, 3, 2) = '42';
```

Solange Sie mit Release V5R4 arbeiten, kann der Optimizer keinen Index verwenden. Die Lösung kommt aber im Release V6R1. Ein neues Feature im Release V6R1 sind abgeleitete Indizes:

```
CREATE INDEX I_PrdType on Prdkt00(substr(PRDTYP,3,2));
```

Die abgeleiteten Indizes ermöglichen den Datenzugriff unter Berücksichtigung derartiger Selektionskriterien:



Abgeleitete Indizes V6R1

Identisch verhält es sich mit den Funktionen UPPER und LOWER. Hier kommt hinzu, dass diese Funktionen im Release V5R4 noch von der CQE ausgeführt werden. Weiterführende Informationen zu abgeleiteten Indizes enthält das Kapitel 7.3.5.6. Falls Sie also nicht in absehbarer Zeit auf das Release V6R1 wechseln, sollten Sie in Erwägung ziehen, diese Funktionen im HLL Programm zu realisieren, anstatt in die SQL Abfrage zu integrieren.

„LIKE"-Abfragen Release V5R4
Das Prozentzeichen und der Unterstrich kann in „LIKE"-Abfragen verwendet werden, um einzelne Zeichen zu maskieren. Bis Release V5R4 führte die CQE derartige Abfragen aus und nutzte hierbei nur einen Index, wenn die Wildcards in der Mitte oder am Ende des vergleichenden Strings eingesetzt wurden.

Seit Release V5R4 kann die SQE diese Abfragen ausführen und die SQE kann Indizes selbst dann verwenden, wenn führende Wildcards verwendet werden. Schauen Sie sich das folgende Beispiel an:

```
create Index I_Text_posorder on PosOrder(Text);

SELECT ISTDATUM, TEXT
FROM POSORDER
WHERE TEXT LIKE '%SQL%';
```

7.3.6
Seite 48

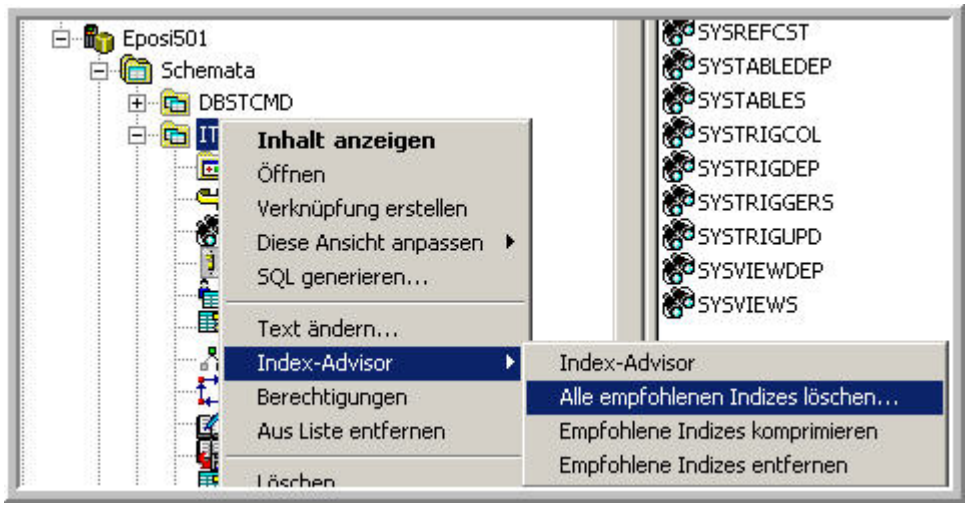
Führende Wildcards in „LIKE"-Abfragen SQE

Andererseits müssen Sie aber darauf achten, dass die CQE über diese Fähigkeiten nicht verfügt. Wenn nun – aus den verschiedensten Gründen – die CQE die Optimierung und Ausführung des SQL Statements übernehmen muss, sollten Sie auf führende Wildcards verzichten, damit Indizes für die Ausführung eingesetzt werden können.

7.3.6.9 Überprüfung der Indexstrategie

Wenn Sie Ihre Indizes erstellt haben, sollten Sie anschließend die Applikation oder das Programm testen. Zuvor löschen Sie aber alle derzeit vorhandenen Indexvorschläge für das Schema oder für einzelne Tabellen. In den meisten Fällen sind die vorhandenen Empfehlungen jetzt wenig hilfreich. Nicht selten finden Sie hunderte von Vorschlägen. Manche sind veraltet und manche unbrauchbar, weil Abfragen nur zu Testzwecken erstellt wurden.

Sie können hierfür System i Navigator nutzen. Öffnen Sie die Datenbank und markieren Sie mit der rechten Maus das entsprechende Schema. Im Kontextmenü wählen Sie den Eintrag „Index-Advisor" ➜ „Alle empfohlenen Indizes löschen".



Alle empfohlenen Indizes löschen

Während der Erstellungsphase Ihrer Indizes haben Sie viele Abfragen zu Testzwecken ausgeführt, die die Nutzungszähler der Indizes beeinflusst haben. Jetzt ist der richtige Zeitpunkt gekommen, auch die Nutzungszähler zurückzusetzen. Nur so können Sie später feststellen, wie häufig die erstellten Indizes verwendet wurden. Um die Nutzungszähler zurückzusetzen, benutze ich nach wie vor am liebsten den i5/OS Befehl „Change Object Description“ (CHGOBJD):

```
CHGOBJD OBJ(ITPSQL/*ALL) OBJTYPE(*FILE) USECOUNT(*RESET)
```

Trotz größter Sorgfalt kann es passieren, dass wir Zugriffspfade doppelt erstellt haben. Sie prüfen am besten jetzt noch einmal, welche Zugriffspfade existieren und entfernen unnötige Indizes. Am einfachsten gelingt dieses Vorhaben, wenn Sie im System i Navigator die Datenbank und das Schema öffnen und anschließend den Menüpunkt „Indizes“ wählen. Es ist geschickt, die Anzeige zu sortieren. Dafür müssen sie nur auf die Spalte „Schlüsselspalten“ klicken.

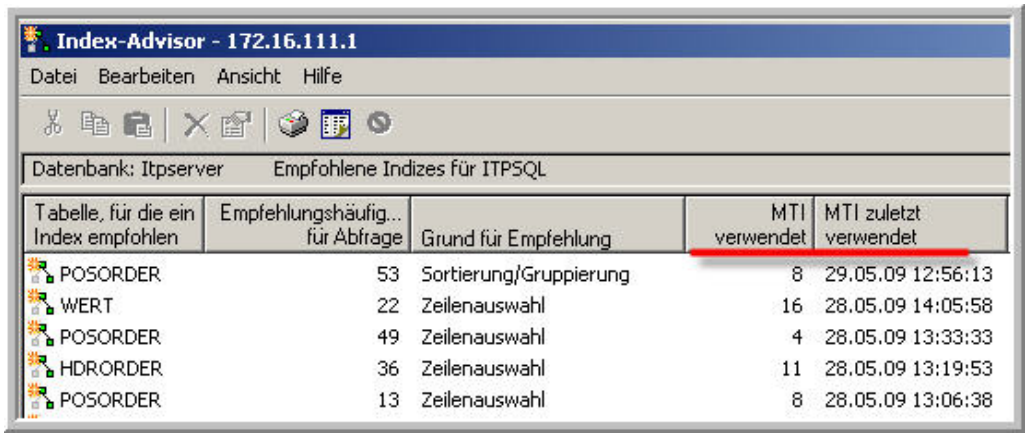
Doppelte Zugriffspfade prüfen

Bedenken Sie aber hierbei, dass eventuell noch logische DDS-Dateien vorhanden sind, die als Index genutzt werden, aber in dieser Anzeige nicht dargestellt werden!

7.3.6
Seite 50

Jetzt ist die Zeit gekommen, die Applikation oder einzelne Programme zu testen!

Nach einer mehr oder weniger langen Testphase prüfen Sie als erstes, ob temporäre Indizes erstellt wurden: Markieren Sie das entsprechende Schema ➜ rechts Klick ➜ Index Advisor ➜ Index Advisor:

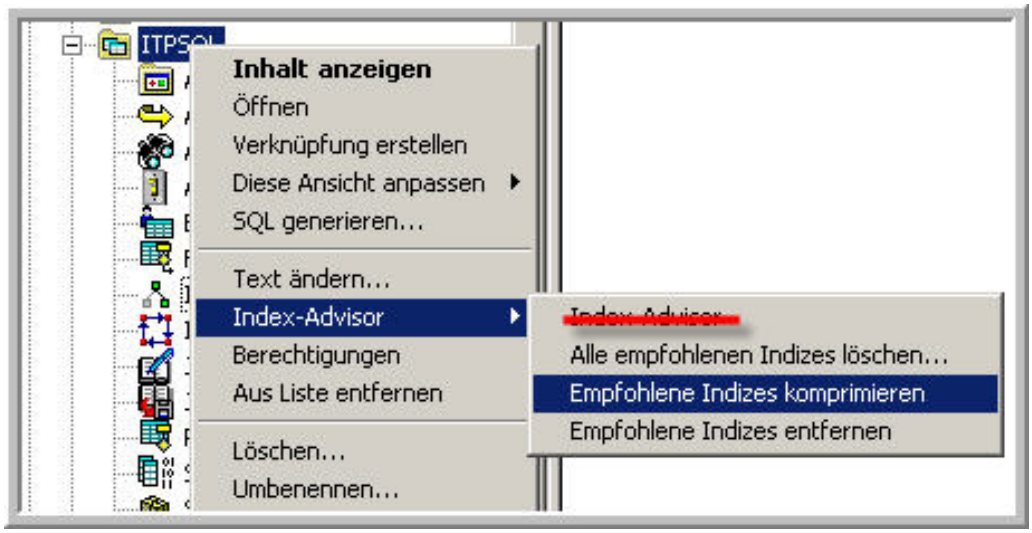


Tabelle, für die ein Index empfohlen	Empfehlungshäufig... für Abfrage	Grund für Empfehlung	MTI verwendet	MTI zuletzt verwendet
POSORDER	53	Sortierung/Gruppierung	8	29.05.09 12:56:13
WERT	22	Zeilenauswahl	16	28.05.09 14:05:58
POSORDER	49	Zeilenauswahl	4	28.05.09 13:33:33
HDRORDER	36	Zeilenauswahl	11	28.05.09 13:19:53
POSORDER	13	Zeilenauswahl	8	28.05.09 13:06:38

Wurden temporäre Indizes erstellt?

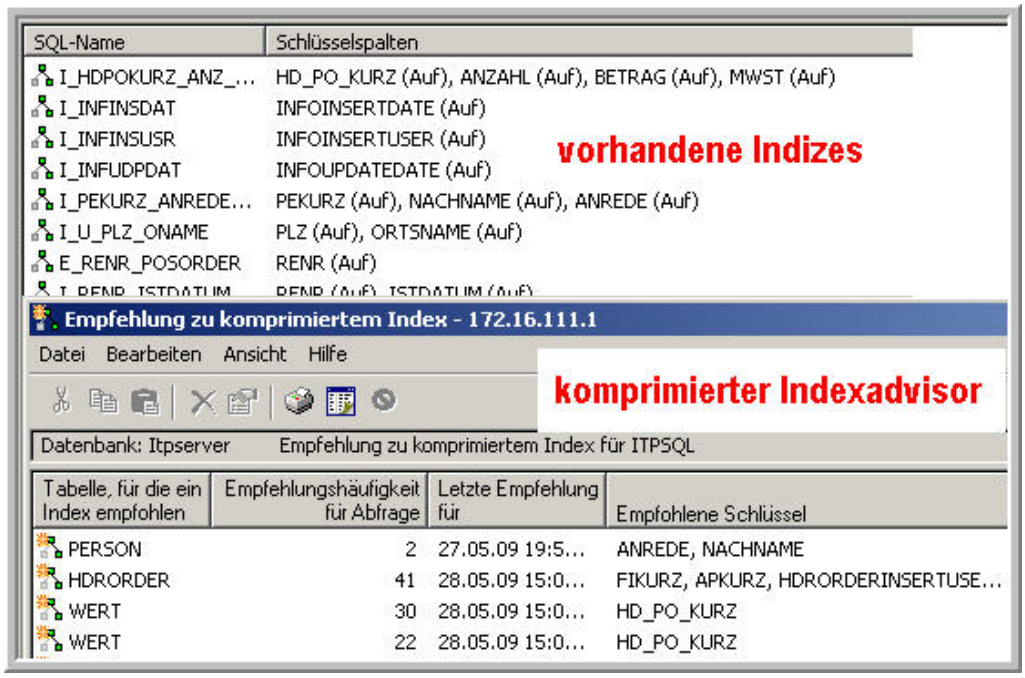
Weit rechts stehend finden Sie die benötigten Informationen. Falls temporäre Indizes erstellt wurden, können Sie sich über das Kontextmenü die Anweisungen anzeigen lassen, die für die Erstellung des temporären Objektes verantwortlich sind. Anhand der SQL Anweisungen und der Verwendung müssen Sie entscheiden, ob es sich lohnt den Index permanent zu erstellen.

In einem nächsten Schritt prüfen Sie, welche Indizes der Indexadvisor Ihnen vorschlägt. Hierzu öffnen Sie besser einen weiteren Advisor. Markieren Sie das Schema ➜ Rechtsklick ➜ „Index Advisor“ ➜ „Empfohlene Indizes komprimieren“:



Welche Indizes schlägt der Index Advisor vor?

Die komprimierte Anzeige hat deutliche Vorteile, da die Zugriffspfade bereits verdichtet sind und keine doppelten Zugriffspfade angezeigt werden. Ich öffne zu diesem Zweck gern ein zweites Fenster mit den vorhandenen Indizes. Die gleichzeitige Anzeige beider Fenster erleichtert die Überprüfung der Indexvorschläge:

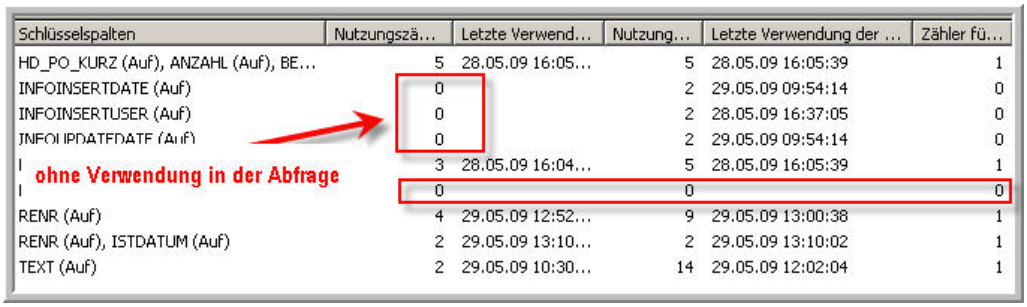


Welche Indizes werden ggf. noch benötigt?

Prüfen Sie jetzt sehr sorgfältig, welche Indizes empfohlen werden und wie häufig der Vorschlag erteilt wurde. Bedenken Sie aber, nicht immer sind die Ratschläge des Advisors brauchbar und Sie erhalten für alle SQL Abfragen Indexvorschläge unabhängig vom Laufzeitverhalten der jeweiligen Abfrage! Am besten wird es wieder sein, sich auch hier die zugehörigen SQL Anweisungen anzuzeigen und deren Laufzeitverhalten zu prüfen. So können Sie entscheiden, ob Sie den zusätzlichen Index benötigen. Leider gibt es bislang keinerlei Möglichkeiten den Index Advisor zu kontrollieren. Auch wenn Sie keine Performanceprobleme mit der Applikation haben, wird der Index Advisor weiterhin seine Arbeit verrichten und auch für eine „rund“ laufende Applikation viele Vorschläge entwickeln.

7.3.6
Seite 52

In einem weiteren Schritt prüfen Sie, wie häufig die erstellten Indizes verwendet wurden, indem Sie die Nutzungszähler kontrollieren. Öffnen Sie im System i Navigator das Schema ➜ Tabellen ➜ markieren Sie eine Tabelle ➜ Rechtsklick ➜ Indizes anzeigen:



Schlüsselspalten	Nutzungszä...	Letzte Verwend...	Nutzung...	Letzte Verwendung der ...	Zähler fü...
HD_PO_KURZ (Auf), ANZAHL (Auf), BE...	5	28.05.09 16:05...	5	28.05.09 16:05:39	1
INFOINSERTDATE (Auf)	0		2	29.05.09 09:54:14	0
INFOINSERTUSER (Auf)	0		2	28.05.09 16:37:05	0
INFOUPDATEDATE (Auf)	0		2	29.05.09 09:54:14	0
	3	28.05.09 16:04...	5	28.05.09 16:05:39	1
	0		0		0
RENR (Auf)	4	29.05.09 12:52...	9	29.05.09 13:00:38	1
RENR (Auf), ISTDATUM (Auf)	2	29.05.09 13:10...	2	29.05.09 13:10:02	1
TEXT (Auf)	2	29.05.09 10:30...	14	29.05.09 12:02:04	1

Welche Indizes werden nicht benötigt?

Falls Sie feststellen, dass einige Indizes nur zu statistischen Zwecken verwendet werden, können Sie in Betracht ziehen den Index zu löschen und durch Spaltenstatistiken zu ersetzen. Der Optimizer bevorzugt Schlüsselanfragen, da die Informationen exakter sind. Häufig ist es aber für die Erstellung des Ausführungsplanes unerheblich, ob Spaltenstatistiken oder Indizes verwendet werden. Für die Performance Ihres Systems sind aber in jedem Fall die Spaltenstatistiken die bessere Wahl. Sollten einige Indizes überhaupt nicht verwendet werden, kann es hierfür verschiedene Gründe geben, wie Sie im vorangegangenen Kapitel erfahren haben. Es kann aber auch sein, dass Sie feststellen, dass der Index tatsächlich überflüssig ist, dann sollten Sie ihn auch umgehend löschen!

7.3.7 Materialized Query Tables – Einsatz und Wirkung

Vielleicht haben Sie schon mal etwas von Materialized Query Tables gehört? Die deutsche Übersetzung lautet „Tabellen für gespeicherte Abfragen“. Eine furchtbare Übersetzung, wie ich finde, lassen Sie uns daher im Folgenden einfach von MQTs sprechen. Andere Datenbankhersteller kennen diese Objektart im Übrigen auch, bezeichnen sie allerdings etwas anders: Microsoft spricht von „Indexed Views“ und Oracle eher von „Materialized Views“.

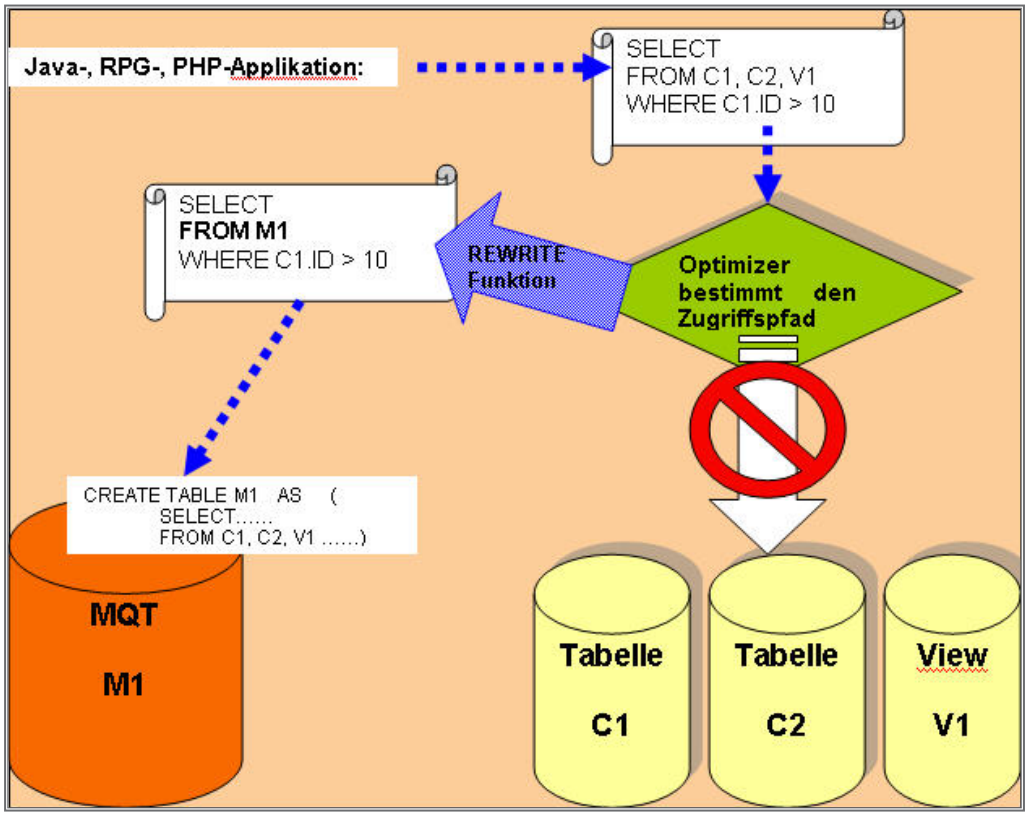
Die Theorie materialisierter Sichten ist schon seit den 80er Jahren bekannt, wurde aber erst in jüngster Zeit von den Datenbankherstellern in ihren Produkten umgesetzt. Die Grundidee hierbei ist, eine Tabelle zu definieren, die auf dem Result Set einer oder mehrerer Tabellenabfragen basiert. So kann man beispielsweise komplexe Join-Verknüpfungen oder aggregierte Abfrageergebnisse als MQT speichern. Jede Anwendung, die die Ergebnisse der komplexen Abfrage benötigt, kann direkt auf das Ergebnis zugreifen und muss nicht selbst die JOIN-Verknüpfungen oder Gruppierungen angeben. Der Trick scheint also darin zu bestehen, vorbereitete Ergebnisse weiter zu verwenden.

Einige Leser werden sich jetzt fragen, worin der Vorteil besteht, denn schließlich könnten wir diese Wirkung ja auch mit einer SQL View erzielen. Der Unterschied zwischen einer SQL View und einer MQT besteht vor allem darin, dass in der MQT die Abfrageergebnisse – also Daten – gespeichert werden. Die Daten werden also redundant gehalten – einmal in der oder den Basistabellen und ein zweites Mal in der MQT. Normalerweise ist dies eher ein negativer Punkt für relationale Datenbanksysteme; schließlich betreiben wir einen nicht unerheblichen Aufwand, um mit der Datenbanknormalisierung genau diese redundante Datenhaltung zu eliminieren! Und dieses Ergebnis ließe sich natürlich auch mit einem „normalen“ CREATE TABLE Befehl erreichen.

Ergänzung 11/2009

7.3.7
Seite 2

Der interessante Punkt ist in diesem Zusammenhang die „REWRITE"-Funktionalität des SQL Optimizers. Der Optimizer kann SQL-Abfragen selbstständig umbauen. Was die „REWRITE"-Funktionalität des SQL Optimizers mit MQTs zu tun hat, soll die folgende Grafik verdeutlichen:



Die Grundidee

Gehen wir zunächst davon aus, dass für eine „JOIN"-Verknüpfung der Tabellen C1, C2 und V1 eine MQT erzeugt wurde. Anschließend fordert irgendein Job alle Daten der verknüpften Tabellen an, deren ID größer als 10 ist. Der Optimizer analysiert das SQL-Statement und erkennt, dass die verknüpften Tabellensätze bereits in der MQT gespeichert sind. Durch den SQL Optimizer, der letztendlich den Zugriffspfad auf die Daten bestimmt, wird die SQL-Abfrage jetzt automatisch auf die MQT umgeleitet, ohne die Ergebnismenge zu verändern. Anwender müssen also weder die vorhandenen MQTs im System kennen noch ihre bestehenden SQL-Statements ändern und profitieren dennoch vom Einsatz der MQTs. Diese sehr mächtige Funktionalität benötigt allerdings bestimmte Optimizereinstellungen und wird nur in Betracht gezogen, wenn Sie es wünschen. Dazu später mehr. Jetzt sollten wir zunächst einmal eine MQT erstellen.

7.3.7.1 MQTs erstellen

Ich denke, die Grundidee ist verständlich. Es gehört auch sicherlich nicht viel Fantasie dazu, um sich die vielfältigen Einsatzmöglichkeiten einer MQT vorzustellen. Andererseits haben wir nur begrenzten Einfluss auf den Optimizer und die Verwendung unserer MQTs im Optimierungsprozess. Und hier liegt auch das Problem: Wenn wir die MQTs nicht sorgfältig planen und die Datenkopien nicht pflegen, könnten die MQTs auch Probleme bereiten. Bevor Sie also beginnen, MQTs im Optimierungsprozess einzusetzen, sollten Sie sich detailliert mit den Möglichkeiten der Erstellung beschäftigen:

Als Basistabellen können Sie „einfache" Tabellen, SQL Views, Table Expressions und auch User Defined Table Functions verwenden. In meinen Beispielen nutze ich die Tabellen „HdrOrder", „PosOrder" und die Tabelle „Wert" als Basistabellen; sie enthalten die Auftragseingänge der vergangenen Jahre und sind Grundlage vieler statistischer Auswertungen. Die Tabellen sind mittlerweile gewachsen, denn sie enthalten nicht nur aktuelle, sondern auch die abgeschlossenen Projekte zurückliegender Jahre. Für diese Tabellen erstelle ich MQTs, um vorhandene Reports und auch zukünftige Abfragen zu beschleunigen. Natürlich könnte ich auch das Datenbankdesign ändern und die Datensätze beispielsweise auf verschiedene Jahrestabellen verteilen. Das hätte aber zur Folge, dass bestehende Auswertungen angepasst und HLL-Programme aktualisiert werden müssten. Und falls ich irgendwann einmal die Auswertungen mehrerer Jahre benötige, hätte ich das Problem, dass komplexe UNION-Abfragen aufgebaut werden müssten. Daher entscheide ich mich für die Erstellung von MQTs und verzichte auf aufwendige Datenbankänderungen.

MQTs können mit der SQL-Anweisung „CREATE TABLE" erstellt werden. Probieren wir es aus, indem wir als Erstes eine MQT erstellen, die die Auftragswerte des Jahres 2008 beinhaltet:

7.3.7
Seite 4

```
CREATE TABLE MQTAUFTRAGSEINGANG2008
AS

(SELECT HDKURZ, FIKURZ, APKURZ, EINGANGSDATUM, SOLLDATUM,
ISTDATUM, STATUS, POSORDER.ISTDATUM AS POSFERTIG, POSORDER.
TEXT, RENR, RECHNUNGSDATUM, EINHEIT, ANZAHL, BETRAG, ANZAHL
* BETRAG AS POSWERT_NETTO, ANZAHL * BETRAG * 1.19 AS POSWERT_
BRUTTO

FROM HDRORDER JOIN POSORDER USING(HDKURZ)
JOIN WERT ON POKURZ = HD_PO_KURZ

WHERE EINGANGSDATUM BETWEEN '01.01.2008' AND '31.12.2008'
)

      DATA INITIALLY IMMEDIATE
      REFRESH DEFERRED
      MAINTAINED BY USER
      ENABLE QUERY OPTIMIZATION;
```

Das oben gezeigte SELECT-Statement wird in der neuen Tabelle MQTAuftragseingang2008 gespeichert. Das Select-Statement wird benötigt, um die Daten der MQT zu aktualisieren. Die Möglichkeiten neue Tabellen mit Hilfe eines SELECT Befehls zu erstellen, gibt es bereits seit Release V5R2. Diese Tabellen wurden allerdings noch nicht im Optimierungsprozess verwendet. Den eigentlichen Unterschied machen seit Release V5R4 die vier Klauseln am Ende des „CREATE TABLE"- Statements aus:

Data Initially Immediate oder Data Initially Deferred
Die Klauseln „Data Initially Immediate" und „Data Initially Deferred" bestimmen, ob die MQT bereits beim Erstellen gefüllt oder ob nur eine leere Datei erzeugt werden soll. Wird „Data Initially Immediate" angegeben, wird die Tabelle sofort mit Daten versorgt. Verwenden Sie hingegen die Klausel „Data Initially Deferred", erhalten Sie zunächst eine leere Datei.

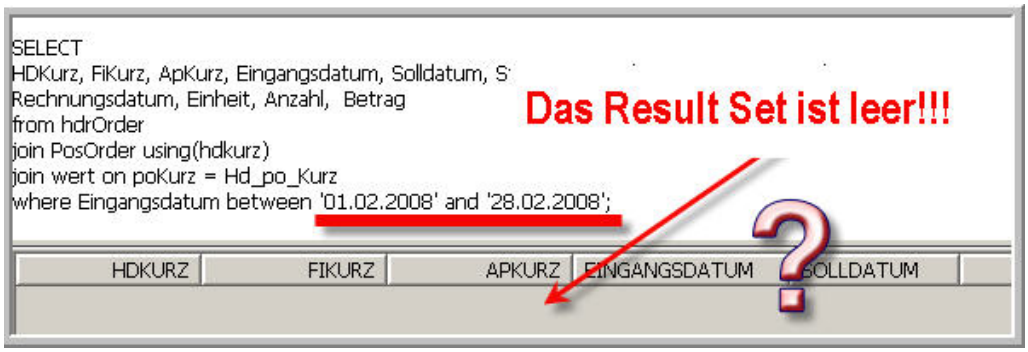
Refresh Deferred und Maintained by User

Aufgrund der Klausel „Maintained by User“ erfolgt die Pflege der MQT durch den Anwender. Er ist für die Aktualisierung der Daten verantwortlich. Diese Klausel muss im direkten Zusammenhang mit der Klausel „Refresh Deferred“ gesehen werden. Mit Hilfe der Option „Refresh Deferred“ soll gesteuert werden, ob die MQT auf Anforderung oder automatisch bei Änderung der Basistabellen aktualisiert wird. Bisher wurde allerdings nur die verzögerte Aktualisierung – also „Refresh Deferred“ für die UDB DB2 for System i – realisiert. Das heißt: Der Inhalt einer MQT zeigt nicht unbedingt die aktuelle Datenlage. Schauen Sie, was passieren kann:

Ich füge zunächst jeweils einen Satz in die Basistabellen ein:

```
INSERT INTO HDRORDER
(FIKURZ, APKURZ, EINGANGSDATUM, SOLLDATUM, ISTDATUM, TEXT,
STATUS)
VALUES(3, 0, '12.02.2008' , '01.04.2008', '05.04.2008',
'RGP PROGRAMMIERUNG ', '4');

INSERT INTO POSORDER
(POKURZ, HDKURZ, ISTDATUM, TEXT, RENR, RECHNUNGSDATUM)
VALUES(1041, 190, '22.02.2008' , 'RGP GRUNDLAGEN I ', '548',
'05.04.2008');

INSERT INTO WERT(HD_PO_KURZ, EINHEIT, ANZAHL, BETRAG, MWST)
VALUES(1041, 20 , 24, 120.00, 0.19);
```

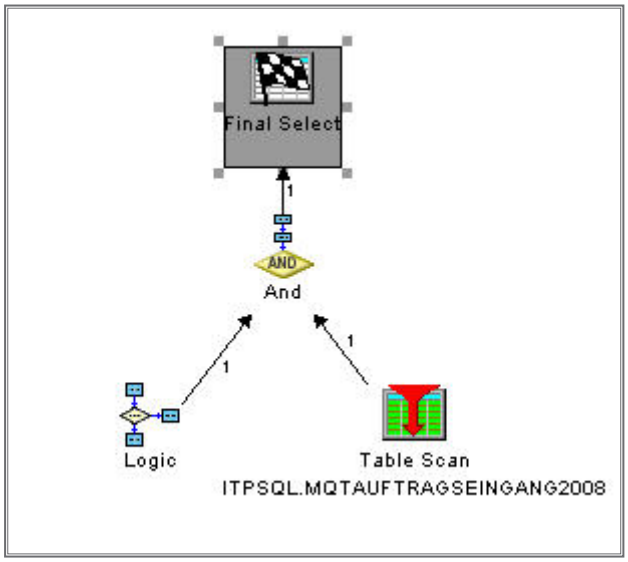
Anschließend selektiere ich alle Auftragseingänge aus dem Monat Februar 2008. Wir müssten somit im Result Set den soeben eingefügten Satz erhalten. Das Ergebnis sieht anders aus:



Wirkung der Option „Refresh Deferred“

7.3.7
Seite 6

Der eingefügte Satz wird nicht angezeigt. Der Optimizer hat – ohne mein Wissen – die MQT für die Ausführung der Abfrage verwendet – wie die folgende Abbildung beweist:



Rewrite-Funktionalität des Optimizers

Aufgrund der Klauseln „Refresh Deferred“ und „Maintained by User“ muss die MQT nämlich jedes Mal, wenn die Basistabellen neue Datensätze erhalten, durch den Benutzer aktualisiert werden. Dies geschieht am einfachsten mittels der Anweisung REFRESH TABLE.

```
REFRESH TABLE MQTAUFTRAGSEINGANG2008;
```

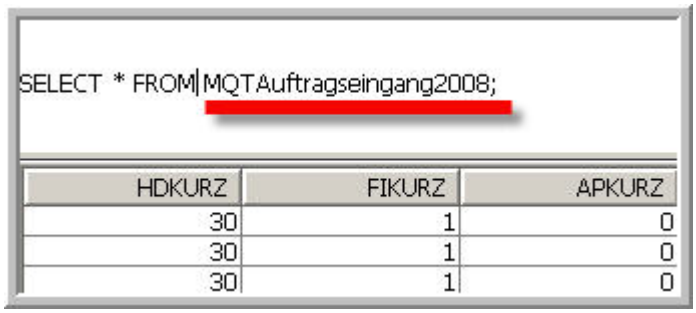
Führe ich die Abfrage jetzt ein zweites Mal aus, ist die Welt wieder in Ordnung:

MQTs müssen aktualisiert werden

Aus diesem Grund sollten Sie zurzeit MQTs nur für Abfragen einsetzen, die keine aktuellen Daten benötigen – beispielsweise kumulierte Umsatzzahlen aus dem letzten Jahr oder dem Vormonat. Andernfalls müssten Sie den Aktualisierungszeitpunkt konkretisieren können – beispielsweise in der Nacht nach einer Datenübertragung. Wenn sich Ihre Datenbasis aber laufend ändert, dann müssen Sie, z. B. über entsprechende Trigger, die Aktualisierung Ihrer MQTs sicherstellen. Ob Sie in diesen Fällen noch entsprechende Performancevorteile haben, kann nicht allgemein beantwortet werden. Dies hängt von der jeweiligen Situation und Ihrer Datenbasis ab. IBM hat bereits für die DB2 der System z Server die Klausel „Refresh Immediate“ und „Maintained by System“ realisiert. Es kann also nicht mehr lange dauern, bis diese Klausel auch für die DB2 for System i verfügbar ist und dann wird dieses Handicap entfallen.

Enable Query Optimization oder Disable Query Optimization
Falls Sie jetzt leicht beunruhigt sind und bereits erwägen MQTs niemals einzusetzen, kann ich Sie beruhigen. Wenn Sie MQTs erstellen, werden diese nicht automatisch vom Optimizer verwendet. Ich habe dem Optimizer ausdrücklich erlaubt, alle MQTs im Optimierungsprozess einzusetzen, um diese Auswirkungen demonstrieren zu können. Außerdem sind die Klauseln „Enable Query Optimization” und „Disable Query Optimization“ im Release V5R4 hinzugekommen. Mit ihnen kann festgelegt werden, ob der Optimizer die MQT für die Optimierung verwenden darf. Wenn „Disable Query Optimization“ angegeben wird, dann berücksichtigt der Query Optimizer die MQT nicht für die Ausführung der SQL Abfrage. Grundsätzlich stehen dem Optimizer für die Ausführung daher nur MQTs zur Verfügung, die mit der Klausel „Enable Query Optimization” erstellt wurden.

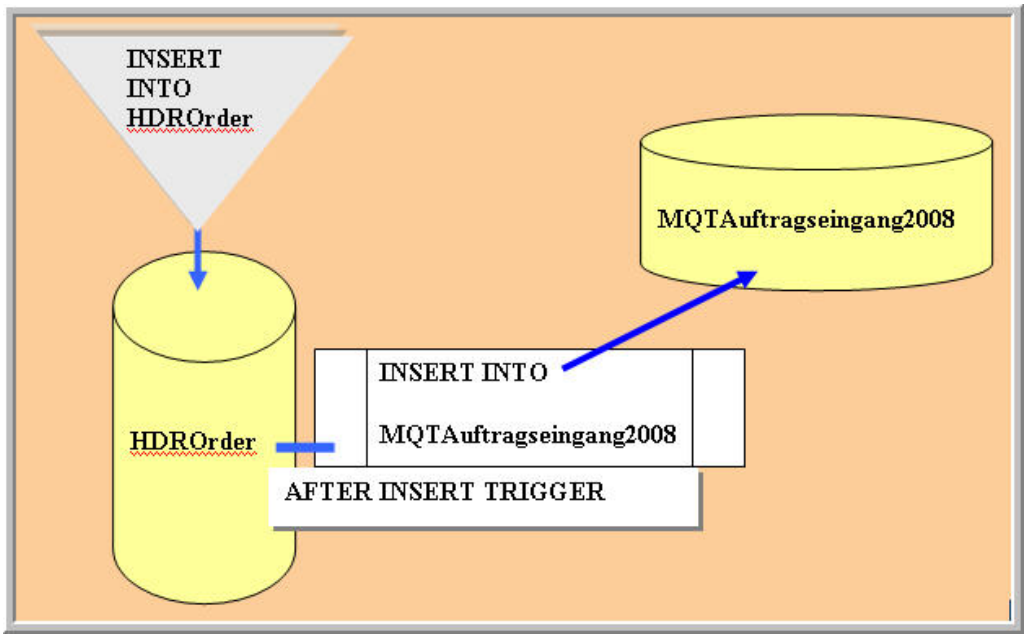
Die MQT kann jedoch immer im SQL Statement explizit angesprochen werden. Sie können nämlich jede MQT – anders als die SQL Indizes – direkt im SQL Statement verwenden:



Explizite Verwendung von MQTs

7.3.7
Seite 8

Auch Update-, Insert- und Delete-Anweisungen sind denkbar. Sie machen aber nur in Kombination mit den Basistabellen Sinn. Denkbar wäre z. B. folgendes Szenario:



Szenario Explizite Verwendung der MQT

Ein After-Insert-Trigger reicht die neuen Datensätze der Basistabelle an die MQT weiter. Einzelne Insert-Anweisungen sind in diesem Fall sicherlich effektiver als ein Refresh-Table-Befehl.

Als letztes bleibt die Frage zu klären, welche SQL-Statements in der SELECT-Anweisung einer MQT unterstützt werden? Sie können viele bekannte SQL Statement und SQL Funktionen verwenden. Allerdings gibt es auch ein paar Einschränkungen, die Sie im SQL Referenzhandbuch finden und die abhängig von ihrem Release sind. Im Release V6R1 gelten beispielsweise u. a. folgende Restriktionen:

- Das SELECT-Statement darf keine anderen MQTs, keine temporären Tabellen, keine programmbeschriebenen Dateien oder ein DDS Logical File in der FROM-Klausel referenzieren.
- Soll die MQT im Optimierungsprozess verwendet werden, dürfen keine Sonderregister und auch keine non-deterministischen Funktionen im SELECT-Statement eingesetzt werden.

7.3.7.2 Systemvorbereitungen für den MQT-Einsatz

Bevor Ihr System MQTs im Optimierungsprozess verwendet, müssen Sie noch einige Umgebungseinstellungen ändern:

Dem Attribut „Materialized_Query_Table_Usage“ in der Attributdatei „QAQQINI“ müssen Sie den Wert „*ALL“ oder „*USER“ zuweisen. Derzeit ist es egal, welchen Wert Sie verwenden, da es ohnehin nur benutzergesteuerte MQTs gibt. Mit dieser Einstellung erlauben Sie dem Optimizer alle MQTs für die Abfrage zu verwenden, die mit der Klausel „Enable Query Optimization” erstellt wurden. Per Default ist das Attribut auf „*NONE“ gesetzt, so dass der Optimizer ohne Ihr Eingreifen keine MQTs im Optimierungsprozess verwenden wird.

Nehmen wir an, dass Sie die Einstellung für das Attribut „Materialized_Query_Table_Usage“ entsprechend geändert haben, dann könnten Sie zusätzlich dem Attribut „Materialized_Query_Table_Refresh_Age“ den Wert „*ANY“ zuweisen. Sie erlauben dadurch dem Optimizer alle MQTs für die Ausführung der Abfragen zu nutzen – unabhängig vom Zeitpunkt der letzten Datenaktualisierung.

Besser könnte es sein, wenn Sie einen Grenzwert hinterlegen. Der Optimizer ermittelt zunächst, wie viele Sekunden seit der letzten Aktualisierung der MQT vergangen sind. Liegt der Zeitpunkt der Aktualisierung vor Ihrem hinterlegten Grenzwert, kann die MQT verwendet werden. Andernfalls muss der Optimizer sich für eine alternative Ausführungsmethode entscheiden.

Gehen wir von folgendem Szenario aus:

Jede Nacht zwischen fünf und sechs Uhr morgens werden alle MQTs aktualisiert. Falls einige Aktualisierungen fehlschlagen, möchte ich verhindern, dass der Optimizer die „veralteten“ MQTs im Optimierungsprozess verwendet. Ich muss also den Optimizer anweisen, nur MQTs zu verwenden, die innerhalb der letzten 24 Stunden aktualisiert wurden. 24 Stunden entsprechen 86400 Sekunden. Ich setzte daher den Parameter „MATERIALIZED_QUERY_TABLE_REFRESH_AGE“ der Optionsdatei auf den Wert „86400“.

```
UPDATE ITPCR.QAQQINI SET QQVAL = '86400'
WHERE QQPARM = 'MATERIALIZED_QUERY_TABLE_REFRESH_AGE';
```

Ergänzung 11/2009

Die Aktualisierung meiner MQT liegt allerdings mehr als 24 Stunden zurück. Dies lässt sich sehr schnell mit folgender Abfrage überprüfen:

```
SELECT TABLE_NAME, LAST_ALTERED_TIMESTAMP, TIMESTAMPDIFF(2,
CAST(CURRENT TIMESTAMP – LAST_ALTERED_TIMESTAMP AS CHAR(22)))
AS
SEKUNDEN, TIMESTAMPDIFF(4, CAST(CURRENT TIMESTAMP – LAST_AL-
TERED_TIMESTAMP AS CHAR(22))) AS MINUTEN
FROM QSYS2.SYSTABLES
WHERE TABLE_SCHEMA = 'ITPSQL' AND TABLE_TYPE = 'M';
```

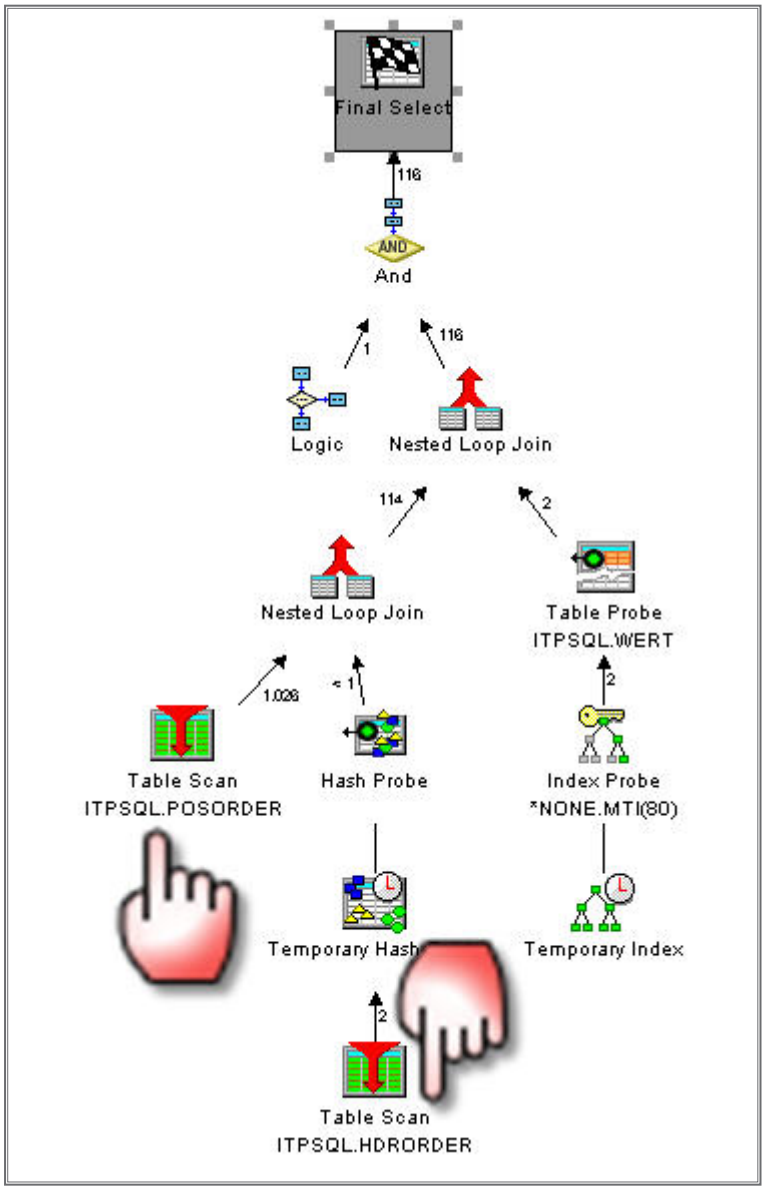
Für die Abfrage nutze ich die Informationen in der Systemtabelle „QSYS2. SYSTABLES". Sie enthält die Spalte „Last_Altered_Timestamp". Die Spalte enthält den Zeitstempel der letzten Aktualisierung. Die Zeitspanne, die seit der letzten Aktualisierung vergangen ist, ermittle ich mit der SQL-Funktion „Timestampdiff". Die Funktion kann eine Zeitspanne zwischen zwei Zeitmarken berechnen. Das erste Argument bestimmt die verwendete Einheit. Enthält das erste Argument den Wert „4", wird das Ergebnis in Minuten dargestellt. Der Wert „2" ermittelt die Differenz in Minuten.

TABLE_NAME	LAST_ALTERED_TIMESTAMP	SEKUNDEN	MINUTEN
MQTAUFTRAGSEINGANG2008	2009-08-01 01:10:13.772000	604667	10077
MQTAUFTRAGSEINGANG2007	2009-08-01 01:10:22.147000	604658	10077
MQTFINR2008	2009-08-01 01:10:56.409000	604624	10077
ORDERSUM	2009-08-08 01:03:40.796000	260	4
MQTSUMFINR2007	2009-08-01 00:24:41.326000	607399	10123

Das Ergebnis

Die Abfrage zeigt, dass die Daten meiner MQT vor mehr als 24 Stunden das letzte Mal aktualisiert wurden. Führe ich jetzt eine Abfrage aus, die mittels der MQT beantwortet werden könnte, dürfte die SQL Engine die „veraltete" MQT nicht mehr für den Datenzugriff verwenden. Ich nutze zum Test folgende Abfrage:

```
SELECT HDKURZ, FIKURZ, EINGANGSDATUM, ANZAHL * BETRAG
AS POSWERT_NETTO, ANZAHL * BETRAG * 1.19 AS POSWERT_BRUTTO

FROM HDRORDER JOIN POSORDER USING(HDKURZ)
JOIN WERT ON POKURZ = HD_PO_KURZ

WHERE EINGANGSDATUM BETWEEN '01.02.2008' AND '28.02.2008'
```



Die „veraltete" MQT wird nicht genutzt

Der oben gezeigte Ausführungsplan bestätigt meine Vermutung. Der hinterlegte Zeitwert verhindert somit, dass versehentlich Daten abgefragt werden, die nicht mehr der aktuellen Datenlage entsprechen. Sie können sich also vor „falschen" Abfrageergebnissen schützen, indem Sie in der Attributdatei „QAQQINI" einen entsprechenden Zeitwert hinterlegen.

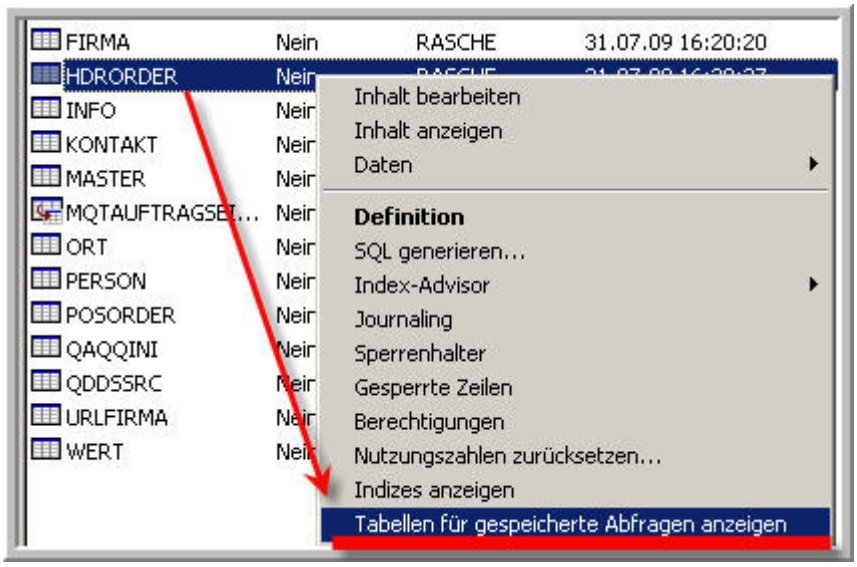
7.3.7
Seite 12

Ich habe mir anhand der Systemtabelle QSYS2.SYSTABLES einen Überblick über den Zeitpunkt der letzten MQT-Aktualisierung verschafft. Diese Systemtabelle ist aber auch nützlich, wenn Sie sich irgendwann einmal einen Überblick über die erstellten MQTs im System verschaffen wollen.

```
SELECT *
FROM SYSTABLES
WHERE TABLE_TYPE = 'M';
```

Durch die Abfrage erhalten Sie schnell einen Überblick über die bestehenden MQTs und können so von Zeit zu Zeit die Verwendung der erstellten MQTs prüfen.

Welche MQTs im System verfügbar sind, lässt sich über die grafischen Oberflächen des System i Navigators nicht ganz so einfach feststellen. Der System i Navigator bietet nur die Möglichkeit, sich über das Kontextmenü einer Tabelle die verfügbaren MQTs für die Tabelle anzeigen zu lassen:



Tabellen für gespeicherte Abfragen (MQT) anzeigen

Dies setzt allerdings voraus, dass Sie wissen für welche Tabelle MQTs erstellt wurden.

7.3.7.3 Lohnt sich der Einsatz von MQTs?

Indizes spielen sicherlich eine entscheidende Rolle, wenn es um die Performance von SQL-Abfragen geht. Aber auch MQTs können die Ausführung Ihrer SQL-Abfragen positiv beeinflussen. Unsere Datenbanken enthalten oft zentrale Basistabellen, die in vielen Auswertungen benötigt werden und mittlerweile einen erheblichen Umfang haben. In den folgenden Beispielen spielen die bereits vorgestellten Tabellen eine solche Rolle; sie enthalten die Auftragswerte der vergangenen Jahre und sind aufgrund dieser historischen Daten Grundlage vieler Auswertungen. Für diese Basistabellen habe ich MQTs erstellt, d. h. ich habe meine Auftragswerte der einzelnen Jahre auf verschiedene MQTS verteilt und in der MQT auch gleich den Auftragswert der einzelnen Positionen ermittelt. Dabei habe ich mich allerdings aus den bereits erläuterten Gründen nur auf die Daten der vergangenen Jahre beschränkt, weil ich so sicherstellen kann, dass die Datenbasis nicht mehr aktualisiert wird.

7.3.7
Seite 14

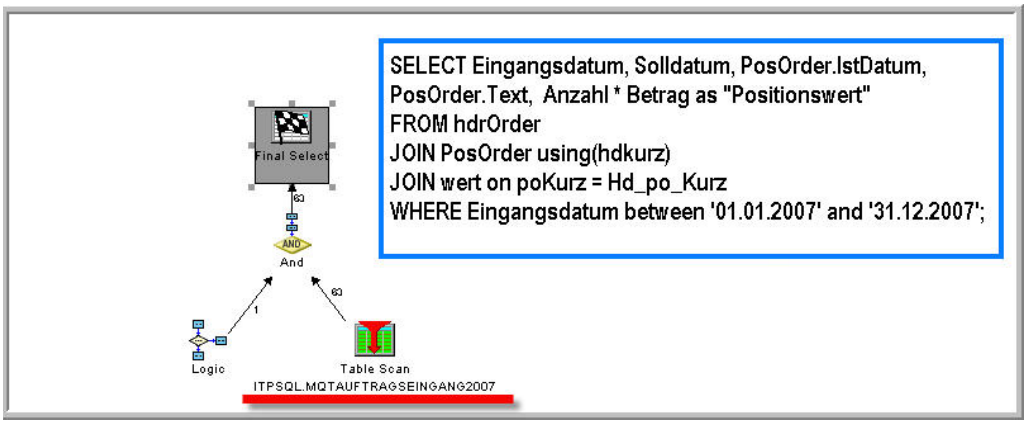
```
CREATE TABLE MQTAUFTRAGSEINGANG2008
(HDKURZ, FIKURZ, APKURZ, EINGANGSDATUM, SOLLDATUM, STATUS,
ISTDATUM, POSTEXT, RENR, REDATUM, EINHEIT, ANZAHL, BETRAG,
POSSUMME)
AS
(

SELECT HDKURZ, FIKURZ, APKURZ, EINGANGSDATUM, SOLLDATUM,
STATUS, POSORDER.ISTDATUM, POSORDER.TEXT, RENR, RECHNUNGSDA-
TUM, EINHEIT, ANZAHL, BETRAG, ANZAHL * BETRAG

FROM HDRORDER
JOIN POSORDER USING(HDKURZ)
JOIN WERT ON POKURZ = HD_PO_KURZ
WHERE EINGANGSDATUM BETWEEN ,01.01.2008‘ AND ,31.12.2008‘

)

DATA INITIALLY IMMEDIATE
REFRESH DEFERRED
MAINTAINED BY USER
ENABLE QUERY OPTIMIZATION;
```

```
CREATE TABLE MQTAUFTRAGSEINGANG2007(,,,
(HDKURZ, FIKURZ, APKURZ, EINGANGSDATUM, SOLLDATUM, STATUS,
ISTDATUM, POSTEXT, RENR, REDATUM, EINHEIT, ANZAHL, BETRAG,
POSSUMME)
AS
(

SELECT HDKURZ, FIKURZ, APKURZ, EINGANGSDATUM, SOLLDATUM,
STATUS, POSORDER.ISTDATUM, POSORDER.TEXT, RENR, RECHNUNGSDA-
TUM, EINHEIT, ANZAHL, BETRAG, ANZAHL * BETRAG

FROM HDRORDER
JOIN POSORDER USING(HDKURZ)
```

```
JOIN WERT ON POKURZ = HD_PO_KURZ
WHERE EINGANGSDATUM BETWEEN ‚01.01.2007‘ AND ‚31.12.2007‘

)

DATA INITIALLY IMMEDIATE
REFRESH DEFERRED
MAINTAINED BY USER
ENABLE QUERY OPTIMIZATION;
```

Der eigentliche Vorteil von MQTs liegt anschließend darin, dass MQTs vom SQL Optimizer während des Optimierungsprozesses automatisch berücksichtigt werden – ähnlich einem SQL Index. Das heißt: anstatt auf die Basistabellen zuzugreifen, nutzt der Optimizer eventuell die bestehenden MQTs für die Ausführung der SQL-Abfrage.

Schauen wir uns hierzu ein erstes Beispiel an: Auf meinem Server existiert die MQT „MQTAuftragseingang2007“. Diese MQT enthält nur die Aufträge des Jahres 2007. Für das folgende SQL Statement untersucht der Optimizer alle verfügbaren Indizes und MQTs; er entscheidet sich anschließend anstelle der Basistabellen die erstellte MQT „MQTAuftragseingang2007“ zu benutzen.



Der Optimizer entscheidet!

Sie sehen, ich muss die verfügbaren MQTs nicht im SQL Statement verwenden, so dass ich die vorhandenen MQTs also auch gar nicht kennen muss. Der Optimizer entscheidet ohne mein Zutun, ob sich der Einsatz der MQTs lohnt. Somit können sich die MQTs auch positiv auf bereits bestehende SQL-Abfragen auswirken, ohne dass Sie die SQL-Abfragen verändern müssen.

7.3.7
Seite 16

Aber nicht für jede Abfrage kann der Optimizer MQTs einsetzen. Damit sie im Optimierungsprozess verwendet werden, müssen einige Bedingungen erfüllt sein:

- Sie dürfen nicht mit Live-Daten oder einem sensitiven Cursor arbeiten.
- Verwenden Sie nicht die Klausel „For Update“ in Ihrem Statement „SELECT“.
- Die Abfrage muss von der SQE ausgeführt werden.
- Die MQT muss die gleiche oder eine höhere Commit-Stufe verwenden als die Abfrage.

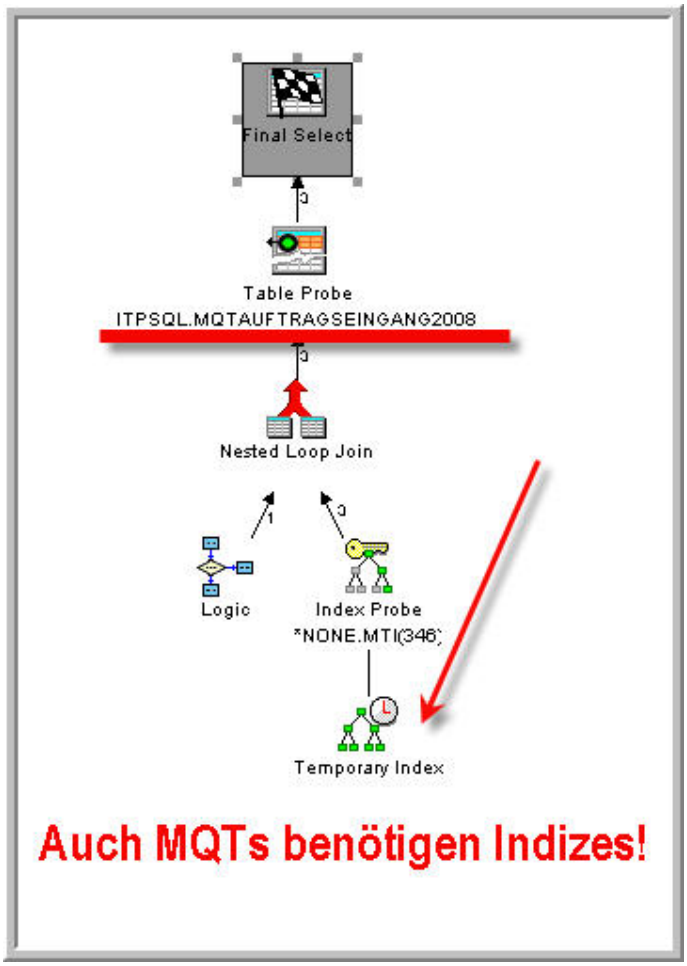
Wenn diese Kriterien erfüllt sind, kann der Optimizer die MQT vielfältig einsetzen. So können die Spaltenliste und die Selektionskriterien verändert werden und dennoch hätte der Optimizer die Möglichkeit, die MQT zu verwenden. Wichtig für die Verwendung zusätzlicher Selektionskriterien ist lediglich, dass die MQT die zusätzlichen Bedingungen anhand der eigenen Daten auflösen kann. Das gleiche gilt natürlich auch für die verwendete Spaltenliste.

Prüfen wir diese Aussage mit einer weiteren Abfrage:

Ich benötige alle Auftragspositionen im ersten Quartal 2008 mit einem Positionswert über 1.000,00 Euro.

```
SELECT EINGANGSDATUM, SOLLDATUM, POSORDER.ISTDATUM, POSORDER.
TEXT, ANZAHL * BETRAG AS „POSITIONSWERT“
FROM HDRORDER JOIN POSORDER USING(HDKURZ)
JOIN WERT ON POKURZ = HD_PO_KURZ
WHERE EINGANGSDATUM BETWEEN ‘01.01.2008’
AND ‘31.03.2008’ AND ANZAHL * BETRAG > 1000.00;
```

Die Selektion des Auftragswertes mit Hilfe der MQTs sollte problemlos möglich sein, da die MQT die Spalte „ANZAHL“ und „Betrag“ enthält.



MQTs brauchen Indizes

Tatsächlich benutzt der Optimizer – wie vermutet – die MQT für den Datenzugriff. Allerdings baut die SQL-Engine einen temporären Index für den Datenzugriff auf.

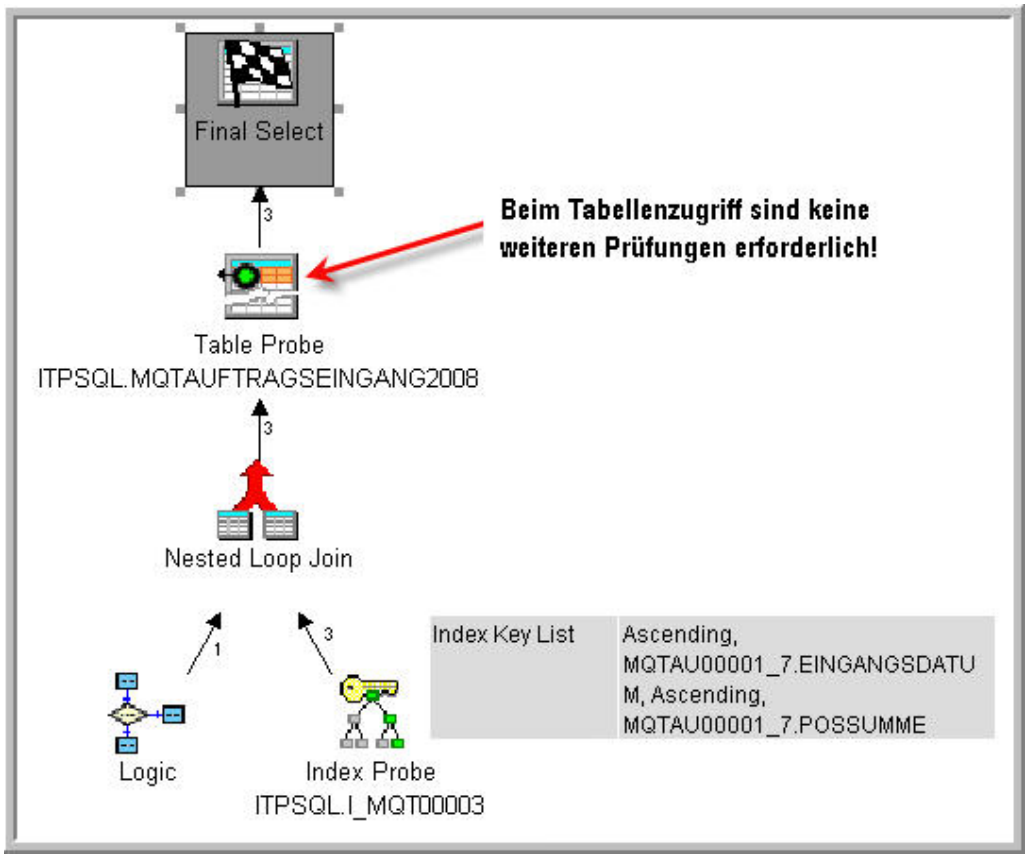
Glauben Sie bitte nicht, dass Indizes jetzt nebensächlich werden. Der Optimizer ergänzt lediglich den Optimierungsprozess und prüft neben den verfügbaren SQL Indizes auch die verfügbaren MQTs. Damit er sich für die Verwendung entscheidet, sind auch hier sinnvolle Indizes nötig. Interessant sind in diesem Zusammenhang im Übrigen die Empfehlungen des Index Advisors:

Erste...	Tabellenname	Schema	Indextyp	Spalten
☑	POSORDER	ITPSQL	Binärradix	HDKURZ
☑	HDRORDER	ITPSQL	Binärradix	EINGANGSDATUM
☑	WERT	ITPSQL	Binärradix	HD_PO_KURZ
☑	HDRORDER	ITPSQL	Binärradix	HDKURZ EINGANGSDATUM
☑	MQTAUFTRAGS...	ITPSQL	Binärradix	EINGANGSDATUM

Die Indexempfehlungen des Index Advisors

7.3.7
Seite 18

Der Index Advisor berücksichtigt in seinen Empfehlungen nicht nur die verwendete MQT, sondern auch die zugehörigen Basistabellen – wie die Abbildung zeigt. Überlegen Sie genau, ob es sinnvoll ist, alle Indizes zu erstellen! Und noch etwas fällt auf: Der Index Advisor empfiehlt für die MQT keinen zusammengesetzten Index für die Spalten „Eingangsdatum“ und „PosSumme“. Die Kombination der Spalten könnte aber sinnvoll sein, damit der Optimizer in einem Arbeitsschritt beide Bedingungen prüfen kann.



Ein zusammengesetzter Index für die MQT

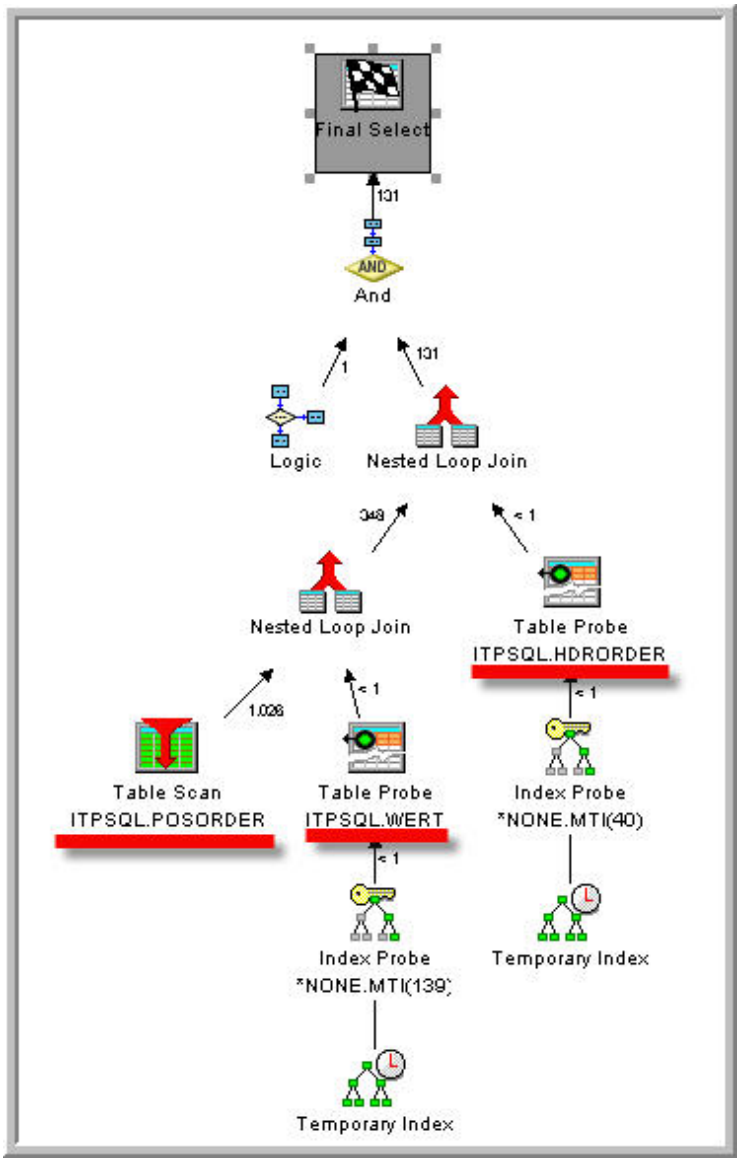
Durch den zusammengesetzten Index werden weniger I/O Counts für den Datenzugriff benötigt. Aber auch der Arbeitsspeicher Ihres Servers wird entlastet, da weniger Daten in den Arbeitsspeicher verlagert werden müssen. Weitere Informationen zum Aufbau von Indizes entnehmen Sie bitte dem Kapitel 7.3.6.

Wir wollen uns ein weiteres Beispiel anschauen. Der Optimizer verwendet MQTs auch für Abfragen, die nicht nur die Tabellen der MQT verwenden. In der folgenden Abfrage wird zusätzlich die Firmentabelle benötigt. Der Optimizer weiß, dass die von mir erstellte MQT lediglich die Daten des Jahres „2008“ enthält und weniger I/O Operationen für den Datenzugriff erforderlich sind, wenn er die MQT an Stelle der Tabelle „HDROrder“ nutzen würde. Daher formuliert der Optimizer die Abfrage auch dieses Mal neu und verwendet die MQT anstelle der Basistabellen. Dies gelingt, weil die MQT ebenfalls die Spalte „FIKURZ“ enthält.

Bleibt die Frage zu klären, ob der Optimizer ggf. sogar zwei MQTs über entsprechende „UNION“-Klauseln verknüpfen kann, wenn ich meine Abfrage auf die Jahre 2008 und 2007 erweitere?

```
SELECT FIKURZ, EINGANGSDATUM, SOLLDATUM, POSORDER.ISTDATUM,
POSORDER.TEXT, ANZAHL * BETRAG AS „POSITIONSWERT“
FROM HDRORDER
JOIN POSORDER USING(HDKURZ)
JOIN WERT ON POKURZ = HD_PO_KURZ
JOIN FIRMA USING (FIKURZ)
WHERE EINGANGSDATUM BETWEEN '01.01.2007' AND '31.03.2008'
AND ANZAHL * BETRAG > 1000.00;
```

Die Antwort ist Nein, wie folgender Ausführungsplan deutlich zeigt:



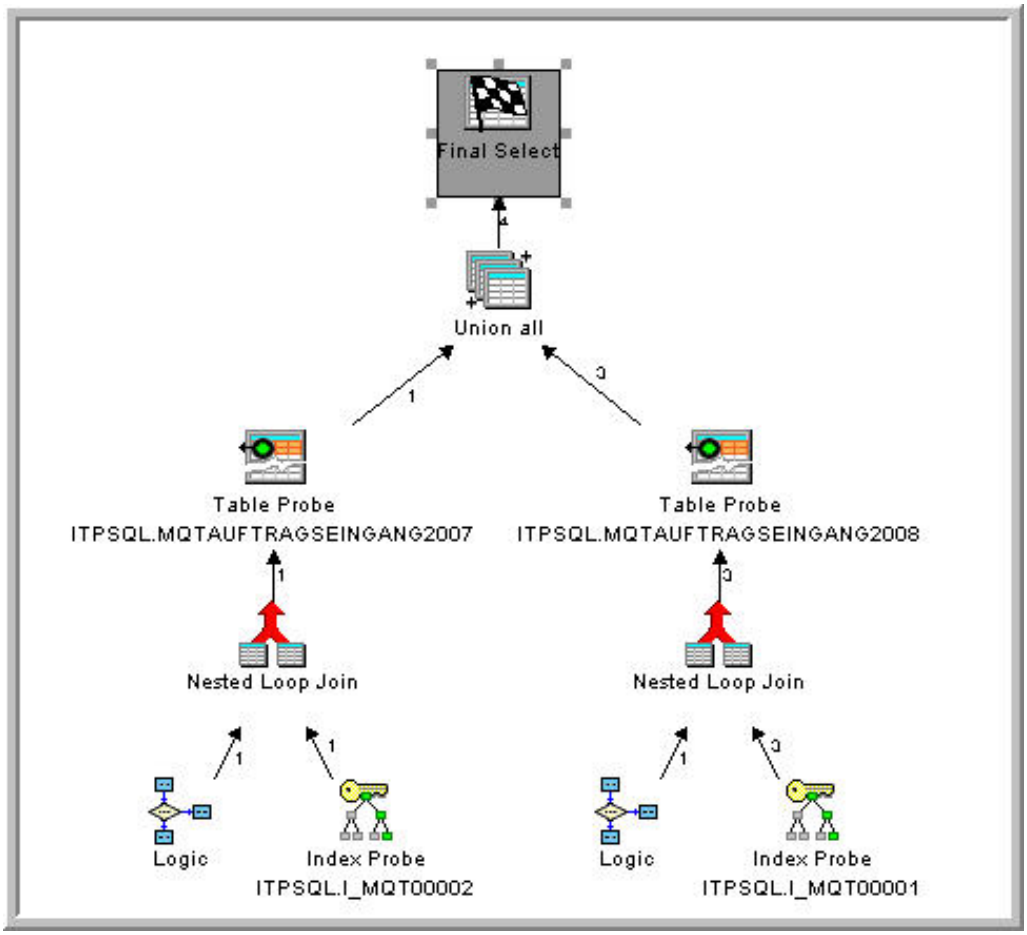
Die „Rewrite“-Funktionalität beinhaltet kein UNION

7.3.7
Seite 20

Der Optimizer ist derzeit nicht in der Lage das Statement als UNION-Abfrage selbstständig zu formulieren. Hier sind Sie gefragt: Wenn Sie glauben, dass der Zugriff über die MQTs performanter sein könnte, müssen Sie Ihre Abfrage einfach neu formulieren:

```
SELECT EINGANGSDATUM, SOLLDATUM, POSORDER.ISTDATUM, POSORDER.
TEXT, ANZAHL * BETRAG AS "POSITIONSWERT"
FROM HDRORDER
JOIN POSORDER USING(HDKURZ)
JOIN WERT ON POKURZ = HD_PO_KURZ
WHERE EINGANGSDATUM BETWEEN '01.01.2007' AND '31.03.2007'
AND ANZAHL * BETRAG > 1000.00

UNION ALL

SELECT EINGANGSDATUM, SOLLDATUM, POSORDER.ISTDATUM, POSORDER.
TEXT, ANZAHL * BETRAG AS "POSITIONSWERT"
FROM HDRORDER
JOIN POSORDER USING(HDKURZ)
JOIN WERT ON POKURZ = HD_PO_KURZ
WHERE EINGANGSDATUM BETWEEN '01.01.2008' AND '31.03.2008'
AND ANZAHL * BETRAG > 1000.00;
```

Diese Abfrage kann der Optimizer wieder problemlos mit Hilfe der bestehenden MQTs auflösen:



Der Optimizer nutzt jetzt wieder die MQTs für die Teilabfragen

MQTs eignen sich aber auch hervorragend, um Daten großer Tabellen zu verdichten. Enthält die SELECT-Anweisung einer MQT Berechnungen, die mittels der Aggregatfunktionen und der GROUP By Klausel aufgebaut werden, spricht man auch von einer Automatic Summary Table (AST).

Überlegen wir einmal, welche Kumulierungen bei meinen Beispieltabellen sinnvoll sein könnten. Ich könnte beispielsweise die Umsatzwerte eines Jahres zu Wochen- oder Monatsumsätzen verdichten. Auch die Umsatzwerte je Kunde sind sicherlich interessant. Es gibt eine Vielzahl an Möglichkeiten. Sie werden sicherlich sehr schnell eigene Ideen entwickeln!

7.3.7
Seite 22

Ich erstelle zunächst eine MQT, die die Auftragswerte eines Jahres je Kunde verdichtet:

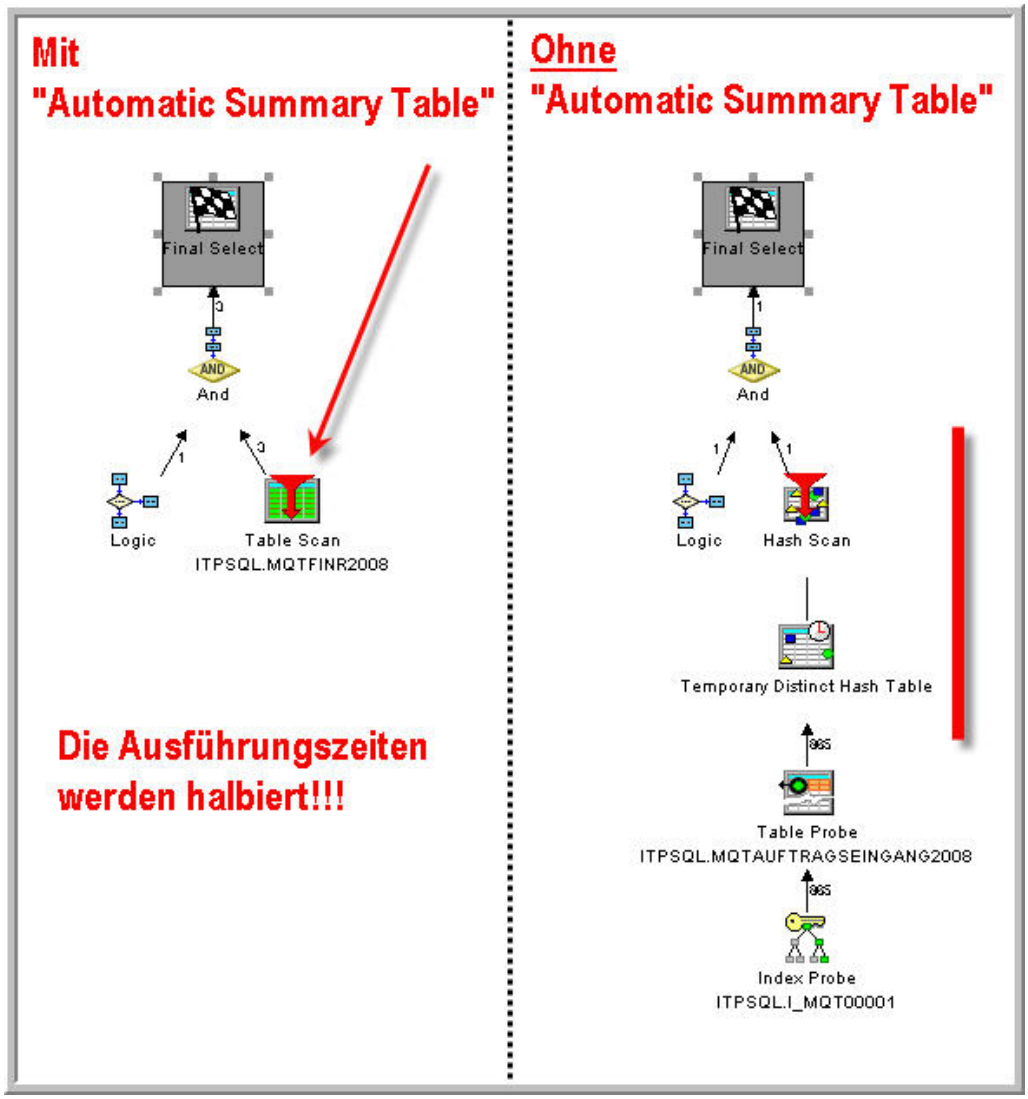
```
CREATE TABLE MQTFINR2008(FIKURZ, FISUMME, FIAVG, FICOUNT )
AS
(
SELECT FIKURZ, SUM(ANZAHL * BETRAG), AVG(ANZAHL * BETRAG),
COUNT(*)
FROM HDRORDER
JOIN POSORDER USING(HDKURZ)
JOIN WERT ON POKURZ = HD_PO_KURZ
WHERE EINGANGSDATUM BETWEEN '01.01.2008' AND '31.12.2008'
GROUP BY FIKURZ
)
DATA INITIALLY IMMEDIATE
REFRESH DEFERRED
MAINTAINED BY USER
ENABLE QUERY OPTIMIZATION;
```

Wenn Sie Aggregatfunktionen verwenden – so wie ich es getan habe – oder temporäre Spalten aufbauen, dann müssen Sie den Spalten des Result Sets entweder eine Überschrift zu weisen oder alternativ explizit die Spalten der MQT im Statement „CREATE TABLE“ benennen. Andernfalls erhalten Sie als Fehlermeldung den SQL-Status 42908 „Spaltenliste erforderlich“.

Die kumulierten MQTs können anschließend vom Optimizer geschickt für Gruppierungsabfragen verwendet werden.

```
SELECT FIKURZ, SUM(ANZAHL * BETRAG)
FROM HDRORDER
JOIN POSORDER USING(HDKURZ)
JOIN WERT ON POKURZ = HD_PO_KURZ
WHERE EINGANGSDATUM BETWEEN '01.01.2008' AND '31.12.2008'
GROUP BY FIKURZ HAVING COUNT(*) > 2;
```

Aufgrund der zuvor erstellten MQT „MQTFiNr2008“ entfällt die Aggregation der Gruppen über Hashtabellen. So erklären sich die folgenden Ausführungspläne:



Die Ausführungspläne im Vergleich

Wenn keine „Summary MQT“ für die Ausführung zur Verfügung steht, kann der Optimizer nur die WHERE-Bedingungen verwenden, um Daten von der Verarbeitung auszuschließen. Zusätzlich müssen temporäre Hash-Tabellen aufgebaut werden, um die Daten zu gruppieren. Erst im Anschluss an die Gruppierung kann die zweite Bedingung – die Anzahl der Kundenaufträge und deren Summierung – realisiert werden. Die Verwendung der MQT erlaubt dem Optimizer stattdessen, die sofortige Prüfung aller Bedingungen, da die MQT die kumulierte Anzahl der Kundenkontakte bereits enthält. Auch die Summen sind vorhanden, so dass die temporären Hash-Tabellen entfallen.

7.3.7
Seite 24

Sie können selbstverständlich die Gruppierungen verändern. Ich könnte beispielsweise in meiner Abfrage eine Monatsgruppierung hinzufügen oder bei mehrstufigen Gruppierungen auf eine Gruppierungsstufe verzichten. In diesen Fällen ändert sich natürlich der Ausführungsplan. Der Optimizer muss aber keineswegs auf die MQT verzichten. Er kann die MQT auch dann verwenden, wenn in der SQL-Abfrage höhere Gruppierungsstufen benötigt werden. Dies gilt natürlich auch, wenn Sie weitere Gruppierungen hinzufügen. Entscheidend ist lediglich, dass der Optimizer die Anforderung mit den Daten der MQT realisieren kann.

Ich denke, dass die wenigen Beispiele bereits die Vorteile der MQTs verdeutlichen konnten:

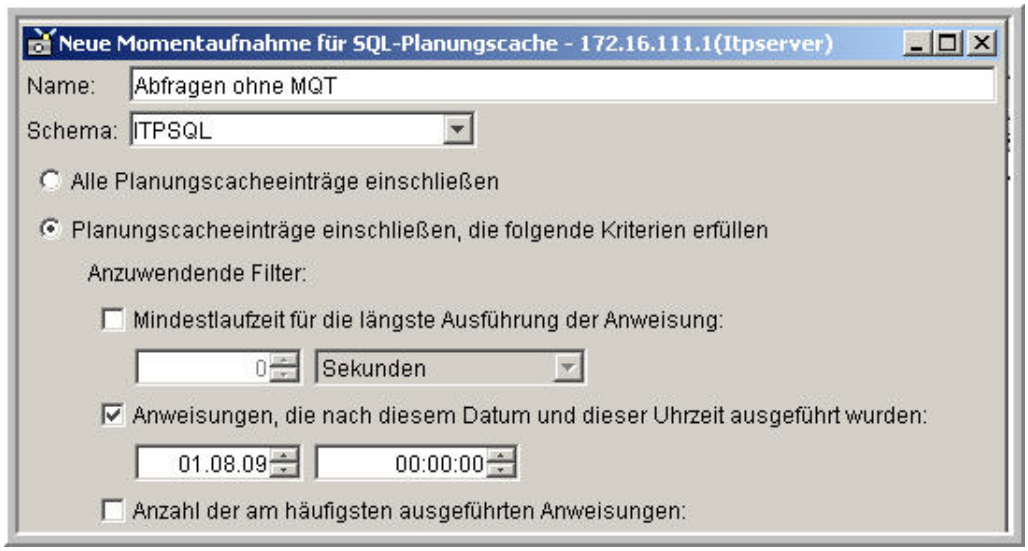
- Bestehende Reports und HLL-Programme müssen nicht verändert werden und profitieren unter Umständen dennoch von den neuen Objekten. MQTs können somit nachträglich erstellt werden, ohne dass Änderungen in der Applikation erforderlich sind.
- SQL-Anwender müssen die Inhalte bestehender MQTs nicht kennen. Der Optimizer entscheidet, ob es besser ist die Basistabelle oder die verdichteten Daten zu verwenden.
- Der Optimizer kann gezwungen werden, die MQT zu verwenden, indem sie direkt im SQL Statement verwendet wird. Mit SQL Indizes ist das nicht möglich, wie Sie wissen.

Wenn Sie beginnen MQTs für den Optimierungsprozess einzusetzen, sollten Sie nach einiger Zeit sorgfältig prüfen, welche Abfragen von den MQTs beeinflusst werden. Hierfür eignen sich hervorragend Snapshot-Vergleiche des Plan Cache. Erstellen Sie zu diesem Zweck zunächst einen Snapshot, bevor Sie MQTs erstellen.

Öffnen Sie hierfür System i Navigator ➜ Datenbanken ➜ SQL Planungscache ➜ Momentaufnahmen für SQL Planungscache

Momentaufnahme für SQL Planungscache erstellen

Im dann folgenden Bildschirm können Sie Filterkriterien definieren, um die Anzahl der Ausführungspläne zu begrenzen.



Selektionskriterien auswählen

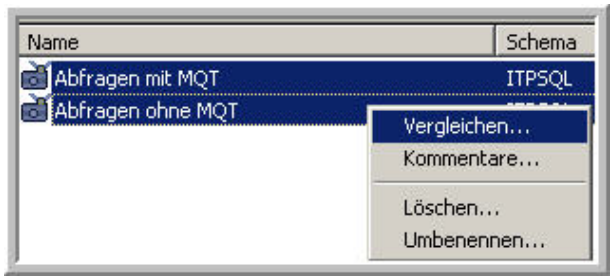
Die Daten werden anschließend im angegebenen Schema gespeichert. Wundern Sie sich also nicht, wenn Sie plötzlich in Ihrem Schema Tabellen finden, die mit der Zeichenfolge „QZG“ beginnen. Diese Dateien enthalten die Snapshotdaten des Plan Cache.

Erstellen Sie anschließend Ihre MQTs und führen Sie die Applikation einige Zeit aus. Anschließend erstellen Sie einen zweiten Snapshot. Sie müssten jetzt beide Snapshots im System i Navigator sehen, wenn Sie den Eintrag „Momentaufnahmen für SQL Planungscache“ öffnen.

Momentaufnahmen für SQL Planungscache anzeigen

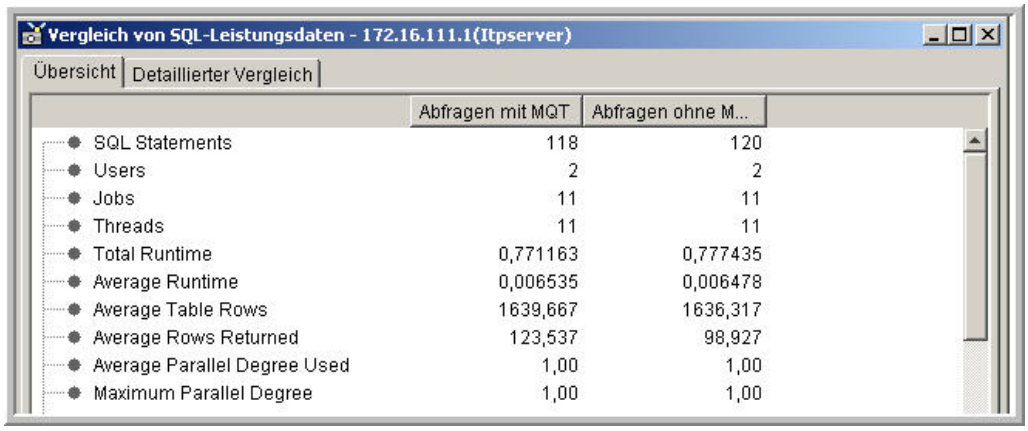
7.3.7
Seite 26

In einem weiteren Schritt können Sie anschließend die beiden Snapshots miteinander vergleichen. Markieren Sie hierfür beide Einträge und wählen Sie danach im Kontextmenü den Eintrag „Vergleichen“.



Snapshot vergleichen

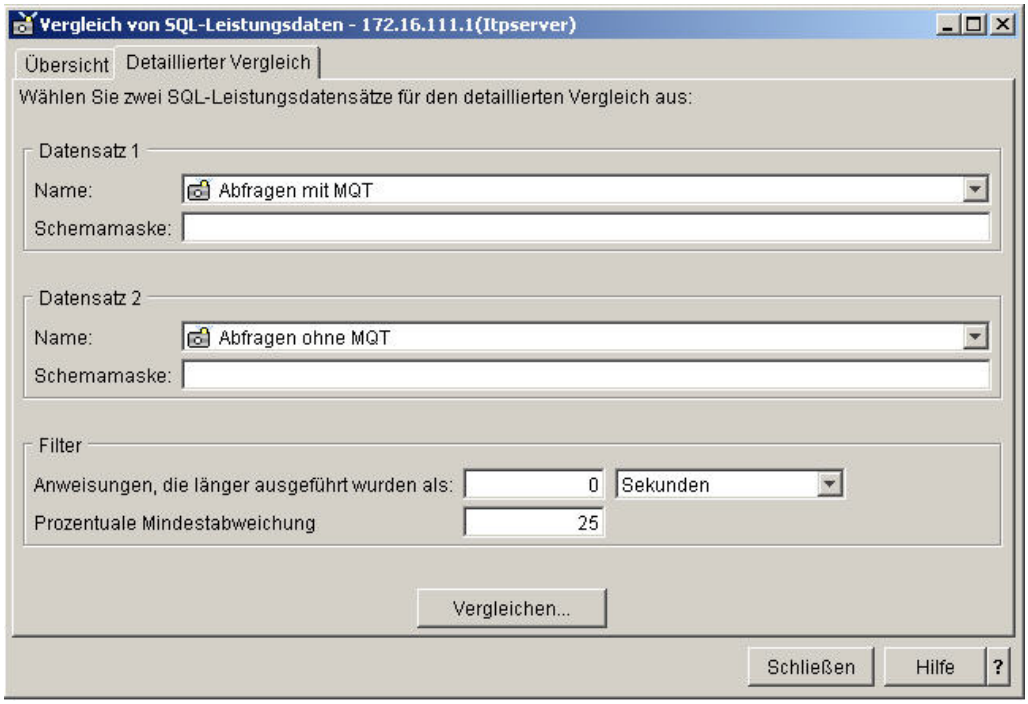
Daraufhin öffnet sich ein weiterer Dialog:



Snapshot vergleichen II

Der Dialog besteht aus zwei Registern. Das Register „Übersicht“ zeigt eine Übersicht der beiden Snapshots – ähnlich einem detaillierten Performance Monitor. Die Beschreibung der einzelnen Einträge finden Sie im Kapitel 7.4.3, so dass ich an dieser Stelle auf eine erneute Erläuterung verzichten möchte. Falls bei Ihnen dieses Register nicht angezeigt wird, liegt dies am Releasestand. IBM hat diesen Dialog im Release V6R1 geringfügig verändert.

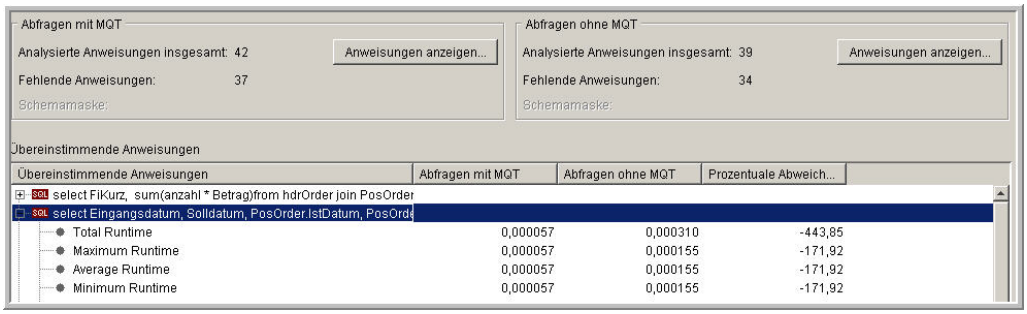
Wichtiger erscheinen mir in diesem Zusammenhang ohnehin die Angaben im Register „Detaillierter Vergleich“:



Detaillierter Snapshotvergleich

Sie haben hier die Möglichkeit eine Schemamaske einzutragen. Die Schemamaske dient dazu, SQL Anweisungen zu vergleichen, die in unterschiedlichen Schemata ausgeführt wurden. Stellen Sie sich folgendes Szenario vor. Eine Anwendung wurde optimiert und wird in einem Testschema ausgeführt. Sie verschieben die Applikation jetzt in das Produktionsschema und möchten ihre Ausführung dort vergleichen. Die Anweisungen in dem Vergleich sind identisch; der einzige Unterschied besteht darin, dass die Anweisungen in der Testumgebung das Schema „TEST“ verwenden, und die Anweisungen in der Produktionsumgebung das Schema „PROD“. Mit der Schemamaske können Sie „TEST“ in der ersten Überwachung und „PROD“ in der zweiten Überwachung ignorieren, so dass die Anweisungen in den beiden Überwachungen identisch wirken. In dem Dialog „Detaillierter Vergleich“ finden Sie außerdem einen Filter. Die Option „Anweisungen vergleichen, die länger ausgeführt wurden als“ bestimmt die Mindestlaufzeit für Anweisungen, die verglichen werden sollen. Mit dem Filterkriterium bestimmen Sie den prozentualen Mindestunterschied bei Schlüsselattributen der beiden Anweisungen, die verglichen werden. Ich benötige keine Filterkriterien und möchte alle Anweisungen vergleichen. Aus diesem Grund ändere ich die Mindestlaufzeit auf 0 Sekunden ab. Anschließend starte ich – durch Mausklick auf den Button „Vergleichen“ – den Monitorvergleich.

Die Analyse zeigt übersichtlich die betroffenen Ausführungspläne sowie die Laufzeitunterschiede im direkten Vergleich, wenn Sie einen der Einträge erweitern.



Anzeige der übereinstimmenden Anweisungen

Anhand der Laufzeitinformationen kann man jetzt entscheiden, ob sich der zusätzliche Aufwand der MQTs rechtfertigen lässt.

Bei aller Euphorie sollten Sie aber folgendes bedenken: Sie müssen berücksichtigen, dass Ihre Daten jetzt redundant gespeichert werden und entsprechenden Platz auf der Festplatte belegen. Aus diesem Grund sollten Sie von Zeit zu Zeit prüfen, ob und wie häufig Ihre erstellten MQTs vom Optimizer verwendet werden. Hierfür öffnen Sie das Kontextmenü einer entsprechenden Tabelle und wählen den Eintrag „Tabellen für gespeicherte Abfrage anzeigen“.

Informationen – MQT-Verwendung

Wie für Indizes werden auch für MQTs viele Informationen angezeigt. MQTs, deren Nutzungszähler längere Zeit „Null“ ist, müssen Sie prüfen. Entweder es fehlen geeignete Indizes oder die MQT ist einfach überflüssig und kann gelöscht werden. MQTs lassen sich an vielen Stellen mit Indizes vergleichen. Sie können die Laufzeiten der SQL Abfragen drastisch verbessern, benötigen aber zusätzliche Systemressourcen für die Speicherung und Pflege. Achten Sie also sehr sorgfältig bei der Erstellung der MQTs darauf, dass Sie sich später als nützlich erweisen.

7.3.7.4 Bestehende Tabellen als MQTs nutzen

Wenn Sie jetzt MQTs in Ihrer Umgebung einsetzen wollen, dann müssen Sie diese aber nicht unbedingt neu erstellen. Stellen wir uns einmal vor, in unserer Anwendung gäbe es bereits physische Dateien, die komprimierte Daten enthalten, dann wäre es unsinnig diese Dateien alle neu zu erstellen. Besser, wir nutzen die bestehenden Dateien als Basistabellen für eine MQT. Diese Dateien können im Übrigen durchaus mit DDS erzeugt worden sein. Dieser Sachverhalt ist unerheblich. Stellen wir uns das folgende Szenario vor:

Es existiert bereits die mit DDS erzeugte physische Datei „ORDERSUM“. Die Datei enthält die kumulierten Auftragseingänge je Monat und Jahr.

```
...A.Name++++++RLängeDDsF.....
**** Datenanfang ***************
     R S_JM_FMT
       JAHR            4  0
       MONAT           2  0
       EINGANG        11  2
     K JAHR
     K MONAT
```

Die Datei „ORDERSUM“

Die Datei wurde auch bereits über entsprechende HLL Programme mit Daten versorgt und wird auch sonst vielfach in den HLL Programmen für entsprechende Reports genutzt.

```
7.08.09
Programm: SUM_REPORT
-------------------------------------
  JAHR     MONAT     BETRAG(EUR)
  2007      4            4.120,00
  2007      5            8.240,00
  2007     10           21.879,00
  2007     11           42.704,00
  2008      2           11.663,00
  2008      4          860.600,00
  2008      5           11.590,00
  2008     10            6.350,00
  2008     11            8.814,00
  2009      4           50.902,00
  2009      5           33.880,00
  2009      6            8.714,00
```

Die Datei wird in bestehenden HLL Programmen verwendet

7.3.7
Seite 30

Ich möchte jetzt, dass die physische Datei „ORDERSUM“ auch als MQT im SQL Umfeld verwendet wird. Dafür muss ich zur bestehenden Datei lediglich das entsprechende SQL-Statement und die MQT-Klauseln hinzufügen. Der entsprechende SQL-Befehl ist einfach:

```
ALTER TABLE ITPSQL.ORDERSUM
ADD MATERIALIZED QUERY
(

SELECT
YEAR(EINGANGSDATUM) , MONTH(EINGANGSDATUM) , SUM(ANZAHL *
BETRAG)

FROM ITPSQL.HDRORDER
JOIN ITPSQL.POSORDER POSORDER USING ( HDKURZ )
JOIN ITPSQL.WERT ON POKURZ = HD_PO_KURZ

GROUP BY YEAR(EINGANGSDATUM), MONTH(EINGANGSDATUM)
ORDER BY YEAR(EINGANGSDATUM) , MONTH(EINGANGSDATUM)
)

       DATA INITIALLY DEFERRED
       REFRESH DEFERRED
       MAINTAINED BY USER
       ENABLE QUERY OPTIMIZATION ;
```

Natürlich muss das SELECT-Statement die gleiche Anzahl Spalten haben wie die Basistabelle und auch die Datentypen müssen kompatibel sein. Bzgl. der MQT-Klauseln habe ich mich dieses Mal entschieden, die Daten nicht bei der Erstellung zu aktualisieren. Schließlich handelt es sich um eine bestehende physische Datei, die bereits mit Daten versorgt wurde, aber die Bedeutung und Wirkungsweise der Klauseln kennen Sie ja bereits.

Und falls Sie den umgekehrten Weg gehen wollen und aus einer MQT wieder eine ganz „normale“ physische Datei machen wollen, ist auch dies kein Problem:

```
ALTER TABLE ORDERSUM
DROP MATERIALIZED QUERY
```

Die Klausel „DROP MATERIALIZED TABLE“ entfernt das SQL-Statement aus dem Dateiobjekt. Das Ergebnis ist eine physische Datei.

Bleibt die Frage zu klären: „Wie verhalten sich die bestehenden HLL-Programme, die die Datei „ORDERSUM“ verwenden?“ Die Anweisung „ALTER TABLE“ fügt lediglich das SQL-Statement in das Dateiobjekt ein; alle anderen Attribute bleiben erhalten – so auch beispielsweise der ursprünglich verwendete Satzformatname oder die Umwandlungsoption „Satzformat-Aktualitätsprüfung (LVLCHK)“. Wurde die Basistabelle „ORDERSUM“ mit der Option LVLCHK *NO erzeugt, werden Ihre RPG- oder Cobol-Programme gar nichts von der Änderung mitbekommen und wie bisher arbeiten. Wurde die Basistabelle hingegen ursprünglich mit der Option „Satzformat-Aktualitätsprüfung (LVLCHK)“ *YES erstellt, müssen Sie Ihre RPG- und Cobol-Programme neu wandeln, da durch die Anweisung „ALTER TABLE“ die Datei-ID natürlich verändert wird. Weitere Anpassungen im Quelltext der HLL-Programme sind aber nicht erforderlich. Sie sehen, die „alte“ Welt muss nicht zwingend verändert werden, wenn Sie neue SQL-Techniken und SQL-Objekte zukünftig nutzen wollen.

Ein letztes Problem muss noch geklärt werden: „Was passiert, wenn Sie die Datei „ORDERSUM“ neu erstellen müssen? Es ist nicht mehr ausreichend die DDS-Sourcen zu kompilieren, da den DDS-Sourcen die benötigten MQT-Einstellungen fehlen. Es gibt und wird – auch im zukünftigen Release – keine DDS-Anweisungen für MQTs geben. Wir brauchen daher eine schnelle, praktikable Lösung für die Dateierstellung. Am einfachsten wäre es natürlich von DDS- auf SQL-Sourcen umzustellen. Dieser Schritt sollte allerdings sorgfältig geplant werden. Eine weitere Möglichkeit ist die Integration eines SQL-Skriptes in Ihre bisherigen CL-Programme, denn vermutlich nutzen Sie derzeit CL-Programme, welche die Dateierstellungsbefehle enthalten. Im CL-Programm werden Sie sicherlich zunächst eine Kopie der Datei erstellen, um die Daten zu sichern, anschließend die Datei neu erstellen und im letzten Schritt die Daten zurück kopieren. Ihr CL-Programm könnte folgendermaßen aussehen:

```
PGM

        CRTDUPOBJ  OBJ(ORDERSUM) FROMLIB(ITPSQL) OBJTYPE(*FILE) +
                     TOLIB(QTEMP) DATA(*YES) FILEID(*YES)
        DLTF       FILE(ORDERSUM)
        CRTPF      FILE(ORDERSUM) LVLCHK(*NO)
        CPYF       FROMFILE(QTEMP/ORDERSUM) +
                     TOFILE(ITPSQL/ORDERSUM) MBROPT(*ADD) +
                     FMTOPT(*MAP)
ENDPGM
```

Das „alte“ CL-Programm

7.3.7
Seite 32

Nun könnten wir ja einfach einen „RUNSQLSTM“ Befehl hinzufügen, um nach der Dateierstellung die entsprechenden MQT-Einstellungen hinzuzufügen. Dazu hinterlegen wir das „ALTER TABLE“-Statement einfach in einer beliebigen Teildatei:

```
ALTER TABLE ITPSQL.ORDERSUM
ADD MATERIALIZED QUERY
(
 SELECT
 YEAR(EINGANGSDATUM),
 MONTH(EINGANGSDATUM),
 SUM(ANZAHL * BETRAG)

 FROM ITPSQL.HDRORDER
 JOIN ITPSQL.POSORDER POSORDER USING ( HDKURZ )
 JOIN ITPSQL.WERT ON POKURZ = HD_PO_KURZ
 GROUP BY YEAR(EINGANGSDATUM), MONTH(EINGANGSDATUM)
 ORDER BY YEAR(EINGANGSDATUM) , MONTH(EINGANGSDATUM)
)

 DATA INITIALLY DEFERRED
 REFRESH DEFERRED
 MAINTAINED BY USER
 ENABLE QUERY OPTIMIZATION
```

Das „ALTER TABLE“ Statement

Welche Quellendatei Sie hierfür verwenden, ist unerheblich. Ich nenne diese Quellendatei meistens QSQLSRC. Auch die Teildateiart ist irrelevant. Es bietet sich allerdings die Teildateiart SQL an. Aber Sie können die Statements sogar im IFS ablegen und dann trotzdem im CL-Programm ausführen. Um das SQL-Statement auszuführen, verwenden Sie einfach den Befehl RUNSQLSTM im CL-Programm. Auch ich erweitere jetzt mein ursprüngliches CL-Programm um den entsprechenden RUNSQLSTM-Befehl:

```
PGM

        CRTDUPOBJ  OBJ(ORDERSUM) FROMLIB(ITPSQL) OBJTYPE(*FILE) +
                     TOLIB(QTEMP) DATA(*YES) FILEID(*YES)
        DLTF       FILE(ORDERSUM)
        CRTPF      FILE(ORDERSUM) LVLCHK(*NO)
        CPYF       FROMFILE(QTEMP/ORDERSUM) +
                     TOFILE(ITPSQL/ORDERSUM) MBROPT(*ADD)
        RUNSQLSTM  SRCFILE(QSQLSRC) SRCMBR(C_ORDERSUM) +
                   COMMIT(*NC) NAMING(*SQL)
ENDPGM
```

Das modifizierte CL-Programm

Sie sehen, ich nutze den Befehl RUNSQLSTM um die entsprechende „ALTER TABLE“-Anweisung auszuführen. Da ich die SQL-Namenskonventionen im SQL Statement verwende und keine COMMIT-Steuerung wünsche, ersetze ich noch die entsprechenden Optionen und hoffe, dass nun alles wie zuvor fehlerfrei ausgeführt wird. Doch dies ist ein Irrtum, wie die folgenden Fehlermeldungen zeigen:

```
_7>> CALL ITPSQL/C_ORDERSUM
     Basisdatei HDRORDER nicht in Bibliothek QTEMP.
     Basisdatei POSORDER nicht in Bibliothek QTEMP.
     Basisdatei WERT nicht in Bibliothek QTEMP.
     Datei ORDERSUM in QTEMP kann nicht dupliziert werden.
     0 Objekte dupliziert. 1 Objekte nicht dupliziert.
```

Die Fehlermeldungen

Da es sich bei der Datei „ORDERSUM“ jetzt um eine MQT handelt, kann ich das Objekt nicht mehr ohne weiteres duplizieren. Ich müsste zusätzlich die Basistabellen kopieren, die ich im zugehörigen SELECT-Statement anspreche. Dies wäre sicherlich eine Variante. Ich finde allerdings folgende Lösung besser:

Ich verzichte im CL-Programm auf die Dateikopie und setze dafür im „ALTER TABLE“-Statement die Option „DATA INITIALLY“ wieder auf IMMEDIATE. Jetzt gehen zwar zunächst die Daten verloren, werden aber neu generiert, wenn die MQT erstellt wird.

Das CL-Programm:

```
PGM
        Der Befehl CRTDUPOBJ entfällt
        CRTPF      FILE(ORDERSUM) LVLCHK(*NO)
        RUNSQLSTM  SRCFILE(QSQLSRC) SRCMBR(C_ORDERSUM) +
                   COMMIT(*NC) NAMING(*SQL)
ENDPGM
```

Die Teildatei C_ORDERSUM

```
ALTER TABLE ITPSQL.ORDERSUM
ADD MATERIALIZED QUERY
(
 SELECT

)

 DATA INITIALLY IMMEDIATE
 REFRESH DEFERRED
 MAINTAINED BY USER
 ENABLE QUERY OPTIMIZATION
```

Die Lösung

7.3.7
Seite 34

Alternativ können Sie natürlich auch das „ALTER TABLE“-Statement über eine Query-Manager-Abfrage ausführen lassen. In diesem Fall ersetzen Sie einfach den Befehl „RUNSQLSTM“ durch den folgenden Befehl:

```
STRQMQRY QMQRY(ITPSQL/CRT_ORDSUM) NAMING(*SQL)
```

Es ist egal, ob Sie sich für den RUNSQLSTM-Befehl oder für eine Query-Abfrage entscheiden. Wichtig ist lediglich, dass Sie ggf. Ihre CL-Programme modifizieren müssen, um das neue Dateiobjekt zu erstellen. So ganz ohne Modifikationen können die neuen SQL-Objekte anscheinend doch nicht integriert werden. Vielleicht denken Sie doch einmal darüber nach, ob es nicht sinnvoll ist, die alten DDS-Sourcen durch SQL-Syntaxen abzulösen? Sie sollten mit der Umstellung nicht mehr zu lange warten, da IBM DDS stabilisiert hat. Stabilisiert bedeutet in diesem Zusammenhang, dass sich „kein IBM Entwickler mehr um DDS kümmert“. Es gibt keine Weiterentwicklung und auch Fehler werden nicht mehr behoben. Warum also sollten Sie sich um DDS kümmern?

7.3.8 Der OmniFind Text Search Server für DB2

OmniFind ist eine Plattform für die Suche in unstrukturierten Texten innerhalb oder außerhalb der DB2 in Dokumenten im integrierten File-System, in Spoolfiles und in Source Physical Files.

Mit OmniFind kann danach gesucht werden, ob bestimmte Suchausdrücke in einem Text vorhanden sind und welche Relevanz die Begriffe in den Texten haben. OmniFind nutzt die SQL-Funktionen CONTAINS und SCORE. Mit CONTAINS wird der Suchausdruck definiert, während die Funktion SCORE die Wertigkeit ermittelt, wie oft der Suchausdruck in den zu durchsuchenden Daten vorkommt.

Damit können folgende Anwendungen realisiert werden:

- Suche in Programmsourcen,
- Suche in Spoolfiles,
- Suche in DB2- und IFS-Daten, die miteinander in Verbindung stehen, z. B. Personaldaten mit Lebensläufen, Kundendaten mit Besuchsberichten und Artikeldaten mit QS-Daten,
- Suche in Dokumenten im integrierten File-System,
- Suche in XML-Daten,
- Suche in Textfeldern der DB2-Datenbank.

Der Suchausdruck wird mit CONTAINS definiert. Die Suche kann interaktiv oder mit embedded SQL durchgeführt werden.

Vorteile der Suche mit OmniFind:

- Die Suche mit OmniFind ist durch spezielle Indizes sehr schnell.
- Groß- und Kleinschreibung spielen keine Rolle.
- Umlaute spielen keine Rolle, z. B. die Suche nach „Müller“ findet auch „Mueller“.
- OmniFind kann mit Wildcards suchen, z. B. Wasser* findet alle Begriffe, die mit „Wasser“ beginnen.
- Es kann nach Synonymen gesucht werden, z. B. die Suche nach „Auto“ findet auch „Volvo“.
- OmniFind kommt mit vielen verschiedenen Datenformaten zurecht: Text, PDF, BLOB, CLOB, XML usw.
- OmniFind arbeitet mit einfachen Verknüpfungen: AND, OR, NOT.
- OmniFind ist kostenlos für die IBM i.

Ergänzung 36/2016

7.3.8
Seite 2

Beispiel für einen Suchausdruck mit OmniFind:

```
Select * from Artikel
where contains(Text,'wasserdicht AND luftdurchlässig - gelb') = 1
and Produktgruppe = 50
order by score(Text,'wasserdicht');
```

Es werden alle Artikel gesucht, die wasserdicht, luftdurchlässig und nicht gelb sind sowie der Produktgruppe 50 angehören. Die Sortierung erfolgt nach der Relevanz des Begriffs „wasserdicht“.

Beispiel für die OmniFind-Suche mit embedded SQL:

```
Dcl-s suchausdruck varchar(100);
EXEC SQL DECLARE C1 CURSOR FOR
Select Id, Bezeichnung from artikel
where contains(Text, :suchausdruck) = 1 and Produktgruppe = 50;
EXEC SQL SET :suchausdruck = 'wasserdicht NOT gelb';
EXEC SQL OPEN C1;
dou SQLCOD = 0;
if SQLCOD = 0;
EXEC SQL Fetch C1 into :Id, :Bezeichnung;
endif;
enddo;

EXEC SQL CLOSE C1;
```

OmniFind-Suchausdruck im embedded SQL

7.3.8.1 Installation

Der OmniFind Text Search Server entspricht dem kostenlosen Lizenzprogramm 5733OMF, das über den Electronic Software Service heruntergeladen und über das Menü LICPGM installiert werden kann. Weitere Informationen sind im Handbuch enthalten.

7.3.8.2 OmniFind Starten und Stoppen

Das Starten und Stoppen des OmniFind Text Search Server geschieht in der SQL-Umgebung mit den Befehlen:

- CALL SYSPROC.SYSTS_START
- CALL SYSPROC.SYSTS_STOP

Während der Installation wird standardmäßig ein Text Search Server eingerichtet. Es besteht die Möglichkeit, mehrere Text Search Server einzurichten und separat zu starten oder zu stoppen. Die Einträge befinden sich in der Tabelle QSYS2.SYSTEXTSERVERS.

7.3.8.3 Der OmniFind-Textsuchindex

OmniFind verwendet für die Indizierung der Daten einen eigenen Textsuchindex. Die Verwaltung des Textsuchindexes wird mit folgenden Befehlen durchgeführt:

- CALL SYSPROC.SYSTS_CREATE – Erstellt einen Textsuchindex. Gleichzeitig werden drei Trigger für Einfügen, Ändern und Löschen der Daten für den Textsuchindex erstellt. Bitte beachten Sie, dass nach dem CREATE noch keine Daten im Index enthalten sind. Dem CREATE-Befehl können mehrere Parameter mitgegeben werden, z. B. die Frequenz in Zeiteinheiten (Std./Min.), wie oft der Index aktualisiert werden soll.
- CALL SYSPROC.SYSTS_UPDATE – Wenn der Index leer ist, wird er mit Daten befüllt. Wenn bereits Daten im Index vorhanden sind, wird der Index aktualisiert.
- CALL SYSPROC.SYSTS_DROP – Löschen des Textsuchindexes. Damit werden der Index und die drei Trigger gelöscht.

7.3.8.4 Unterstützte Datentypen des Textsuchindexes

OmniFind unterstützt eine ganze Reihe von Datentypen, mit denen ein Textsuchindex aufgebaut werden kann.

Datentypen innerhalb der DB2-Tabellen

- CHAR
- VARCHAR
- CLOB
- BLOB
- DBCLOB
- GRAPHIC
- VARGRAPHIC
- BINARY
- VARBINARY
- XML

Datentypen außerhalb der DB2-Tabellen

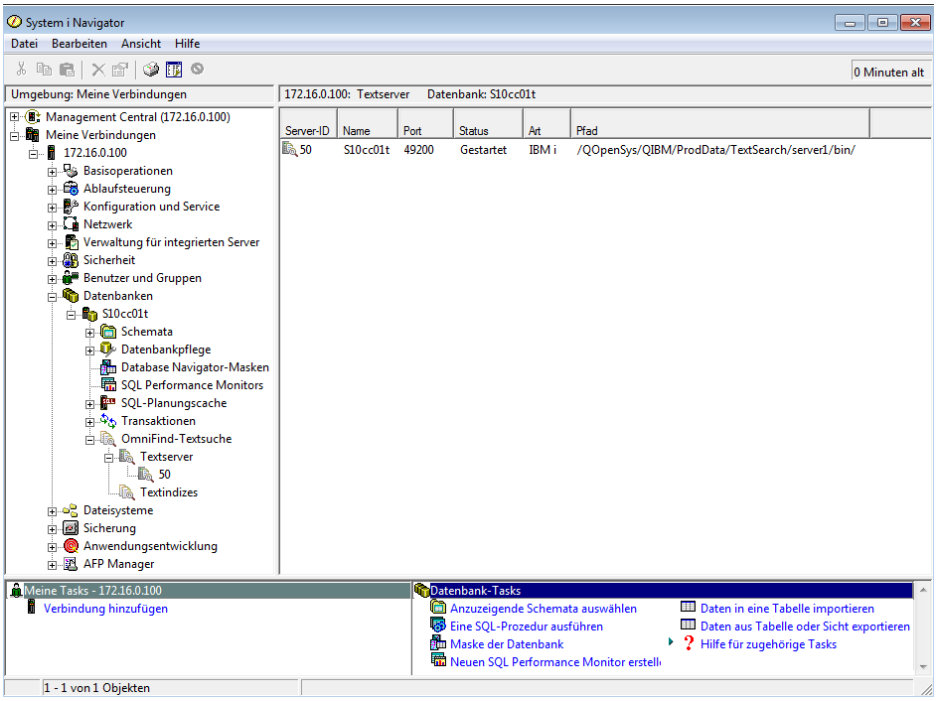
- Spoolfiles
- TEXT: Unstrukturierter Text
- HTML: Hypertext Markup Language
- XML: Extensible Markup Language
- INSO: Der OmniFind Text Search Server für DB2 for i verwendet Filter, um das Format von Textdokumenten festzustellen. Die folgenden INSO-Dokumentformate werden unterstützt:
  - XML,
  - HTML,
  - JustSystems Ichitaro,
  - Lotus 123,
  - Lotus Freelance,
  - Lotus WordPro,
  - Microsoft Excel,
  - Microsoft PowerPoint,
  - Microsoft Rich Text Format,
  - Microsoft Visio,
  - Microsoft Word,
  - Microsoft Write,
  - Portable Document Format (PDF),
  - Quattro Pro,
  - Rich Text RTF,
  - StarOffice Calc und OpenOffice Calc.

Ergänzung 36/2016

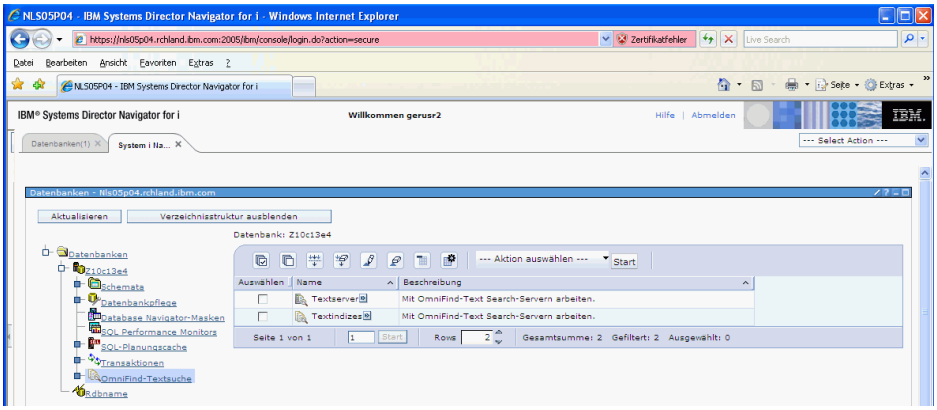
7.3.8.5 OmniFind-Verwaltungs-Tools

Zur Verwaltung des OmniFind Text Search Server wie auch der Textsuchindizes sind drei Möglichkeiten vorgesehen:

- direkte Eingabe der OmniFind-Befehle in der SQL-Umgebung,
- mit dem System-i-Navigator,



- mit dem IBM Navigator for i.



7.3.8.6 OmniFind-Suchoperatoren

Für die Funktionen CONTAINS und SCORE können folgende Suchoperatoren verwendet werden:

Operator	Beispiel	Kommentar
AND	Müller Schmidt Müller AND Schmidt „Hans Müller" AND Schmidt	Wenn zwischen Begriffen kein AND-Operator angegeben ist, wird automatisch AND verwendet.
OR	Müller OR Schmidt	Findet „Müller" oder „Schmidt".
" " Exakte Übereinstimmung	„Die schöne Müllerin" „Hans Müller"	Exakte Übereinstimmung bei mehreren Begriffen. Findet auch „Hans Mueller".
* Wildcard	Pool* *pool P*l	Poolbillard oder Poolbar Innenpool oder Außenpool Pool oder Paul
NOT (-)	Müller NOT Schmidt Müller – Schmidt	Findet Ergebnisse in Texten, die „Müller", „Mueller", aber nicht „Schmidt" enthalten.
^ Relevanz	Hans^5 Heinrich	Suchergebnisse, die „Hans" häufiger als „Heinrich" enthalten, werden in den Suchergebnissen weiter vorn dargestellt.

7.3.8.7 Suche mit Synonymen

OmniFind bietet die Möglichkeit, Texte mit Synonymen zu durchsuchen. Im nachfolgenden Beispiel wird genau erklärt, wie das funktioniert.

Ein Kunde sucht in einem Webshop eine Jacke mit der Eigenschaft „regenfest". In der Artikelbeschreibung ist die Eigenschaft „wasserdicht" angegeben. Der Kunde findet keine Artikel, und das Geschäft kommt nicht zustande. In diesem Fall ist die Synonymsuche genau das Richtige.

Da diese Synoyme firmenspezifisch sind, hat OmniFind die Möglichkeit geschaffen, diese im XML-Format selbst zu definieren und dem Textsuchindex hinzuzufügen.

Damit OmniFind weiß, dass nach Synonymen gesucht wird, gibt es eine Option zum Befehl CONTAINS: SYNONYM=ON. Der Befehl sieht wie folgt aus:

```
Select * from Tabelle
where (text,'regenfest','SYNONYM=ON') = 1;
```

Suchbeispiel mit der Synonymoption

Vorgehensweise:

1. In einem Datenbankfeld befindet sich folgender Text:

   Jemand musste Josef K. verleumdet haben, denn ohne dass er etwas Böses getan hätte, wurde er eines Morgens verhaftet, als er mit dem Maserati zur Arbeit fuhr …

2. Die Suche soll einen Treffer landen, wenn mit folgendem Begriff gesucht wird:

   „mit dem Auto zur Firma fuhr"

3. Definition der Synonyme. Diese müssen OmniFind im XML-Format zur Verfügung gestellt werden:

```
<?xml version="1.0" encoding="UTF-8"?>
<synonymgroups version="1.0">
     <synonymgroup>
        <synonym>Auto</synonym>
        <synonym>PKW</synonym>
        <synonym>Maserati</synonym>
     </synonymgroup>
     <synonymgroup>
        <synonym>Arbeit</synonym>
        <synonym>Arbeitsort</synonym>
        <synonym>Firma</synonym>
     </synonymgroup>
</synonymgroups>
```

Tipp:

Zum Erstellen der XML-Datei aus einer DB2-Tabelle gibt es die SQL-Funktionen XMLSERIALIZE, XMLGROUP und XMLDECLARATION.

4. Hinzufügen der Synonyme zum Textindex. Hier gibt es zwei Möglichkeiten: entweder mit dem IBM Navigator for i oder mit dem Import-Tool, das über die Qshell aufgerufen werden kann. Die Beschreibung befindet sich im Handbuch „OmniFind Text Search Server für DB2 for i“.

7.3.8.8 Daten aus dem IFS für die OmniFind-Suche einbinden

Es ist eine interessante Möglichkeit von OmniFind, auf Daten zuzugreifen, die sich im IFS befinden. Auf diese Weise kann eine Suche in Handbüchern realisiert werden, die als PDF-Dokumente vorliegen. Eine weitere Anwendung ist die Verbindung von Daten in der Datenbank mit IFS-Dokumenten. Ein Anwendungsfall ist die Verknüpfung von Personaldaten aus der Datenbank mit den Lebensläufen, die als PDF oder Word-Dokument gespeichert sind. Es ist ebenso möglich, Kundendaten und Besuchsberichte oder Artikeldaten mit QS-Berichten zu verknüpfen. Voraussetzung ist die Zuordnung des IFS-Dokuments zum Schlüssel in der DB2-Tabelle.

In folgendem Beispiel geht es um die Indexierung von IFS-Dokumenten. In diesem Fall handelt es sich um IBM-Handbücher, die als PDF-Dokumente vorliegen. Im Verzeichnis /tmp/Manuals sind folgende Dokumente vorhanden:

Name	Änderungsdatum	Typ	Größe
DB2 and XML Redbook.pdf	28.10.2013 23:54	Adobe Acrobat Document	3.023 KB
DB2 SQL Reference V7R2.pdf	16.09.2015 20:36	Adobe Acrobat Document	7.247 KB
RPG Programming Reference V7R2.pdf	08.10.2015 19:39	Adobe Acrobat Document	7.186 KB

Handbücher im Verzeichnis /tmp/Manuals im PDF-Format

Zuerst werden zwei Tabellen erstellt, die die Pfadeinträge enthalten. Anstelle von „lib“ geben Sie bitte Ihren eigenen Bibliotheksnamen an.

```
create table lib.ifsdocs(
       idpath     char(2000)
);

Label on Table  ifsdocs is 'Tabelle IFS-Dokumente';


create table lib.ifssuche(
       isid       int
        generated always as identity (start with 1 increment by 1),
       ispath     varchar(2000),
       primary    key(isid)
);

Label on Table  ifssuche is 'Tabelle IFS-Suche';
```

Erstellen der Tabellen IFSDOCS und IFSSUCHE

Achtung:

Das Laden der Dokumentinhalte geschieht mit der SQL-Funktion GET_BLOB_FROM_FILE(Pfadname). Der Pfadname darf keine Blanks enthalten. Deshalb ist das Feld ISPATH in der Tabelle IFSSUCHE mit variabler Länge definiert.

Das Lesen der Dokumente geschieht mit dem Qshell-Befehl „find <Pfad> -type f“, der über ein CL aufgerufen wird und den Inhalt des Verzeichnisses in die Tabelle IFSDOCS schreibt. Die Option „–type f“ listet nur File-Namen auf und keine Verzeichnisse.

```
PGM

DCL        VAR(&DIR)  TYPE(*CHAR) LEN(050)
DCL        VAR(&CMD)  TYPE(*CHAR) LEN(100)
DCL        VAR(&FILE) TYPE(*CHAR) LEN(010) VALUE(IFSDOCS)

DCLF       FILE(IFSDOCS)

CLRPFM     FILE(&FILE)

OVRDBF     FILE(STDOUT) TOFILE(&FILE)

CHGVAR     VAR(&DIR) VALUE('/tmp/Manuals')
CHGVAR     VAR(&CMD) VALUE('find' *BCAT &DIR *BCAT '-type f')

STRQSH     CMD(&CMD)

DLTOVR     FILE(STDOUT)

ENDPGM
```

CL zum Auslesen eines IFS-Verzeichnisses und Schreiben der Ergebnisse in die Tabelle IFSDOCS

Mit der folgenden SQL-Anweisung werden Daten in die Tabelle IFSSUCHE kopiert:

```
insert into ifssuche (ispath) select trim(idpath) from ifsdocs;
```

Kopieren der Daten von der Tabelle IFSDOCS nach IFSSUCHE

7.3.8
Seite 12

Mit dem folgenden SQL-Statement wird der Suchindex erstellt:

```
call sysproc.systs_create(
 'lib',
 'ifsindex',
 'lib.ifssuche(get_blob_from_file(ispath))',
 'format inso'
);
```

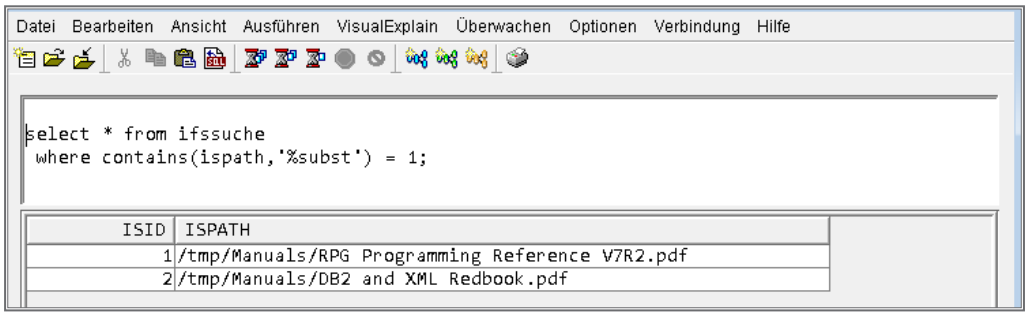
Suchindex erstellen

Die Aktualisierung des Suchindexes geschieht mit dem SQL-Statement.

```
call sysproc.systs_update('lib','ifsindex','using update minimum');
```

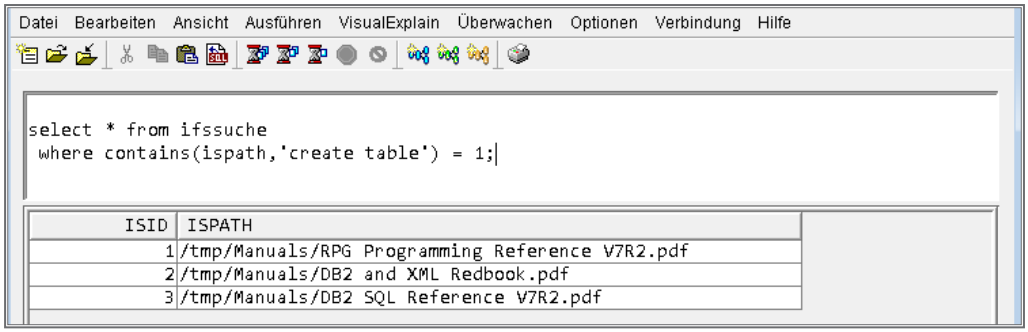
Suchindex aktualisieren

Jetzt können Abfragen auf den Inhalt der Handbücher durchgeführt werden. Als Erstes wird nach der RPG-Funktion „%subst“ gesucht.



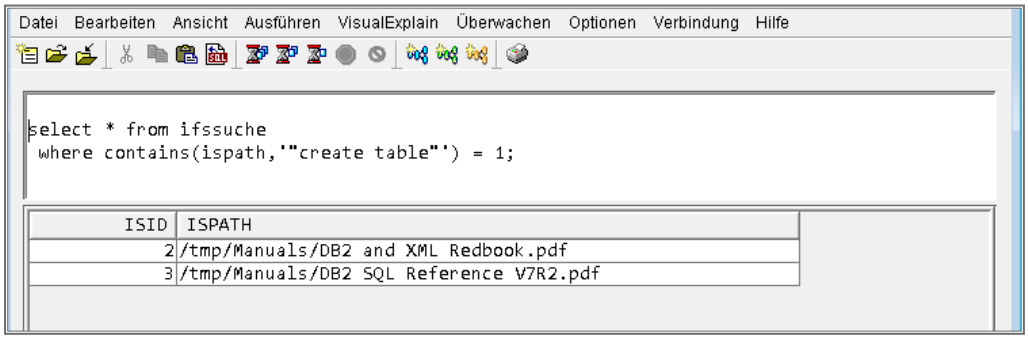
Suche nach „%subst“ in indexierten PDF-Handbüchern

Das nächste Beispiel sucht nach den Handbüchern, in denen die Begriffe „create“ und „table“ vorkommen.



Suche nach „create“ und „table“ in indexierten PDF-Handbüchern

Mit diesem Beispiel wird nach dem Begriff „create table“ gesucht. Bitte beachten Sie, dass für die exakte Suche der Begriff in doppelte Hochkommas gesetzt werden muss:



Exakte Suche nach „create table“ in indexierten PDF-Handbüchern

7.3.8.9 Textsuchgruppen

Eine Neuerung in V7R1 sind Textsuchgruppen. Sie werden verwendet, um Daten zu indexieren, die sich außerhalb der DB2 befinden:

- Spool-Daten,
- Daten im Integrated File System – IFS,
- physische Quellendateien mit mehreren Teildateien-Source Physical Files.

Ein praktischer Anwendungsfall ist das Erstellen einer Textsuchgruppe, die alle Sourcen eines Projekts oder alle Sourcen der IT-Abteilung enthält. Somit können auf einfache Weise und sehr schnell die Sourcen nach vorgegebenen Suchkriterien durchsucht werden.

Beispiel: Es werden alle Programmsourcen benötigt, die die Dateien „Kunden“ und „Artikel“ enthalten, nicht aber die Datei „Vertreter“.

```
call sourcen.search('kunden artikel -vertreter');
```

Programmsourcen der Textsuchgruppe „Sourcen“ durchsuchen

7.3.8.10 Textsuchgruppe für Source Physical Files erstellen

Die Textsuchgruppe wird mit dem SQL-Befehl SYSTS_CREATE_COLLECTION erstellt. Im folgenden Beispiel wird die Textsuchgruppe „Sourcen“ erstellt, die Quellen aus verschiedenen Bibliotheken enthalten soll.

```
CALL SYSTS_CREATE_COLLECTION('sourcen');
```

Erstellen der Textsuchgruppe „Sourcen“ für die Indexierung von Source Physical Files

Der Befehl erstellt die Bibliothek „SOURCEN“, die folgende Objekte enthält:

- Kataloge zum Aufzeichnen der Konfiguration der Objektgruppe,
- Kataloge zum Aufzeichnen der indexierten Objekte,
- gespeicherte SQL-Prozeduren zum Verwalten und Durchsuchen der Objektgruppe.

SYSTS_CREATE_COLLECTION wird mit der Berechtigung *EXECUTE ausgeliefert. Damit der Aufrufer den Befehl ausführen kann, muss er über Folgendes verfügen:

- Berechtigung zum Erstellen eines DB2-Schemas,
- Berechtigung zum Erstellen eines Textsuchindexes.

Nachdem die Textsuchgruppe erstellt worden ist, können mit dem SQL-Befehl ADD_SRCPF_OBJECT_SET die Source Physical Files hinzugefügt werden.

```
CALL sourcen.ADD_SRCPF_OBJECT_SET('TSTS','*ALLSRCPF');
CALL sourcen.ADD_SRCPF_OBJECT_SET('DEVS','QDDSSRC');
CALL sourcen.ADD_SRCPF_OBJECT_SET('TOOL','QRPGSRC');
```

Hinzufügen der Source Physical Files zur Textsuchgruppe „Sourcen“

Tipp:

Mit der Option *ALLSRCPF werden automatisch alle Source Physical Files einer Bibliothek dem Textsuchindex hinzugefügt.

Das Löschen eines Eintrags in die Textsuchgruppe geschieht mit dem Befehl RMV_SRCPF_OBJECT_SET oder mit REMOVE_OBJECT_SET.

QUERY_OBJECT_SET gibt den Inhalt und den Status der Textsuchgruppe aus. Da die Sourcen noch nicht indexiert sind, ist das Feld LASTREFRESHTIME noch leer.

```
call sourcen.QUERY_OBJECT_SET;
```

SETID	SETSTATE	LASTREFRESHTIME	ADDOBJSETSQL
3	ACTIVE	-	CALL "SOURCEN".ADD_SRCPF_OBJECT_SET('TSTS ','*ALLSRCPF')
4	ACTIVE	-	CALL "SOURCEN".ADD_SRCPF_OBJECT_SET('DEVS ','QDDSSRC ')
5	ACTIVE	-	CALL "SOURCEN".ADD_SRCPF_OBJECT_SET('TOOL ','QRPGSRC ')

Inhaltsanzeige der Textsuchgruppe „Sourcen“ mit QUERY_OBJECT_SET

Eine Suche ist jetzt noch nicht möglich. Mit dem SQL-Befehl UPDATE wird der Suchindex aufgebaut.

```
call sourcen.update;
```

Aufbau des Textsuchindexes der Textsuchgruppe „Sourcen“

Der QUERY_OBJECT_SET führt jetzt zu folgendem Ergebnis:

SETID	SETSTATE	LASTREFRESHTIME	ADDOBJSETSQL
3	ACTIVE	2015-11-08 17:10:08.772935	CALL "SOURCEN".ADD_SRCPF_OBJECT_SET('TSTS ','*ALLSRCPF')
4	ACTIVE	2015-11-08 17:10:09.495459	CALL "SOURCEN".ADD_SRCPF_OBJECT_SET('DEVS ','QDDSSRC ')
5	ACTIVE	2015-11-08 17:10:09.831206	CALL "SOURCEN".ADD_SRCPF_OBJECT_SET('TOOL ','QRPGSRC ')

Inhalt der Textsuchgruppe „Sourcen“ mit QUERY_OBJECT_SET

Das Feld LASTREFRESHTIME enthält nun den Zeitstempel der letzten Aktualisierung des Textindexes.

Jetzt kann in den Sourcen der Textsuchgruppe gesucht werden, z. B. alle Sourcen, die im Header NOMAIN enthalten, den Befehl %SCANRPL verwenden, aber nicht das Feld h#html enthalten dürfen.

```
call sourcen.search('nomain %scanrpl -h#html');
```

Suche in der Textsuchgruppe „Sourcen“

Die Suche liefert folgendes Ergebnis: Die Ergebnisse werden nach Score absteigend sortiert.

OBJTYPE	OBJATTR	CONTAINING_OBJECT_LIB	CONTAINING_OBJECT_NAME	OBJECTINFOR	MODIFY_TIME	SC
*SRCPF	*MBR	TOOL	QRPGSRC	<Source_Phys...	2015-09-09 15:32:22.000000	0.173
*SRCPF	*MBR	TOOL	QRPGSRC	<Source_Phys...	2015-08-07 15:10:20.000000	0.148
*SRCPF	*MBR	TOOL	QRPGSRC	<Source_Phys...	2014-11-13 00:35:29.000000	0.134
*SRCPF	*MBR	TOOL	QRPGSRC	<Source_Phys...	2014-10-06 23:05:45.000000	0.129
*SRCPF	*MBR	TOOL	QRPGSRC	<Source_Phys...	2014-11-29 15:06:37.000000	0.128
*SRCPF	*MBR	TOOL	QRPGSRC	<Source_Phys...	2014-11-14 16:49:14.000000	0.112
*SRCPF	*MBR	TOOL	QRPGSRC	<Source_Phys...	2015-11-02 18:17:40.000000	0.045

Suchergebnisliste mit den Ergebnisspalten

7.3.8
Seite 16

Ergebnisspalten des Suchergebnisses:

Bezeichnung	Format
OBJTYPE	CHAR(10) CCSID 1208
OBJATTR	CHAR(10) CCSID 1208
CONTAINING_OBJECT_LIB	CHAR(10) CCSID 1208
CONTAINING_OBJECT_NAME	CHAR(10) CCSID 1208
OBJECTINFOR	XML
MODIFY_TIME	TIMESTAMP
SCORE	DOUBLE

OBJTYPE: Der Objekttyp des Ergebnisses. In diesem Fall *SRCPF.

OBJATTR: Das Objektattribut des Ergebnisses. Hier ist es das *MBR eines *SRCPF.

CONTAINING_OBJECT_LIB: Die Bibliothek, die das Suchergebnis enthält.

CONTAINING_OBJECT_NAME: Der Objektname des Source Physical File.

OBJECTINFOR: Ein XML-String, der die Bibliothek, die SRCPF und den Member-Namen enthält. Der String hat folgendes Format:

```
<Source_Physical_File_Member xmlns="http://www.ibm.com/xmlns/prod/db2textsearch/obj1>
      <file_library>MYLIB</file_library>
      <file_name>MYPF</file_name>
      <member_name>teildatei1</member_name>
</Source_Physical_File_Member>
```

MODIFY_TIME: Der Zeitpunkt der letzten Änderung.

SCORE: Ein Wert zwischen 0 und 1, der darüber Auskunft gibt, wie oft der Suchausdruck im Suchergebnis vorhanden ist. Je höher dieser Wert ist, umso öfter wurde eine Übereinstimmung mit den Suchkriterien gefunden.

Hinweis:

Eine Textsuchgruppe kann IFS-Daten, Spool-Daten und Source-Physical-File-Daten enthalten. Es ist ebenso möglich, jeweils eine Textsuchgruppe zu erstellen, die nur IFS- oder SRCPF-Daten enthält.

7.3.8.11 Textsuchgruppe für Spools erstellen

Die Textsuchgruppe wird mit dem SQL-Befehl SYSTS_CREATE_COLLECTION erstellt. Im folgenden Beispiel wird die Textsuchgruppe „Spools“ erstellt.

```
CALL SYSTS_CREATE_COLLECTION('spools');
```

Erstellen der Textsuchgruppe „Spools“

Mit dem SQL-Befehl ADD_SPLF_OBJECT_SET werden die Spools zur Textsuchgruppe hinzugefügt. Als Parameter können angegeben werden:

- Bibliothek der Ausgabewarteschlange,
- Name der Ausgabewarteschlange,
- Benutzername,
- Jobname,
- Jobbenutzer,
- Jobnummer,
- Benutzerdaten,
- Anfangszeit,
- Endezeit,
- Ausgabesatz-id.

Im Beispiel werden drei Einträge in die Textsuchgruppe aufgenommen. Der erste Eintrag betrifft die Spools in der OUTQ PRT01 in der Bibliothek QUSRSYS, der zweite Eintrag die Spools in der QPRINTS. Der dritte Eintrag enthält alle Spools des Benutzers ROSS.

```
CALL spools.ADD_SPLF_OBJECT_SET('QUSRSYS','PRT01');
CALL spools.ADD_SPLF_OBJECT_SET('QGPL','QPRINTS');
CALL spools.ADD_SPLF_OBJECT_SET('','','ROSS');
```

Hinzufügen von Objektsätzen für Spoolfiles in die Textsuchgruppe „Spools“

Das Löschen eines Eintrags in die Textsuchgruppe geschieht mit dem Befehl RMV_SPLF_OBJECT_SET oder mit REMOVE_OBJECT_SET.

7.3.8
Seite 18

Mit QUERY_OBJECT_SET werden der Inhalt und der Status der Textsuchgruppe angezeigt. Da die Spools noch nicht indexiert sind, ist das Feld LASTREFRESHTIME noch leer.

```
call spools.query_object_set;
```

SETID	SETSTATE	LASTREFRESHTIME	ADDOBJSETSQL
1	ACTIVE	-	CALL "SPOOLS".ADD_SPLF_OBJECT_SET('QUSRSYS','PRT01','','','','','',NULL,NULL)
2	ACTIVE	-	CALL "SPOOLS".ADD_SPLF_OBJECT_SET('QGPL','QPRINTS','','','','','',NULL,NULL)
3	ACTIVE	-	CALL "SPOOLS".ADD_SPLF_OBJECT_SET('','','ROSS','','','','',NULL,NULL)

Anzeige des Inhalts der Textsuchgruppe „Spools“ mit QUERY_OBJECT_SET

Mit dem SQL-Befehl UPDATE wird der Suchindex aufgebaut.

```
call spools.update;
```

Aufbau des Textsuchindexes der Textsuchgruppe „Spools“

Der QUERY_OBJECT_SET führt jetzt zu folgendem Ergebnis:

SETID	SETSTATE	LASTREFRESHTIME	ADDOBJSETSQL
1	ACTIVE	2015-11-10 14:33:19.643406	CALL "SPOOLS".ADD_SPLF_OBJECT_SET('QUSRSYS','PRT01','','','','','',NU...
2	ACTIVE	2015-11-10 14:33:20.344674	CALL "SPOOLS".ADD_SPLF_OBJECT_SET('QGPL','QPRINTS','','','','','',NUL...
3	ACTIVE	2015-11-10 14:33:20.622356	CALL "SPOOLS".ADD_SPLF_OBJECT_SET('','','ROSS','','','','',NULL,NULL)

Inhalt der Textsuchgruppe „Spools“ mit QUERY_OBJECT_SET

Das Feld LASTREFRESHTIME enthält nun den Zeitstempel der letzten Aktualisierung des Textindexes.

Jetzt kann in den Spools der Textsuchgruppe gesucht werden, z. B. alle Spools, die die Begriffe „TCP/IP“ und „CCSID“ enthalten.

```
call spools.search('TCP/IP AND CCSID');
```

Suche in der Textsuchgruppe „Spools“

OBJTYPE	OBJATTR	CONTAINING_OBJECT_LIB	CONTAINING_OBJECT_NAME	OBJECTINFOR	MODIFY_TIME	SC
*OUTQ...	*SPLF...	QUSRSYS	PRT01	<Spool_File xmlns="http:/...	2015-03-23 12:32:45.000000	0.593
*OUTQ...	*SPLF...	QUSRSYS	PRT01	<Spool_File xmlns="http:/...	2015-08-19 14:03:29.000000	0.494
*OUTQ...	*SPLF...	QUSRSYS	PRT01	<Spool_File xmlns="http:/...	2015-08-19 14:06:12.000000	0.494
*OUTQ...	*SPLF...	QUSRSYS	PRT01	<Spool_File xmlns="http:/...	2015-08-19 14:11:14.000000	0.494
*OUTQ...	*SPLF...	QUSRSYS	PRT01	<Spool_File xmlns="http:/...	2015-02-23 12:15:07.000000	0.494
*OUTQ...	*SPLF...	QUSRSYS	PRT01	<Spool_File xmlns="http:/...	2015-03-23 12:29:32.000000	0.494

Ergebnis der Suche in der Textsuchgruppe „Spools“

Die Ergebnisspalten sind im Abschnitt „Textsuchgruppe für Source Physical Files erstellen“ beschrieben. Der Inhalt der Spalte OBJECTINFOR enthält jetzt einen XML-String, der die Informationen der gefundenen Spoolfiles enthält.

```
<Spool_File xmlns="http://www.ibm.com/xmlns/prod/db2textsearch/obj1">
      <job_name>QPADEV000C</job_name>
      <job_user_name>USERA</job_user_name>
      <spool_file_name>DSXSVRALS</spool_file_name>
      <spool_file_number>1</spool_file_number>
      <job_system_name>ZD21BP1</job_system_name>
      <create_date>1081027</create_date>
      <create_time>035554</create_time>
</Spool_File>
```

7.3.8.12 Textsuchgruppe für IFS-Dateien erstellen

Im Abschnitt 7.3.8.8 ist dargestellt, wie mit geringem Programmieraufwand IFS-Dateien indiziert werden können. Mithilfe einer Textsuchgruppe wird dieser Prozess vereinfacht.

Der Hauptunterschied ist, dass beim Hinzufügen von Verzeichnissen in die Textsuchgruppe keine Unterverzeichnisse angegeben werden können. Es muss jedes Verzeichnis einzeln angegeben werden.

Ein weiterer Vorteil der Textsuchgruppe ist, dass mit den SQL-Befehlen GET_OBJECT_STATUS und GET_OBJECTS_NOT_INDEXED gearbeitet werden kann, die komfortable Auskunftsmöglichkeiten über die Inhalte der Textsuchgruppe bieten.

Mit dem SQL-Befehl SYSTS_CREATE_COLLECTION wird eine Textsuchgruppe „Dokumente“ erstellt, die IFS-Dokumente enthalten soll. Die Formatangabe „INSO“ bewirkt, dass Dokumente im Rich-Text-Format, wie z. B. PDF-Dokumente, verarbeitet werden können.

```
CALL SYSTS_CREATE_COLLECTION('dokumente','FORMAT INSO');
```

Erstellen der Textsuchgruppe „Dokumente“

Mit dem SQL-Befehl ADD_IFS_STMF_OBJECT_SET werden die Verzeichnisse, in denen sich die Dokumente befinden, zur Textsuchgruppe hinzugefügt.

7.3.8
Seite 20

Im nachfolgenden Beispiel sind es zwei Verzeichnisse. Das Verzeichnis /tmp/Manuals enthält Handbücher im PDF-Format. Das Verzeichnis /tmp/Dictionary enthält ein Wörterbuch im Textformat.

```
CALL dokumente.ADD_IFS_STMF_OBJECT_SET('/tmp/Manuals');
CALL dokumente.ADD_IFS_STMF_OBJECT_SET('/tmp/Dictionary');
```

Hinzufügen von Objektsätzen für IFS-Daten in die Textsuchgruppe „Dokumente“

SETID	SETSTATE	LASTREFRESHTIME	ADDOBJSETSQL
1	ACTIVE	-	CALL "DOKUMENTE".ADD_IFS_STMF_OBJECT_SET('/tmp/Manuals/')
2	ACTIVE	-	CALL "DOKUMENTE".ADD_IFS_STMF_OBJECT_SET('/tmp/Dictionary/')

Ergebnisliste des Befehls „dokumente.GET_OBJECT_STATUS“

Das Löschen eines Eintrags in die Textsuchgruppe geschieht mit dem Befehl RMV_IFS_STMF_OBJECT_SET oder mit REMOVE_OBJECT_SET.

Mit dem SQL-Befehl UPDATE wird der Suchindex aufgebaut.

```
call dokumente.update;
```

Aufbau des Textsuchindexes der Textsuchgruppe „Dokumente“

Der QUERY_OBJECT_SET führt jetzt zu folgendem Ergebnis:

SETID	SETSTATE	LASTREFRESHTIME	ADDOBJSETSQL
1	ACTIVE	2015-11-10 18:39:19.756718	CALL "DOKUMENTE".ADD_IFS_STMF_OBJECT_SET('/tmp/Manuals/')
2	ACTIVE	2015-11-10 18:39:19.920186	CALL "DOKUMENTE".ADD_IFS_STMF_OBJECT_SET('/tmp/Dictionary/')

Inhalt der Textsuchgruppe „Dokumente“ mit QUERY_OBJECT_SET

Das Feld LASTREFRESHTIME enthält nun den Zeitstempel der letzten Aktualisierung des Textindexes.

Jetzt kann in den Dokumenten der Textsuchgruppe gesucht werden.

```
call dokumente.search('Performance OR Storage');
```

Suche in der Textsuchgruppe „Dokumente“

OBJTYPE	OBJATTR	CONTAINING_OBJECT_LIB	CONTAINING_OBJECT_NAME	OBJECTINFOR	MODIFY_TIME	SC
*STMF	*DATA	-	-	<Stream_F...	2015-10-24 08:44:18.000000	0.11
*STMF	*DATA	-	-	<Stream_F...	2015-05-09 21:01:13.000000	0.0010

Ergebnis der Suche in der Textsuchgruppe „Dokumente“

Die Ergebnisspalten sind im Abschnitt „Textsuchgruppe für Source Physical Files erstellen“ beschrieben. Der Inhalt der Spalte OBJECTINFOR enthält jetzt einen XML-String, der die Information der gefundenen Dokumente enthält.

```
<Stream_File xmlns="http://www.ibm.com/xmlns/prod/db2textsearch/obj1">
      <file_path>/home/usera/a.txt</file_path>
</Spool_File>
```

7.3.9 SQL-Objekte und i5/OS-Berechtigungen

Fragt man nach den Unterschieden zwischen einer i5/OS-Bibliothek und einem SQL-Schema, werden meistens sofort die Journale und Katalogsichten erwähnt. Tatsächlich sind dies zwei Unterschiede, die auch sofort auffallen, aber bedeutender sind, meiner Meinung nach, auch die vom Betriebssystem gewählten Zugriffsberechtigungen für die SQL-Objekte. Tatsächlich verhält sich das Betriebssystem bei der Vergabe der Berechtigungen in Abhängigkeit der verwendeten Erstellungsbefehle unterschiedlich.

7.3.9.1 i5/OS-Berechtigungen für SQL-Schemata

Um zu verstehen, was gemeint ist und welche Auswirkungen die unterschiedlichen Berechtigungen haben können, sollten wir uns ein einfaches Beispiel anschauen:

Nehmen wir einmal an, dass mehrere Programmierer für die Entwicklung zuständig sind und diese Programmierer in der Gruppe „Group_PGM“ zusammengefasst sind. Das entsprechende Gruppenprofil wurde vor langer Zeit erstellt und auch die Benutzerprofile der Programmierer sind seit längerem im Einsatz:

```
_                    Benutzerprofil anzeigen - Basis

 Benutzerprofil . . . . . . . . . . . . . . . :   PRO01

 Gruppenprofil  . . . . . . . . . . . . . . . :   GROUP_PGM
 Eigner . . . . . . . . . . . . . . . . . . . :   *GRPPRF
```

Die bestehenden Benutzerprofile

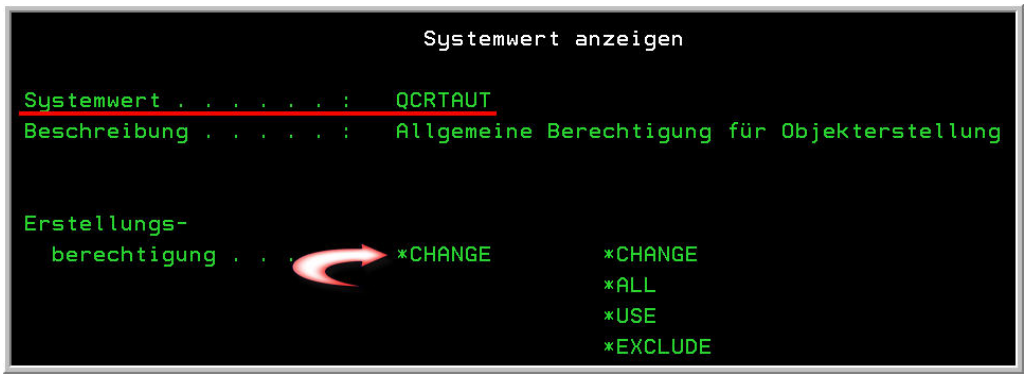
7.3.9
Seite 2

Sie können anhand der Grafik erkennen, dass das Benutzerprofil „Pro01“ der Gruppe „Group_PGM“ angehört und die Eignerschaft aller Objekte, die der Benutzer „Pro01“ erstellt, auf die Gruppe übertragen wird. Erstellt nun der Benutzer „Pro01“ die Bibliothek „DB_Lib“ mit dem CL_Befehl CRTLIB (Create Library), werden die Berechtigungen für die Bibliothek seitens des Betriebssystems folgendermaßen erteilt:

Objektberechtigungen für das neu erstellte Objekt „DB_LIB“

Wie erwartet erhält die Gruppe das Eigentumsrecht. Alle Programmierer, die Mitglied der Gruppe „Group_PGM“ sind, können mit der Bibliothek in vollem Umfang arbeiten. Sie können die Bibliotheksattribute ändern, den Inhalt der Bibliothek manipulieren aber auch über die Existenz des Objektes entscheiden. Soweit bieten die Berechtigungen keine Überraschungen. Der Öffentlichkeit (*PUBLIC) erteilt das Betriebssystem die Berechtigung *CHANGE. Lassen Sie uns in diesem Zusammenhang kurz die Frage klären: „Was versteht man in diesem Kontext unter dem Begriff der Öffentlichkeit?“ Zur Gruppe *PUBLIC gehört jeder Benutzer, der nicht direkt über den Benutzerprofilnamen oder indirekt durch ein Gruppenprofil für die Bibliothek berechtigt wurde. Als nächstes stellt sich die Frage: „Warum wird die Berechtigungsstufe *CHANGE vergeben und was beinhaltet das Recht *CHANGE?“

Die zu verwendende Berechtigungsstufe ist durch den Systemwert „QCRTAUT“ festgelegt.



Der Systemwert QCRTAUT

Die Standardeinstellung ist die Berechtigung *CHANGE. Wenn Sie ein *CHANGE-Recht für ein i5/OS-Objekt haben, können Sie zwar nicht über die Existenz des Objektes entscheiden; Sie dürfen in unserem Beispiel die Bibliothek „DB_Lib“ also nicht löschen oder ihr neue Attribute zuweisen, aber es ist überhaupt kein Problem auf die Inhalte der Bibliothek zuzugreifen; sie können auch eigene Objekte in der Bibliothek erstellen. Es ist also problemlos möglich auf die physischen Dateien in dieser Bibliothek zuzugreifen, um beispielsweise Queries zu erstellen, diese in der Bibliothek zu speichern und anschließend beliebig oft auszuführen.

```
                        Abfrage ausführen (RUNQRY)

Auswahl eingeben und Eingabetaste drücken.

Abfrage  . . . . . . . . . . . . > LISTE01___    Name, *NONE
  Bibliothek . . . . . . . . . . >   DB_LIB____  Name, *LIBL, *CURLIB
```

Query-Abfragen können erstellt und ausgeführt werden

Dieses Szenario sollte Ihnen weitestgehend bekannt sein!

Jetzt wollen wir die i5/OS-Standardbibliothek durch ein SQL-Schema ersetzen. Der Programmierer „Pro01“ erstellt zunächst das SQL-Schema „DB_SQL“. Welches Interface Sie hierfür verwenden ist unerheblich. Ich habe mich daher einfach für die 5250-Umgebung entschieden:

```
                                SQL-Anweisungen eingeben

SQL-Anweisung eingeben und Eingabetaste drücken.
   > create schema db_sql
     Schema DB_SQL wurde erstellt.
```

Das SQL-Schema DB_SQL

Nachdem das SQL-Schema erstellt wurde, versucht ein Kollege, der ebenfalls Mitglied der Gruppe „Group_PGM“ ist, die notwendigen physischen Dateien im neuen SQL-Schema zu erstellen. Doch es gibt Probleme:

```
Parameter oder Befehl
===> CRTPF FILE(DB_SQL/PAAKT00) SRCFILE(EPS005CR/QDBSRC)
F3=Verlassen     F4=Bedienerführung   F5=Aktualisieren     F6=Erstellen
F9=Auffinden     F10=Befehlseingabe   F23=Weitere Angaben  F24=Weitere Tasten
Keine Berechtigung, Objekt DB_SQL in Bibliothek QSYS von Art *LIB zu verwend
```

Keine Berechtigung für Bibliothek DB_SQL

7.3.9
Seite 4

Trotz Gruppenzugehörigkeit kann das Teammitglied keine Objekte in der Bibliothek speichern. Der zweite Programmierer hat anscheinend keine Berechtigung für das SQL-Schema. Was ist passiert? Um das Problem zu analysieren, sehen wir uns zunächst einmal die Berechtigungen für das SQL-Schema an:

```
                        Objektberechtigung editieren

Objekt . . . . . :   DB_SQL          Eigner . . . . . :   PRO01
  Bibliothek . . . :     QSYS        Primärgruppe . . . :   *NONE
Objektart  . . . . :   *LIB          ASP-Einheit  . . . :   *SYSBAS

Aktuelle Berechtigungen ändern und Eingabetaste drücken.

  Objekt durch Berechtigungsliste geschützt . . . . . .   *NONE

                         Objekt-
Benutzer     Gruppe      berechtg.
*PUBLIC                  *EXCLUDE
PRO01                    *ALL
```

Die Berechtigungen für das SQL-Schema

Schnell wird klar, warum der zweite Programmierer nicht auf die Bibliothek zugreifen kann; das Betriebssystem hat nicht die Gruppe „Group_PGM“ zum Eigentümer gemacht, sondern die Objekteignerschaft dem Programmierer „Pro01“ erteilt. Die Gruppenprofileinstellungen bleiben unberücksichtigt. Warum bleiben die Gruppenprofileinstellungen unberücksichtigt? Liegt es am verwendeten Interface? Oder erhalten die SQL-Schemata grundsätzlich andere Zugriffsrechte?

Die Antwort lautet: Weder das Interface noch der Befehl „Create Schema“ ist für die abweichenden Objektberechtigungen verantwortlich, sondern das verwendete SQL-Format bestimmt in unserem Beispiel die Objektberechtigungen.

```
                        Sitzungsattribute ändern

Auswahl eingeben und Eingabetaste drücken.

  Anweisungsverarbeitung . . . .   *RUN
  SELECT-Ausgabe . . . . . . . .   1                              =Drucker

  Datenaktualisierung  . . . . .   *ALWAYS     *ALWAYS, *FORWARD
  Datenkopie zulässig  . . . . .   *YES        *YES, *OPTIMIZE, *NO     *JIS
  Namenskonvention . . . . . . .   *SQL        *SYS, *SQL
  Datumstrennzeichen . . . . . .   '.'         *JOB, '/', '.', ',', '-'
                                               ' ', *BLANK
```

Die verwendete Namenskonvention bestimmt indirekt über die erteilten Objektberechtigungen.

*Namenskonvention *SQL – STRSQL*

Wenn Sie im SQL-Interface das Format *SQL wählen, werden anschließend die erläuterten Gruppenprofileinstellungen ignoriert. Wählen Sie hingegen das SQL-Format *SYS, ist die Welt wieder in Ordnung und zwar unabhängig vom eingesetzten Interface, wie nachfolgende Grafik zeigt:

*Ergebnis des Befehls „Create schema" – Namenskonvention *SYS*

Wir können also feststellen, dass die Schemata dieselben Berechtigungen wie unsere Bibliotheken erhalten, wenn wir im entsprechenden SQL-Interface die Namenskonvention *SYS wählen.

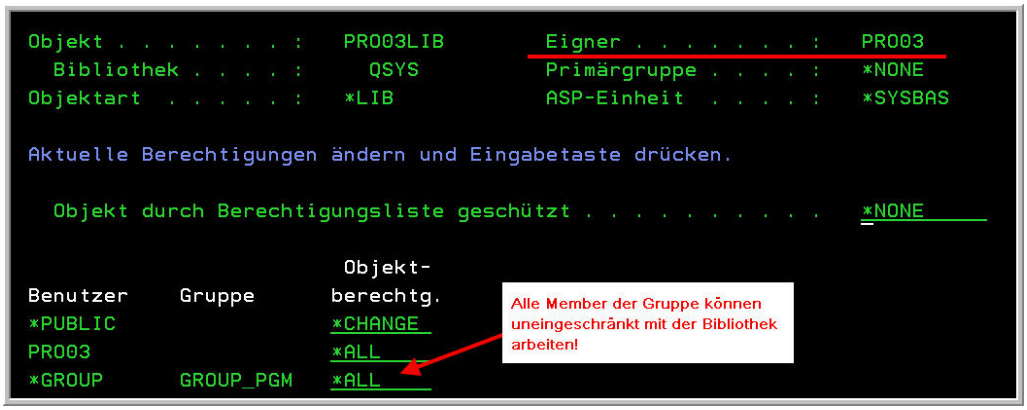
Was passiert aber, wenn Sie nicht sicherstellen können, dass ausschließlich mit der Namenskonvention *SYS gearbeitet wird? Gibt es keine Möglichkeiten mit Gruppenprofilen zu arbeiten, wenn die Namenskonvention *SQL genutzt wird? Tatsächlich gibt es seit längerer Zeit die Möglichkeit Gruppenprofile auf zwei verschiedene Arten zu erstellen. Schauen Sie sich die folgende Abbildung an:

Gruppenprofile Variante 2

In der Gruppenprofilvariante 2 behält das Benutzerprofil die Eignerschaft am Objekt. Den Mitgliedern der Gruppe werden hingegen explizit Rechte am Objekt gewährt.

7.3.9
Seite 6

Erstellt in meinem Beispiel das Benutzerprofil „Pro03“ eine Bibliothek, indem der CL-Befehl „CRTLIB“ verwendet wird, werden folgende Objektberechtigungen vom System erteilt:



Wirkungsweise der Gruppenprofile Variante 2

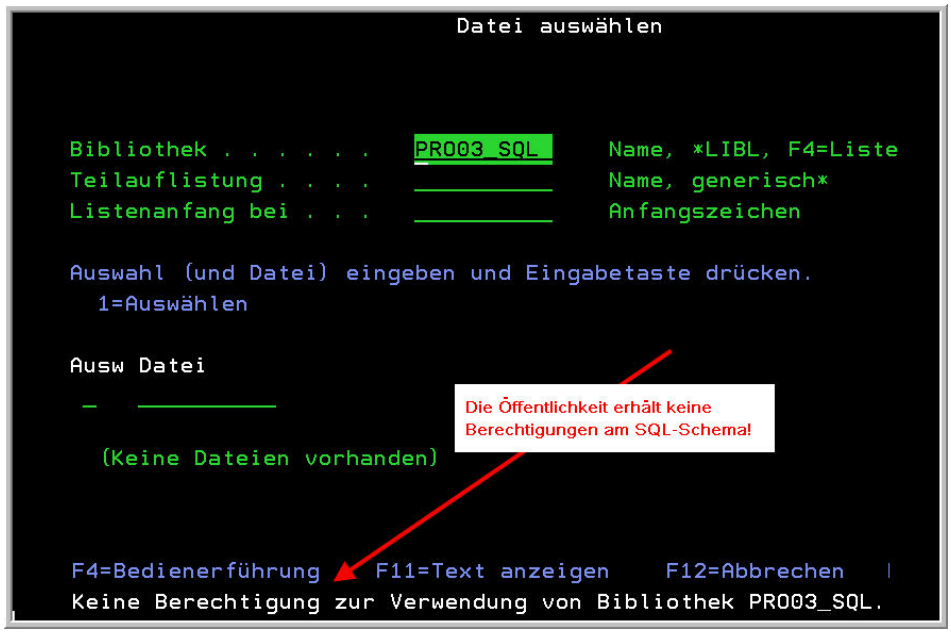
Das Benutzerprofil „Pro03“ wird Eigentümer der Bibliothek und erhält zunächst einmal alle Rechte an der erstellten Bibliothek. Die Mitglieder der Gruppe „Group_PGM“ können aber trotzdem uneingeschränkt mit der Bibliothek arbeiten, da sie über die Gruppenberechtigung ebenfalls alle Rechte am Objekt erhalten.

Was passiert nun aber, wenn der Benutzer „Pro03“ ein SQL-Schema erstellt und dabei die Namenskonvention *SQL verwendet?

Berechtigungen für SQL-Schemata – Gruppenprofile Variante 2

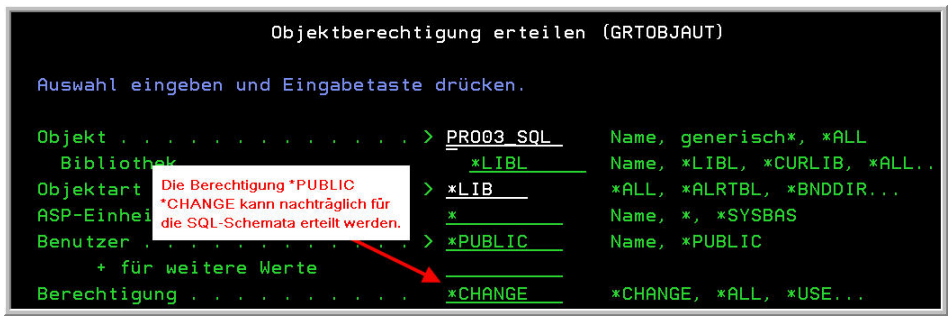
Anders als zuvor werden die Gruppenprofileinstellungen dieses Mal berücksichtigt. Auch jetzt bleibt der Benutzer „Pro03“ zwar Eigentümer, aber das Betriebssystem erteilt auch für die SQL-Objekte automatisch allen Gruppenmitgliedern die Berechtigung *ALL. Sie müssten also ggf. Ihre Benutzerprofileinstellungen überarbeiten, wenn Sie mit der Namenskonvention *SQL arbeiten müssen, aber dennoch Gruppenberechtigungen benötigen.

Wenn Sie die beiden vorangegangenen Grafiken noch einmal genauer betrachten, werden Sie allerdings nach wie vor einen Unterschied feststellen. Für die Bibliothek „Pro03_Lib“ erhält die Öffentlichkeit (*PUBLIC) im ersten Beispiel die Berechtigung *CHANGE, im zweiten Beispiel nur die Berechtigung *EXCLUDE. Sobald Sie ein SQL-Schema erstellen und im Interface die Namenskonvention *SQL gewählt wurde, kann die Öffentlichkeit nicht mehr auf das SQL-Schema zugreifen.



Die Anwender erhalten keine Berechtigung für die Bibliothek „Pro03_SQL“

Falls Sie bisher Ihre Anwender nicht über Gruppenprofile berechtigen, sollten Sie dies jetzt nachholen, wenn Sie mit der Namenskonvention *SQL arbeiten wollen. Alternativ können Sie natürlich auch die Berechtigungen für die Öffentlichkeit nachträglich korrigieren:



Der Befehl „Grant object authority (GRTOBJAUT)“

Bevor Sie sich jedoch für diese Möglichkeit entscheiden, sollten Sie nochmals genau überprüfen, ob Sie der Öffentlichkeit so weit reichende Berechtigungen erteilen wollen. Auch wenn ein wenig mehr Arbeit investiert werden muss, kann es vorteilhaft sein, die Zugriffsberechtigungen auf Ihre Firmendaten detailliert über Gruppen zu steuern.

7.3.9.2 i5/OS-Berechtigungen für SQL-Tabellen

Wir haben gesehen, dass die Objektberechtigungen für die SQL-Schemata von der verwendeten Namenskonvention abhängig sind. Gilt diese Aussage nur für die SQL-Schemata oder sind auch die physischen DDS-Dateien und SQL-Tabellen davon betroffen?

Schauen wir uns auch hierzu ein Beispiel an. Erstellt wird zunächst die physische Datei „PAAKT00“ mit dem Befehl „Create physical file(CRTPF)“ in der Standardbibliothek „DB_LIB“.

CRTPF DB_LIB/PAAKT00

Die Bibliothek „DB_Lib“ ist eine einfache i5/OS-Bibliothek, wie wir wissen. Die physische Datei wird mit der Berechtigung *PUBLIC *CHANGE erstellt.

*Die Öffentlichkeit erhält ein *CHANGE Recht an der physischen Datei*

Die Öffentlichkeitsrechte werden aber nicht von den Bibliotheksrechten abgeleitet, sondern durch den Befehl „Create physical file(CRTPF)“ bestimmt, denn selbst wenn die Öffentlichkeit nicht mit der Bibliothek arbeiten darf (*PUBLIC *EXCLUDE), ändert sich nichts an der eben beschriebenen Berechtigungsvergabe der physischen Dateien. Erweitern Sie einmal die Parameter des Befehls „Create physical file(CRTPF)“. Jetzt finden Sie den Parameter „Authority(AUT)“. Dieser Parameter – den Sie übrigens in allen CL-Erstellungsbefehlen finden – bestimmt stets die Öffentlichkeitsrechte.

```
                    Physische Datei erstellen (CRTPF)

Auswahl eingeben und Eingabetaste drücken.

Zeichensatz-ID . . . . . .   *JOB_____    *JOB, *HEX...
Aktualisieren zulässig . . . .  *YES       *YES, *NO
Löschen zulässig . . . . . .  *YES         *YES, *NO
Satzformat-Aktualitätsprüfung  .  *YES     *YES, *NO
Knotengruppe . . . . . .  *NONE_____       Name, *NONE
  Bibliothek . . . . . .    __________      Name, *LIBL, *CURLIB
Partitionierungsschlüssel  . . .  _____    Name
     + für weitere Werte          _____
Berechtigung . . . . . .  *LIBCRTAUT       Name, *LIBCRTAUT, *ALL
```

Der Parameter „Aut“ des Befehls „Crtlib“

Der Parameter enthält den Wert *LIBCRTAUT. Falls Sie in der Bedienerhilfe nachschauen, finden Sie folgende Erläuterung: „Das System verwendet den Wert, der bei Parameter Berechtigung für neue Objekte (CRTAUT) im Befehl CRTLIB (Bibliothek erstellen) für die Bibliothek, die das zu erstellende Objekt enthält, angegeben wurde“. Die Erklärung ist – wie so oft – schwer zu verstehen. Gemeint ist folgendes: Entscheidend für die Berechtigungsvergabe ist nicht der Erstellungsbefehl selbst, sondern ein Attribut unserer Bibliothek. In den erweiterten Parametern der Bibliothek „DB_Lib“ finden wir das Attribut „Berechtigungen für neue Objekte“. Die hier hinterlegten Berechtigungen bestimmen letztendlich, welche Berechtigungen Benutzer erhalten, die nicht ausdrücklich über ihr Benutzerprofil oder entsprechende Gruppenprofile berechtigt werden. Alle i5/OS-Bibliotheken haben den Standardwert *CHANGE.

So erklärt es sich auch, warum der Öffentlichkeit für die gleiche Datei andere Rechte gewährt werden, wenn sie in einem SQL-Schema erzeugt wird:

```
                      Objektberechtigung editieren

Objekt . . . . . :   DB_SQL          Eigner . . . . . :   PRO01
  Bibliothek . . . . :     QSYS      Primärgruppe . . . . :   *NONE
Objektart  . . . . . :   *LIB        ASP-Einheit  . . . . :   *SYSB

Aktuelle Berechtigungen ändern und Eingabetaste drücken.

  Objekt durch Berechtigungsliste geschützt . . . . . .   *NONE

                       Objekt-
Benutzer     Gruppe     berechtg.
*PUBLIC                 *EXCLUDE
PRO01                   *ALL
```

Berechtigungen des SQL-Schemas „DB_SQL“

7.3.9
Seite 10

Wird die phyische DDS-Datei in einem SQL-Schema erstellt, erhält sie – anders als vorhin – die Rechte *PUBLIC *EXCLUDE. Die Erklärung ist einfach: Alle SQL-Schemata haben den Standardwert *EXCLUDE für den Parameter „Berechtigungen für neue Objekte“.

Treffen diese Unterschiede auch zu, wenn wir SQL-Tabellen erstellen? Unterscheidet das Betriebssystem auch hier zwischen Bibliotheken und Schemata? Und welche Rolle spielt in diesem Kontext die SQL-Namenskonvention?

Erstellen wir zunächst eine SQL-Tabelle in unserer Standardbibliothek „DB_LIB“. Wenn wir die Namenskonvention *SYS wählen, gibt es keinerlei Unterschiede zu den physischen DDS-Dateien – auch die SQL-Tabelle erhält die öffentliche Berechtigung *CHANGE. Wird die Namenskonvention *SYS gewählt, wird auch die Berechtigung für die SQL Tabellen von der zugehörigen Bibliothek bzw. dem verwendeten SQL-Schema abgeleitet. Schauen Sie, was gemeint ist:

```
                        Bibliothek ändern (CHGLIB)

Auswahl eingeben und Eingabetaste drücken.

Bibliothek . . . . . > PR003_SQL_    Name, *CURLIB
Bibliotheksart . . . . . .   *PROD         *SAME, *PROD, *TEST
Text 'Beschreibung'  . . . . . .   'COLLECTION - von SQL erstellt.
      '
                Zusätzliche Parameter

Berechtigung für neue Objekte  .   *use_____    Name, *SAME, *SYSVAL...
Objektprotokollierung  . . . . .   *SAME_       *SAME, *SYSVAL, *NONE...
```

Der Parameter CRTAUT der Bibliothek PR03_SQL

Standardmäßig ist der Parameter „Berechtigung für neue Objekte“ auf den Wert *CHANGE gesetzt, wenn es sich um eine Bibliothek handelt. SQL-Schemata erhalten per Default den Wert *EXCLUDE. In meinem Beispiel habe ich den Wert einfach geändert – wie Sie in der Abbildung erkennen können – und verwende jetzt die Berechtigung *USE. Erstelle ich anschließend eine Tabelle innerhalb des Schemas und nutze dafür die Namenskonvention *SYS für das SQL-Statement oder den Befehl CRTPF, wird die Tabelle mit der Berechtigung *PUBLIC *USE erzeugt:

*Wirkungsweise des Parameters CRTAUT – Namenskonvention *SYS*

Anders sieht es aus, wenn ich die Namenskonvention *SQL für die Erstellung meiner SQL-Tabelle verwende. In diesem Fall erhält die Tabelle immer die öffentliche Berechtigung *EXCLUDE unabhängig von den entsprechenden Bibliotheksattributen.

*Wirkungsweise des Parameters CRTAUT – Namenskonvention *SQL*

Sie sehen, die Berechtigungsvergabe ist ein klein wenig verwirrend, wenn Sie SQL-Objekte verwenden und nicht auf die verwendete Namenskonvention des SQL-Interfaces achten. Komplizierter wird es natürlich auch, wenn Sie beginnen, die Objekte innerhalb einer Bibliothek zu mischen. Sie sollten daher aus Gründen der Übersicht versuchen:

- Die Namenskonvention in den verschiedenen Interfaces einheitlich und durchgängig zu verwenden
- die SQL-Objekte nicht mit den „alten“ i5/OS-Objekten innerhalb einer Standardbibliothek oder eines SQL-Schemas zu vermengen.

7.3.9
Seite 12

Falls Sie dennoch keine Wahl haben, hilft vielleicht die folgende Zusammenfassung:

	Gruppenberechtigung	Öffentlichkeitsrechte
CREATE Schema (Namenskonvention *SYS)	Beide Gruppenprofilvarianten werden berücksichtigt	*CHANGE
CREATE Schema (Namenskonvention *SQL)	Nur wenn die Gruppenrechte explizit im Benutzerprofil hinterlegt werden und der Benutzer Objekteigentümer bleibt, werden die Gruppenprofile berücksichtigt; das ausführende Benutzerprofil wird Eigentümer des Objektes	*EXCLUDE
CRTPF (physische DDS-Datei wird in einer Bibliothek erzeugt)	Gruppenberechtigungen (Variante 1 und Variante 2) werden entsprechend berücksichtigt	Default: *CHANGE Abgeleitet von dem Parameter CRTAUT der zugehörigen Bibliothek
CRTPF (physische DDS-Datei wird in einem SQL-Schema erzeugt)	Gruppenberechtigungen (Variante 1 und Variante 2) werden entsprechend berücksichtigt	Default *EXCLUDE Abgeleitet von dem Parameter CRTAUT der zugehörigen Bibliothek/Schemata
CREATE TABLE (Namenskonvention *SYS)	Gruppenberechtigungen (Variante 1 und Variante 2) werden entsprechend berücksichtigt	Abgeleitet von dem Parameter CRTAUT der zugehörigen Bibliothek/Schemata
CREATE TABLE (Namenskonvention *SQL)	Gruppenberechtigungen (Variante 1 und Variante 2) werden entsprechend berücksichtigt	*EXCLUDE

7.3.9.5 Zeilen und Spaltenberechtigungen (RCAC)

Wir alle kennen die Rechtevergabe auf einzelne Objekte des IBM i Servers. Objektberechtigungen gibt es schon genauso lang wie es die AS/400 gibt. Allerdings können wir mit Mitteln des Betriebssystems nur die Rechte auf das Objekt insgesamt erteilen – oder sie entziehen. Man kann also sagen, mit Objektberechtigungen kann zwar der Zugriff auf das Objekt mit seinen Daten eingeschränkt werden, nicht aber auf die Datenbereiche.

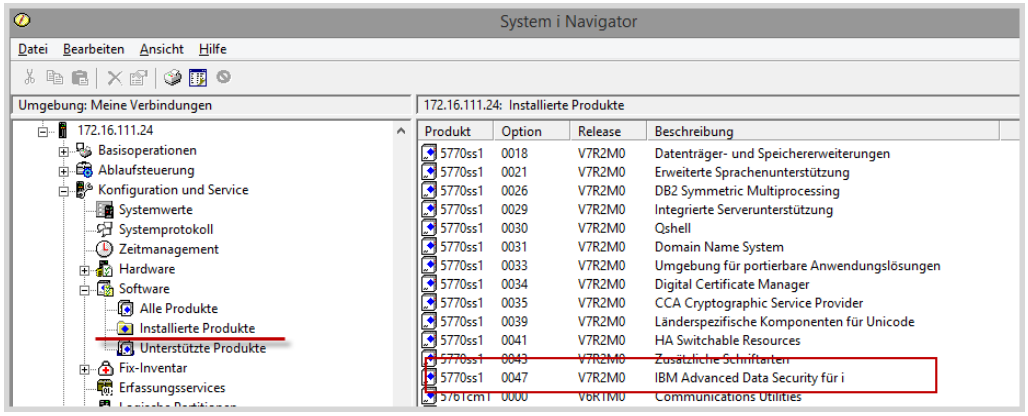
Nun gibt es allerdings Szenarien, in denen der Zugriff genauer gesteuert werden muss, als dass durch einfaches Gewähren, Widerrufen oder Ablehnen von Berechtigungen für Daten möglich ist. Nehmen wir das beliebte Beispiel der Personalabteilungen: Vielleicht dürfen nicht alle Mitarbeiter der Personalabteilung alle Informationen eines jeweiligen Mitarbeiters lesen. Bisher wurde dieses Problem in erster Linie über entsprechende Prüfungen im Programmcode eingeschränkt. Sobald aber der Zugriff auf die Daten in einem anderen Interface (SQL, IBM i Navigator, ODBC, JDBC, QUERY/400) erfolgt, sind die Daten ungeschützt. Natürlich war es möglich, die sensiblen Daten in einer eigenständigen Datei zu speichern, aber dadurch wird der Datenzugriff unnötig kompliziert, und auch das Datenbankdesign wird unübersichtlicher. Das Beispiel mit der Personalabteilung ist stets das Paradebeispiel, das genannt wird, um die Problematik zu verdeutlichen, aber es gibt auch andere Beispiele: Vertriebsmitarbeiter sollen beispielsweise lediglich auf die ihnen zugeordneten Kunden zugreifen und auch nur ihre eigenen Daten auswerten dürfen. Mandantenfähigkeit musste bislang häufig über separate Bibliotheken realisiert werden, um sicherzustellen, dass Mitarbeiter nicht firmenübergreifend Informationen abrufen können. Wenn Sie noch ein klein wenig überlegen, fallen Ihnen sicherlich noch viele weitere Beispiele ein.

Das oben geschilderte Szenario kann seit Release i7.2 gelöst werden. Denn in Release i7.2 ist es jetzt möglich, Zeilen- und Spaltenberechtigungen für Tabellen zu vergeben. Selbst Benutzer mit der Sonderberechtigung *ALLOBJ können diese Berechtigungen nicht unterlaufen und sehen nur noch die für Sie explizit berechtigten Zeilen und Spalten. Diese neuen Berechtigungen werden unter dem Oberbegriff „Row and Column Access“ (RCAC) zusammengefasst.

7.3.9.5.1 Zeilen- / Spaltenberechtigungen (RCAC) – Voraussetzungen

Bevor Sie allerdings mit den neuen Berechtigungsbefehlen arbeiten können, müssen ein paar Voraussetzungen erfüllt sein:

Achten Sie darauf, dass die Option 47 IBM Advanced Data Security for i auf dem Server installiert worden ist.

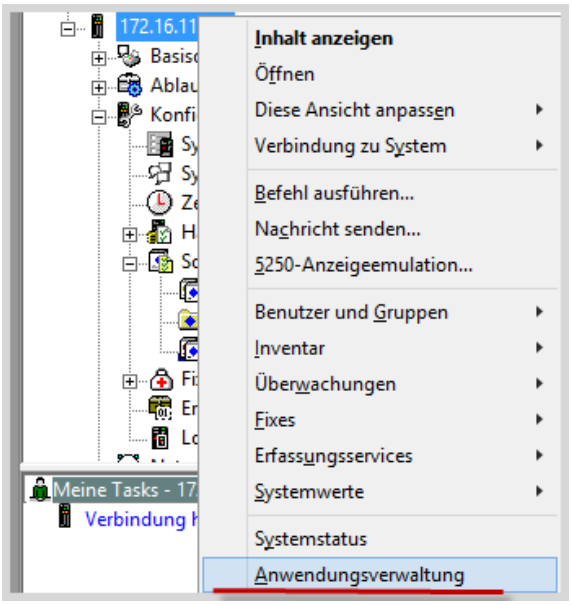


Installation überprüfen

Alle Benutzer, die Zeilen- und Objektberechtigungen vergeben sollen, müssen dafür explizit in der Anwendungsverwaltung registriert werden. Die Sonderberechtigungen *SECADM und *ALLOBJ reichen dieses Mal als Legitimierung nicht aus.

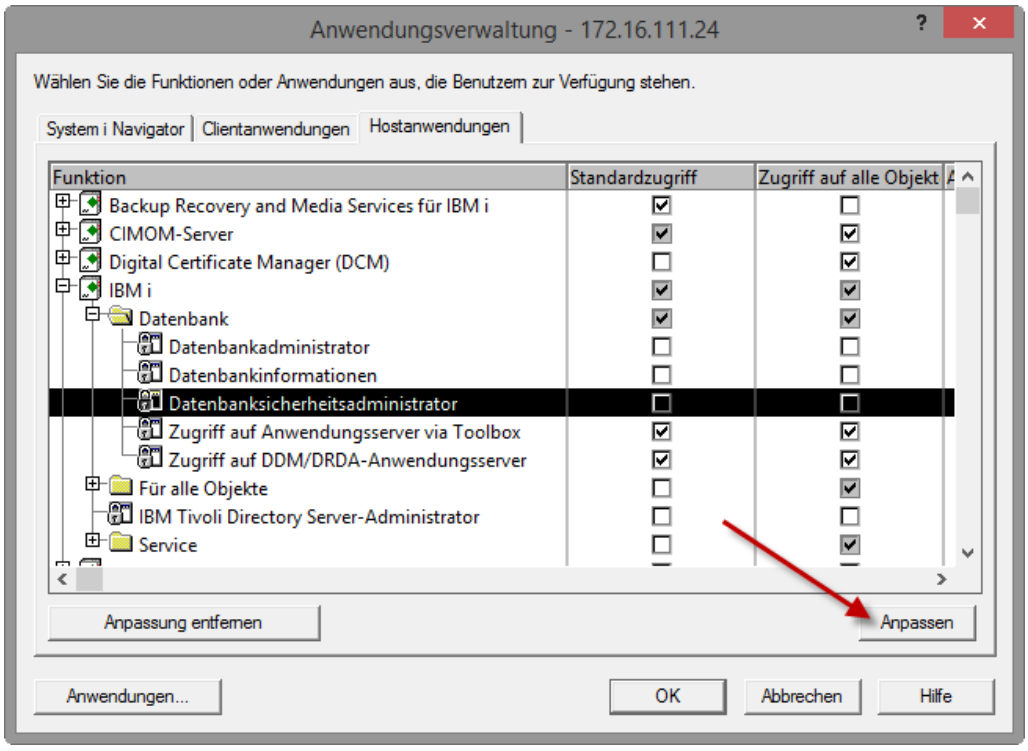
Um die Registrierung vorzunehmen, haben Sie zwei Möglichkeiten:

Im IBM i System Navigator:
Dafür öffnen Sie das Kontextmenü der Serververbindung und wählen den Menüpunkt „Anwendungsverwaltung“ aus, wie in nachfolgender Abbildung gezeigt wird.



Anwendungsverwaltung öffnen

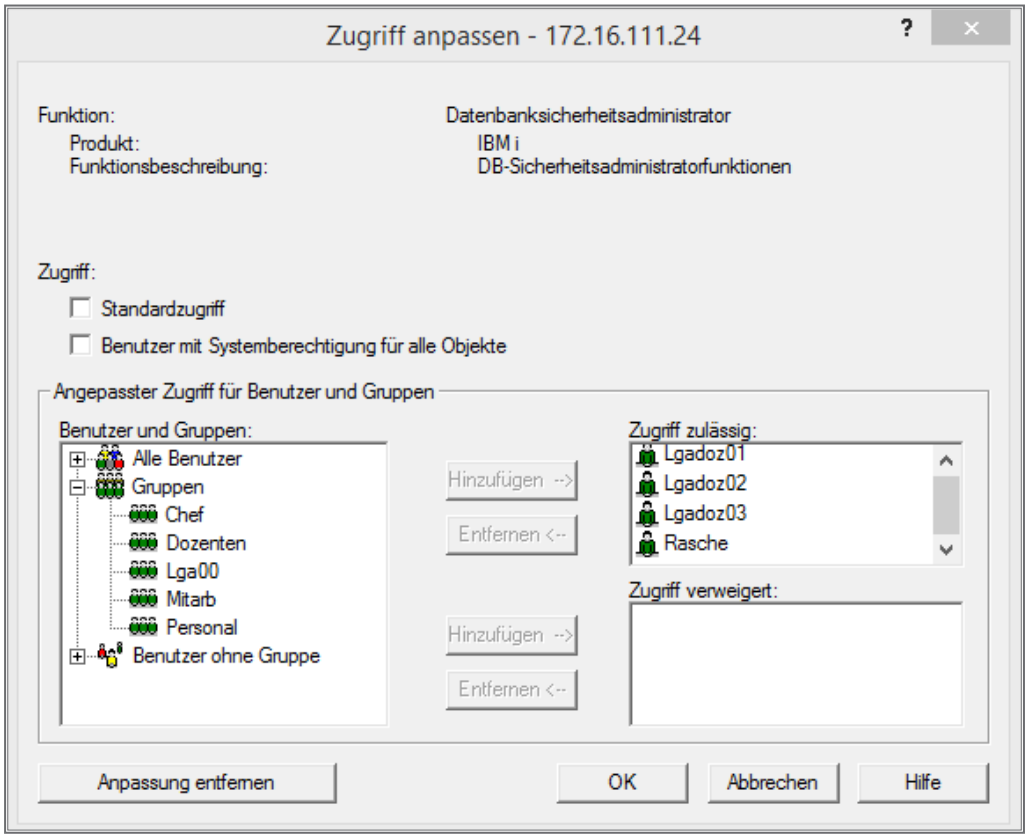
Nachdem Sie sich auf dem zentralen System angemeldet haben, wählen Sie im Dialog den Eintrag „Hostanwendungen“. Markieren Sie jetzt den Eintrag „Datenbanksicherheitsadministrator“:



Anwendungsverwaltung – Hostanwendungen

7.3.9
Seite 16

Klicken Sie jetzt auf den „Button“ „Anpassen“. Nun folgt ein weiterer Dialog, der Ihnen die Auswahl von Gruppenprofilen oder einzelnen Benutzerprofilen ermöglicht.



Anwendungsverwaltung – Zugriff anpassen

Alle Benutzergruppen oder Benutzerprofile, die Sie hier auswählen, dürfen anschließend mit den neuen Berechtigungsbefehlen arbeiten.

Alternativ können Sie diese Berechtigungen natürlich auch mit entsprechenden CL-Befehlen erteilen. Der Befehl WRKFCNUSG (Work with function usage) öffnet einen Dialog, der es ermöglicht die entsprechenden Profile ebenfalls zu berechtigen:

```
                          Work with Function Usage

Type options, press Enter.
  2=Change usage   5=Display usage

Opt  Function ID                    Function Name
     QIBM_DIRSRV_ADMIN              IBM Tivoli Directory Server Administrator
     QIBM_ACCESS_ALLOBJ_JOBLOG      Access job log of *ALLOBJ job
     QIBM_ALLOBJ_TRACE_ANY_USER     Trace any user
     QIBM_WATCH_ANY_JOB             Watch any job
     QIBM_DB_DDMDRDA                DDM & DRDA Application Server Access
     QIBM_DB_SECADM                 Database Security Administrator
     QIBM_DB_SQLADM                 Database Administrator
```

Mit Funktionsnutzung arbeiten – WRKFCNUSG

Mit der Option 5 können Sie sich die aktuellen Einstellungen anzeigen lassen (DSPFCNUSG):

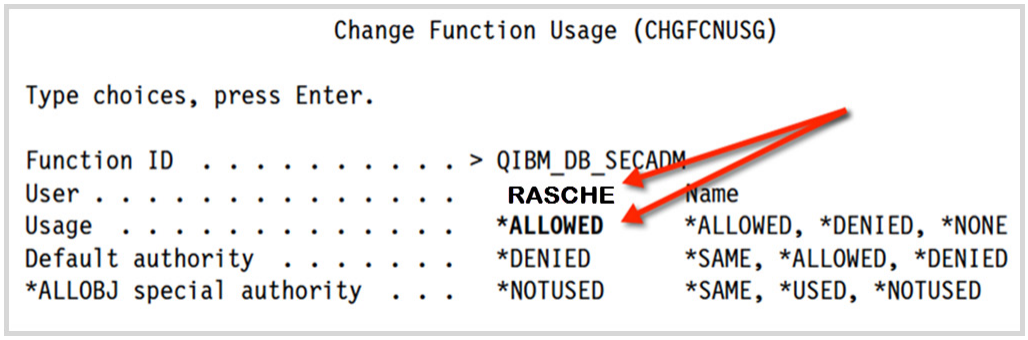
```
                          Display Function Usage

 Function ID  . . . . . :   QIBM_DB_SECADM
 Function name  . . . . . :   Database Security Administrator

 Description  . . . . . :   Database Security Administrator Functions


 Product  . . . . . :   QIBM_BASE_OPERATING_SYSTEM
 Group  . . . . . :   QIBM_DB

 Default authority  . . . . . :   *DENIED
 *ALLOBJ special authority  . . . . . :   *NOTUSED

 User        Type          Usage       User        Type          Usage
 DEVELOPER1  User          *DENIED
 DEVELOPER2  User          *DENIED
```

Funktionsnutzung anzeigen – DSPFCNUSG

In der oberen Abbildung können Sie auch erkennen, warum die *ALLOBJ-Berechtigung nicht ausreicht, um die neuen Berechtigungsbefehle zu verwenden.

7.3.9
Seite 18

Wählen Sie Option 2 im Ausgangsbildschirm „Mit Funktionsnutzung arbeiten (WRKFCNUSG)“ oder den Befehl CHGFCNUSG., dann können Sie anschließend entsprechende Änderungen vornehmen:



Funktionsnutzung ändern – CHGFCNUSG

7.3.9.5.2 Zeilenberechtigung erteilen

Sind die Voraussetzungen erfüllt, können Zeilenberechtigungen mit folgendem SQL-Befehl erteilt werden:

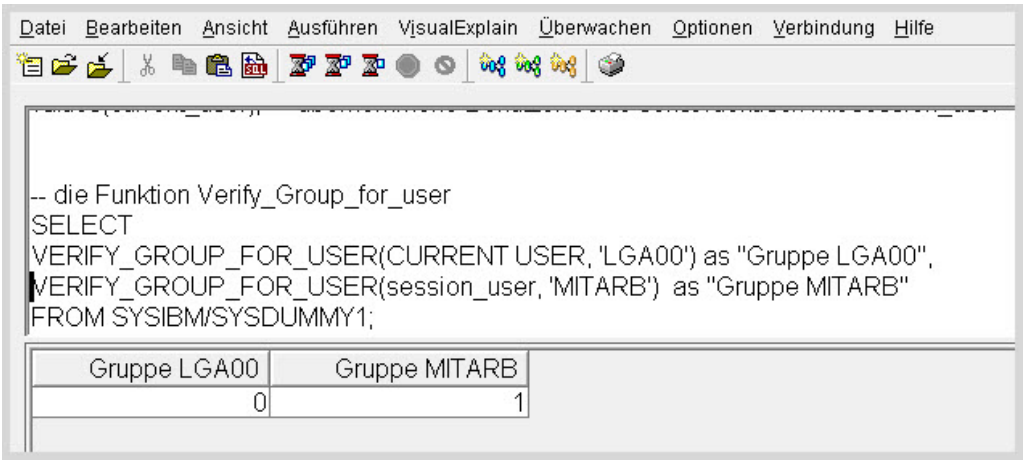
```
CREATE OR REPLACE PERMISSION.
```

Erfreulich ist, dass IBM für diesen Erstellungsbefehl die REPLACE-Option integriert hat. Dies erspart uns bei Änderungen den Befehl DROP PERMISSION.

Für die Erteilung der Berechtigungen benötigen wir spezielle Register. Sie erinnern sich an deren Bedeutung?

Register	Bedeutung
USER oder SESSION_USER	beinhaltet den Namen des aktuellen Benutzerprofils, das den Thread anfordert
SYSTEM_USER	beinhaltet den Namen des Benutzerprofils, das die Verbindung aufgebaut hat (häufig QUSER)
CURRENT USER oder CURRENT_USER	beinhaltet gewöhnlich – wie das Register USER – den aktuellen Benutzer, der die Threadanforderung ausgelöst hat. Nur wenn es sich um Programme oder Umgebungen mit übernommenen Benutzerrechten (adopted authority) handelt, enthält das Register den Benutzerprofilnamen, von dem die Rechte übernommen wurden.

Außerdem benötigen wir gleich die skalare Funktion VERIFY_GROUP_FOR_USER. Diese Funktion prüft, ob das aktuelle Benutzerprofil Mitglied einer bestimmten Benutzergruppe ist.



Wirkungsweise der Funktion VERIFY_GROUP_FOR_USER

In der oberen Abbildung können Sie erkennen, dass die Funktion den Wert 1 liefert, wenn die Bedingung wahr ist. Im Übrigen ist es mit der Funktion natürlich auch möglich, einfach den Namen eines Benutzerprofils zu prüfen.

```
SELECT
VERIFY_GROUP_FOR_USER(session_user, 'RASCHE') as "Gruppe MITARB"
FROM SYSIBM/SYSDUMMY1;
```

Auch diese Abfrage gibt den Wert 1 zurück, obwohl das Profil RASCHE kein Gruppenprofil ist. Und Sie können auch mehr als einen Profilnamen als Vergleichswert verwenden:

```
SELECT
VERIFY_GROUP_FOR_USER(session_user, 'LGA00', 'RASCHE') as "Gruppe MITARB"
FROM SYSIBM/SYSDUMMY1;
```

7.3.9
Seite 20

Der aktuelle Benutzer muss entweder mit dem Profil RASCHE angemeldet sein oder zur Gruppe LGA00 gehören.

Nun haben sich auch SQL-seitig die Voraussetzungen geklärt und können entsprechende Zeilenberechtigungen mit dem Befehl CREATE PERMISSION aufbauen. Stellen Sie sich dafür das folgende Szenario vor.

Es gibt zwei Gruppenprofile:

- MITARB beinhalter alle Mitarbeiter der Firma
- PERSONAL enthält die Mitarbeiter der Personalabteilung

Und es gibt natürlich die Tabelle PERSONAL – unter anderem mit folgenden Informationen:

PERSONALNU...	VORNAME	NACHNAME	MONATSGEHALT	BANKVERBINDUNGIB...
000010	CHRISTINE	HAAS	5078.00	BRLADE21
000020	MICHAEL	MAYER	14000.00	BRLADE21
000030	SALLY	KWAN-CHANG	12800.00	BRLADE21
000050	JOHN	MEYER LUDWIG	11050.00	BRLADE21
000060	IRVING	STERN	10000.00	BRLADE21
000070	EVA	PULASKI	11200.00	BRLADE21
000090	EILEEN	HENDERSON	8900.00	BRLADE21

Die Tabelle PERSONAL

```
CREATE OR REPLACE PERMISSION
ZEILENZUGRIFF_PERSONALDATEN                                    1
ON PERSONAL                                                    2
FOR ROWS WHERE
(VERIFY_GROUP_FOR_USER(SESSION_USER, 'MITARB') = 1             3
AND UPPER(NACHNAME) = SESSION_USER)
OR
(VERIFY_GROUP_FOR_USER(SESSION_USER, 'PERSONAL') = 1)          4
ENFORCED FOR ALL ACCESS                                        5
ENABLE ; -- Deaktivierung DISABLE                              6
```

Schauen wir uns das Statement etwas genauer an:

1. Nach dem eigentlichen Befehl folgt ein beliebiger Name, das heißt, die Berechtigung wird benannt. Dies erleichtert später die Identifizierung der diversen Berechtigungen.
2. Anschließend wird die Tabelle genannt.
3. Und schließlich folgt die eigentliche Bedingung: Im Beispiel muss der jeweilige Benutzer zur Gruppe MITARB gehören, und die anzuzeigenden Datensätze müssen dem Namen des Benutzers entsprechen, das heißt, jeder Mitarbeiter darf seinen eigenen Personalstammsatz sehen, aber nicht den der Kollegen.
4. In der zweiten Bedingung bestimmen wir, dass alle Personen, die zur Gruppe „Personal“ gehören, Zugriff auf alle Tabellenzeilen haben.
5. Diese Regeln gelten für die gesamte Tabelle und alle Tabellenzeilen …
6. und sollen aktiviert werden.

Um die Regel jetzt endgültig zu implementieren, müssen wir noch die Tabelle aktualisieren:

```
ALTER TABLE PERSONAL ACTIVATE ROW ACCESS CONTROL;
ALTER TABLE PERSONAL DEACTIVATE ROW ACCESS CONTROL;
```

Anschließend greifen die Zeilenberechtigungen. Ein An- oder Abmelden der Benutzer ist dafür nicht erforderlich:

PERSONALNU...	VORNAME	NACHNAME	MONATSGE...	BANKVERBI...	KONTOVE...
000010	CHRISTINE	HAAS	5078.00	BRLADE21	...
000020	MICHAEL	MAYER	14000.00	BRLADE21	...
000030	SALLY	KWAN-CHANG	12800.00	BRLADE21	...
000050	JOHN	MEYER LUDWIG	11050.00	BRLADE21	...
000060	IRVING	STERN	10000.00	BRLADE21	...
000070	EVA	PULASKI	11200.00	BRLADE21	...
000090	EILEEN	HENDERSON	8900.00	BRLADE21	...
000100	THEODORE	SPENSER	20000.00	BRLADE21	...
000110	VINCENZO	LUCCHESSI	3500.00	BRLADE21	...

Anzeige für Mitarbeiter der Personalabteilung

PERSONALNU...	VORNAME	NACHNAME	MONATSGE...	BANKVERBINDUNGIB...	KONTOVERBINDUNG...
993400	Carmen	Rasche	3500.00	BRLADE21	DE000000000

Anzeige für einfache Mitarbeiter

Diese Einschränkung lässt sich nicht umgehen, das heißt, auch dann nicht, wenn die Tabelle nativ im RPG-Programm verarbeitet wird oder Sie über eine SQL-View auf die Daten zugreifen. Der Zeilenschutz ist immer wirksam.

7.3.9
Seite 22

Schauen wir uns ein zweites Beispiel an:

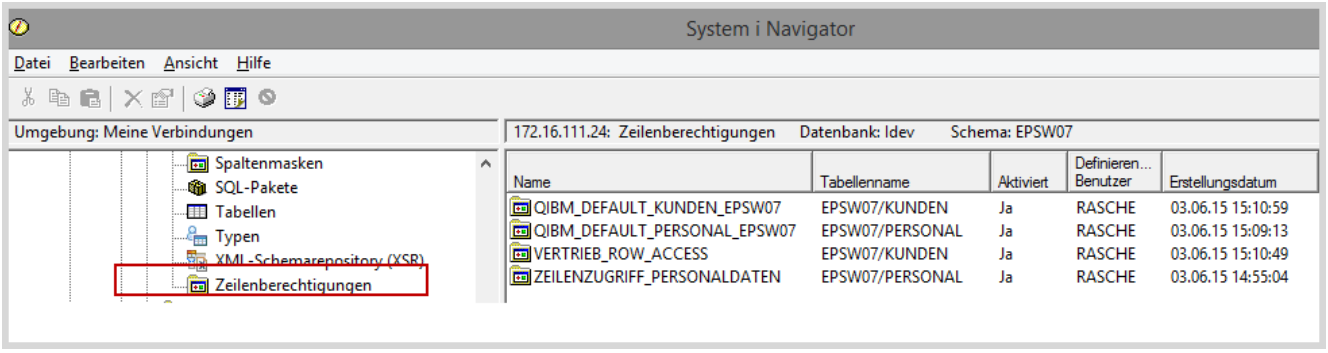
Im zweiten Beispiel wollen wir sicherstellen, dass die Vertriebsmitarbeiter nur auf die eigenen Kunden zugreifen können. Dazu erteilen wir die entsprechende Zeilenberechtigung für die Tabelle KUNDEN:

```
CREATE OR REPLACE PERMISSION
VERTRIEB_ROW_ACCESS
ON KUNDEN
FOR ROWS WHERE
VERIFY_GROUP_FOR_USER(SESSION_USER, 'MITARB') = 1
AND KUNDEN_GEBIET =
                (SELECT VERKAUFSGEBIET
                FROM VERTRIEBSMITARBEITER
                JOIN PERSONAL USING(PERSONALNUMMER)
                WHERE UPPER(NACHNAME) = SESSION_USER)
OR
VERIFY_GROUP_FOR_USER(SESSION_USER, 'CHEF') = 1
ENFORCED FOR ALL ACCESS
ENABLE;
```

Dieses Mal ist die Bedingung sehr viel komplexer aufgebaut. Es wird geprüft, ob das beim Kunden hinterlegte Gebiet dem zugeordneten Verkaufsgebiet des Verkäufers entspricht. Zu diesem Zweck müssen die Tabellen VERTRIEBS-MITARBEITER und PERSONAL verknüpft werden, da das Verkaufsgebiet und der Name des Mitarbeiters in unterschiedlichen Tabellen gespeichert wurden.

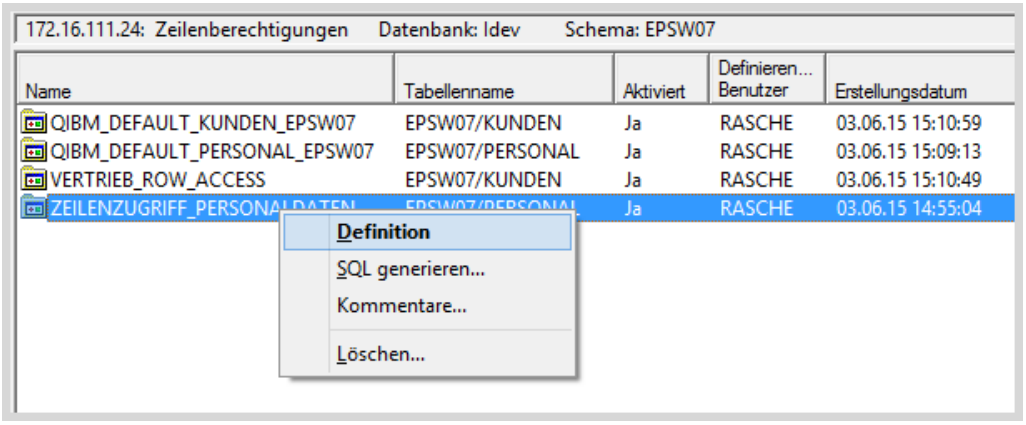
7.3.9.5.3 Zeilenberechtigungen anzeigen

Welche Zeilenberechtigungen erteilt wurden – und ob sie aktiviert oder deaktiviert wurden – können Sie sehr schnell im IBM i Navigator prüfen. Öffnen Sie hierzu die „DATENBANK“ und das Schema mit den betroffenen Tabellen. Im Menüpunkt Zeilenberechtigungen finden Sie alle erteilten Berechtigungen:



Zeilenberechtigungen anzeigen

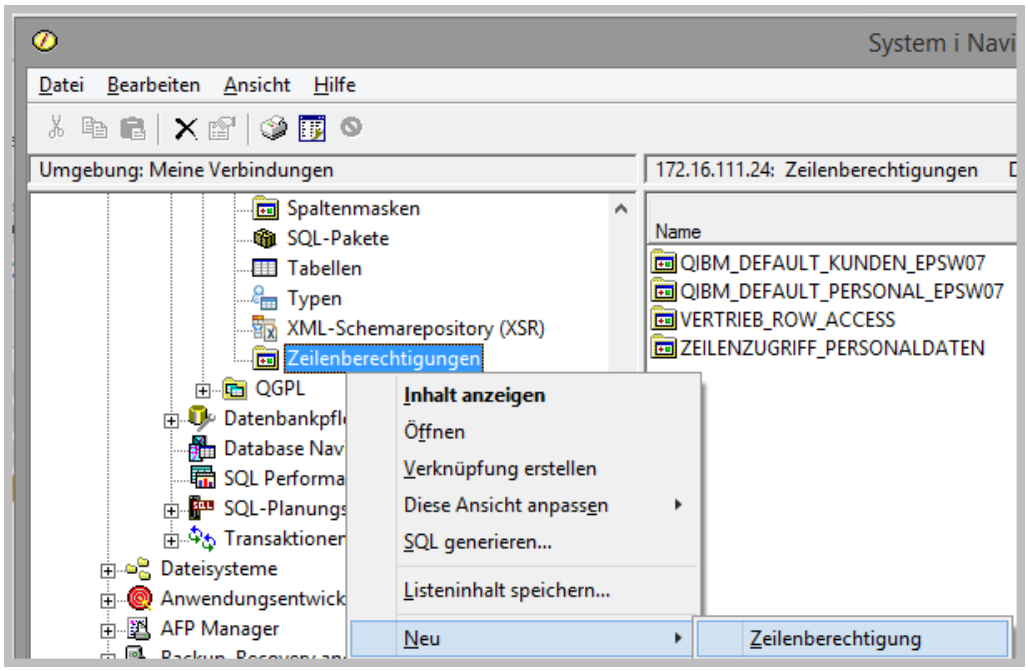
Von hier aus ist es auch sehr einfach, die Berechtigungen zu löschen, das zugehörige SQL-Statement zu generieren oder die Definition anzuzeigen:



Zeilenberechtigungen bearbeiten

7.3.9
Seite 24

Und wenn Sie sich das Kontextmenü der ZEILENBERECHTIGUNGEN ansehen, werden Sie feststellen, dass Sie auch problemlos mit der grafischen Oberfläche die Zeilenberechtigungen erstellen können:



Zeilenberechtigungen mit dem Navigator erstellen (1)

Zeilenberechtigungen mit dem Navigator erstellen (2)

7.3.9.5.4 Spaltenberechtigungen erstellen

Sie können aber nicht nur Berechtigungen für die Zeilen einer Tabelle erteilen, sondern auch die Anzeige von einzelnen Spalten steuern. Mit dem neuen SQL-Befehl CREATE OR REPLACE MASK können Spalteninhalte maskiert werden. Auf diesem Weg können sensible Daten durch die hinterlegten Regeln unsichtbar gemacht werden.

Um die Wirkung wiederum anhand der Gehaltsspalte der Tabelle PERSONAL zu demonstrieren, deaktiviere ich zunächst die zuvor hinterlegte Zeilenberechtigung:

```
ALTER TABLE PERSONAL DEACTIVATE ROW ACCESS CONTROL;
```

```
CREATE OR REPLACE MASK GEHALTS_MASKE
ON PERSONAL
FOR COLUMN MONATSGEHALT
RETURN
CASE WHEN UPPER(NACHNAME) = SESSION_USER
       THEN MONATSGEHALT
       WHEN VERIFY_GROUP_FOR_USER(SESSION_USER, 'PERSONAL' ) = 1
       THEN MONATSGEHALT
ELSE 0
END
ENABLE;
```

Wie zuvor, vergeben wir zunächst wieder einen Namen für die Spaltenberechtigung und bestimmen die Tabelle. Danach geben wir bekannt, für welche Spalte die Berechtigung gesetzt werden soll. In meinem Beispiel verwende ich erneut die Spalte MONATSGEHALT.

Mit der Klausel RETURN legen wir fest, wie der Inhalt der Spalte MONATSGEHALT dargestellt werden soll. Hierbei ist die Anweisung CASE behilflich. Der einzelne Mitarbeiter darf sein eigenes Gehalt sehen – allerdings nicht die Gehälter der Kollegen. Die Mitarbeiter der Personalabteilung erhalten Zugriff auf die Gehälter aller Mitarbeiter. Alle anderen Benutzer sehen anstelle des Gehalts den Wert 0.

Damit die Regel implementiert werden kann, müssen wir wieder den ALTER TABLE-Befehl verwenden. Achten Sie darauf, dass Sie jetzt COLUMN ACCESS aktivieren und nicht wie zuvor ROW ACCESS CONTROL.

```
ALTER TABLE PERSONAL ACTIVATE COLUMN ACCESS CONTROL;;
```

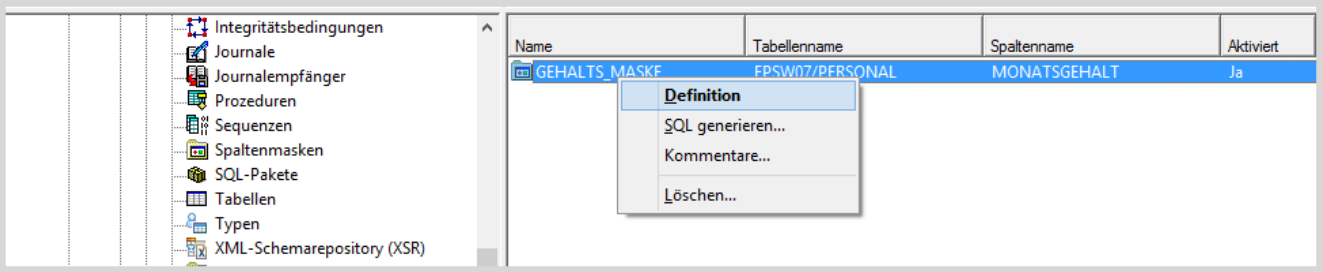
7.3.9
Seite 26

Bevor oder spätestens kurz nachdem Sie die Spaltenberechtigungen aktiviert haben, müssen Sie außerdem noch Ihre Funktionen (UDF) und Trigger, die auf die jetzt maskierten Spalten zugreifen, neu erstellen und das Schlüsselwort SECURED oder NOT SECURED als Eigenschaft hinzufügen:

```
CREATE FUNCTION xxxxx (
                  INDAT DECIMAL(8, 0)
                  )
        RETURNS DATE
        LANGUAGE SQL
        SPECIFIC xxxxxx
        NOT DETERMINISTIC
        READS SQL DATA
        CALLED ON NULL INPUT
        SECURED -- NOT SECURED
        SET OPTION
```

7.3.9.5.5 Spaltenberechtigungen anzeigen

Welche Spaltenberechtigungen erteilt werden – und ob diese aktiviert oder deaktiviert sind – können Sie wieder schnell im IBM i Navigator prüfen. Öffnen Sie hierzu die „DATENBANK“ sowie das Schema mit den betroffenen Tabellen. Der Menüpunkt „Spaltenmasken“ zeigt die gesuchten Informationen:

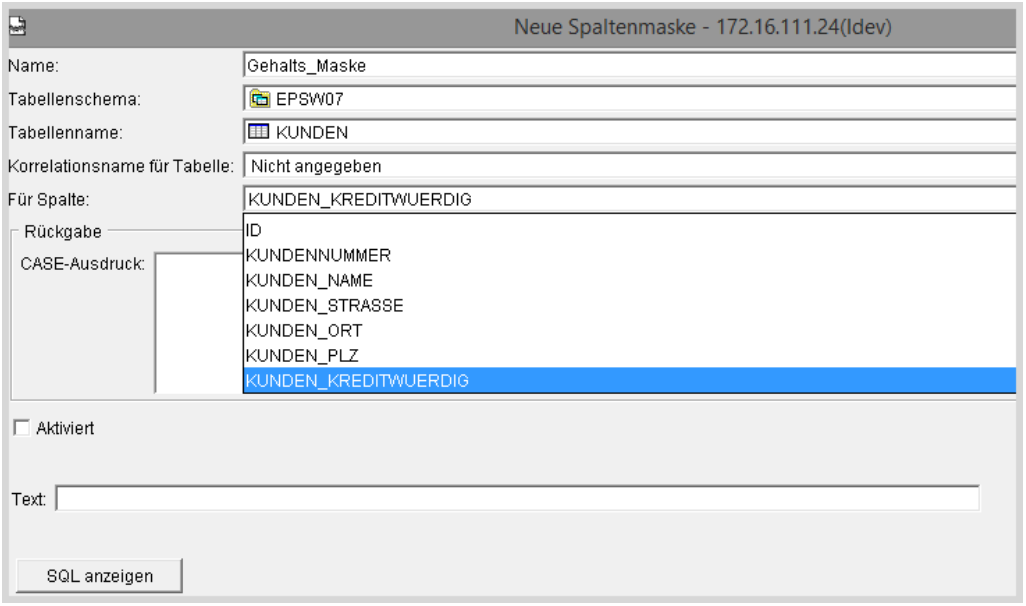


Spaltenberechtigungen anzeigen

Ebenso wie bei den Zeilenberechtigungen, können Sie die bestehenden Spaltenmasken löschen, das zugehörige SQL-Statement generieren oder sich die Definition anzeigen lassen.

7.3.9
Seite 28

Und wenn Sie sich das Kontextmenü der Spaltenmasken ansehen, werden Sie feststellen, dass Sie auch problemlos mit der grafischen Oberfläche die Spaltenmasken erstellen können.



Spaltenberechtigungen im Navigator erstellen

Im grafischen Interface werden Sie dabei gut unterstützt, indem beispielsweise die Spalten der ausgewählten Tabelle angezeigt werden.

7.3.9.5.6 RCAC-Bedingungen und die SQL-Indexierung

Im vorangegangenen Kapitel haben wir Zugriffsberechtigungen auf Zeilen und Spalten einer Tabelle mittels Row and Column Access (RCAC) implementiert. Diese Berechtigungen werden nicht über das Betriebssystem, sondern durch das DBMS gewährt. Nun stellt sich natürlich die Frage: Wie wirken sich diese DBMS-Berechtigungen auf die Administration der Datenbankobjekte aus? Gibt es gar Performance-Einbußen, wenn mit RCAC gearbeitet wird?

Schauen wir uns hierzu das zuvor verwendete Beispiel unter dem Gesichtspunkt der Performance etwas genauer an:

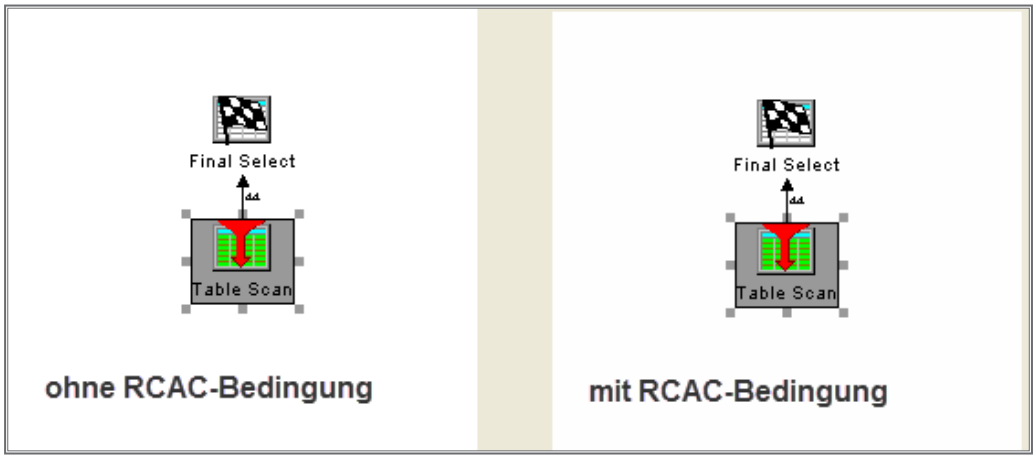
```
CREATE or REPLACE PERMISSION ZEILENZUGRIFF_PERSONALDATEN
ON PERSONAL AS PERSONAL
FOR ROWS WHERE (
VERIFY_GROUP_FOR_USER ( SESSION_USER , 'RASCHE' ) = 1
AND UPPER ( PERSONAL . NACHNAME ) = SESSION_USER
)
OR
(
VERIFY_GROUP_FOR_USER ( SESSION_USER , 'PERSONAL' ) = 1
)
ENFORCED FOR ALL ACCESS
ENABLE;

ALTER TABLE PERSONAL ACTIVATE ROW ACCESS CONTROL
```

Was passiert, wenn wir die Tabelle PERSONAL in unseren SQL-Abfragen verwenden? Nehmen wir hierfür zunächst eine ganz einfache Abfrage:

```
SELECT * FROM PERSONAL;
```

7.3.9
Seite 30

Wenn Sie Pech haben, werden Sie beim Ausführungsplan zunächst gar keine Unterschiede feststellen. Dennoch muss es Unterschiede geben, da die Zeilenbedingungen beim Datenabruf berücksichtigt werden müssen.



Ausführungspläne im Vergleich

SQL verwendet anscheinend in beiden Fällen einen Table-Scan. Schaut man sich aber das Table Scan-Icon etwas genauer an, so wird man feststellen, dass aufgrund der RCAC-Implementierung jeder Tabelleneintrag auf den Nachnamen geprüft wird: Deutlich wird daran auch, dass die Zeilenbedingungen bereits für den Datenabruf relevant sind.

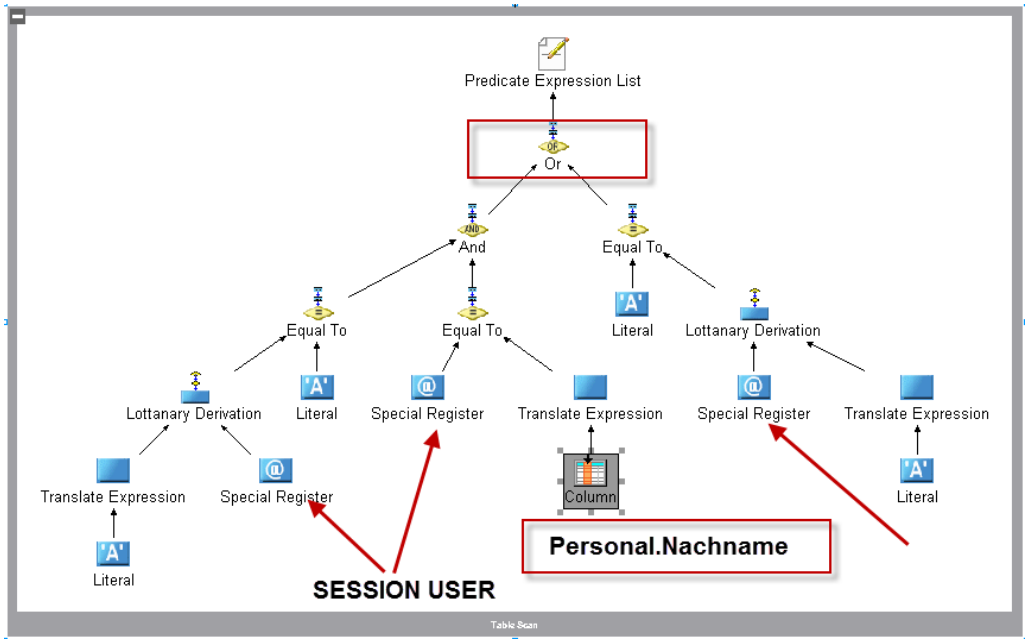
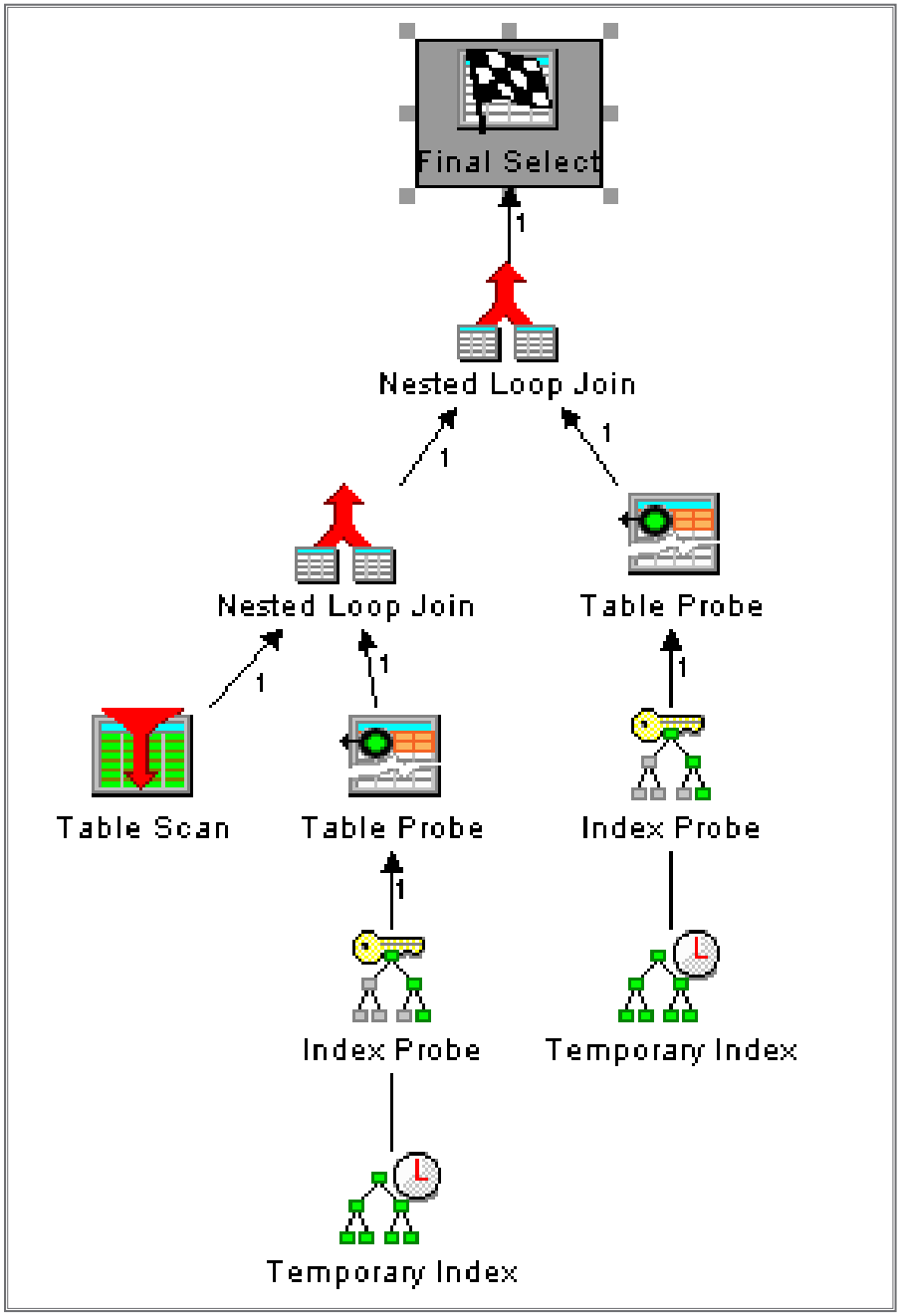


Table-Scan-Darstellung mit RCAC-Implementierung

Und was könnte gar passieren, wenn Ihre Abfragen komplexer werden und Sie mehr Datensätze verarbeiten wollen als ich sie in meinen Beispieltabellen habe?

```
SELECT …….
FROM ABTEILUNGEN A LEFT JOIN PERSONAL P
ON A.ABTEILUNGSNUMMER = P.ABTEILUNG
JOIN VERTRIEBSMITARBEITER V
ON P.PERSONALNUMMER = V.PERSONALNUMMER
WHERE ABTEILUNGSNUMMER IN ('A00', 'B01', 'C01') ;
```

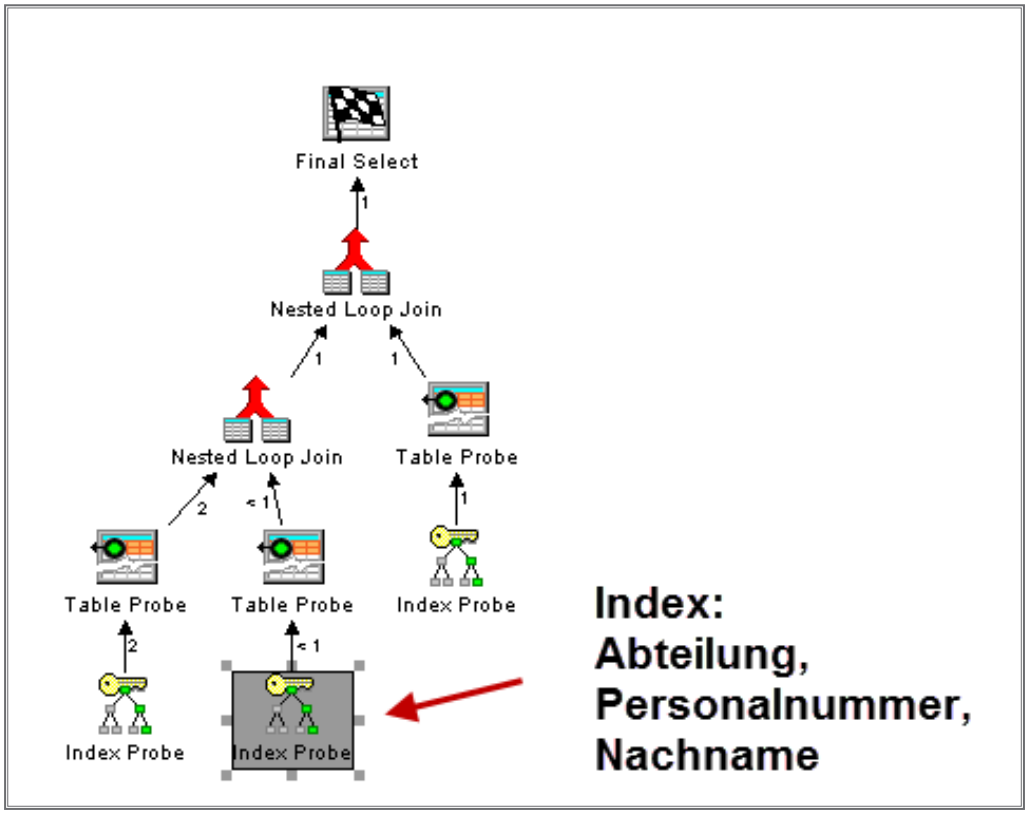


Performance-Einbuße durch Zeilenbedingungen

Um die Zeilenberechtigungen zu prüfen, könnte der SQL-Optimizer ggf. die alten Ausführungspläne verwerfen und im schlimmsten Fall eigene temporäre Indizes implementieren. Wenn Sie also keine Performance-Einbußen im SQL-Umfeld in Kauf nehmen wollen, müssen Sie ggf. Ihre Indizes überarbeiten.

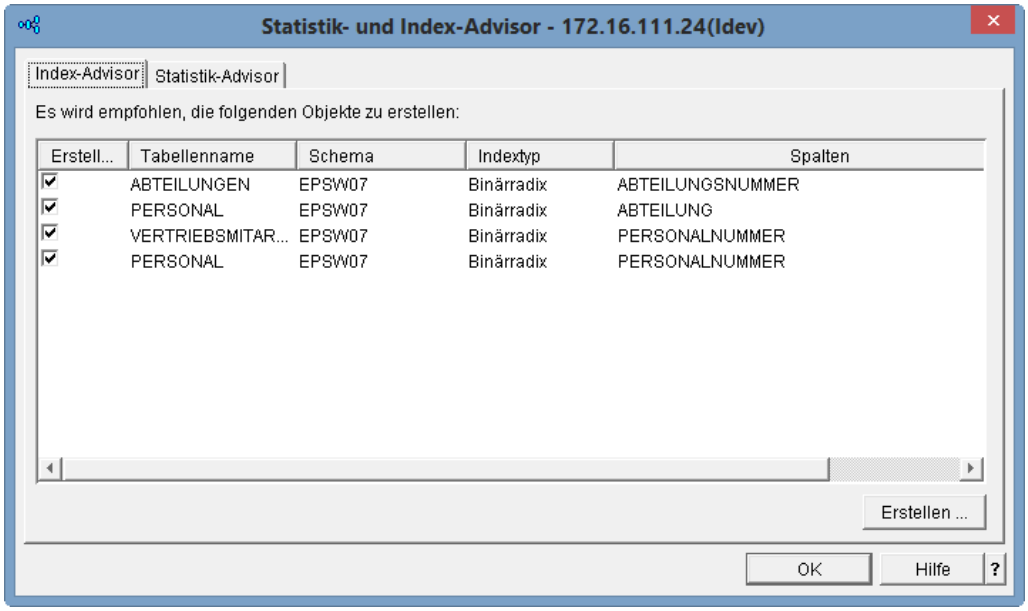
Schauen Sie sich an, was passiert, wenn ich die RCAC-Zeilenbedingung in den Indizes berücksichtige:

```
CREATE INDEX I_PERSONAL ABTEILG_PERSNR_NNAME
ON PERSONAL( ABTEILUNG, PERSONALNUMMER, NACHNAME ) ;
```



Berücksichtigung der RCAC-Bedingung bei der Indexierung

Ein abschließendes Wort zum Index Advisor, den nach wie vor viele Programmierer in der Programmierung als alleinigen Ratgeber verwenden. Laut IBM soll der Index Advisor bereits in den Indexempfehlungen die RCAC-Bedingungen berücksichtigen. In meinem Beispiel kann der Index Advisor für das oben verwendete Statement aber weder erkennen, dass die Berücksichtigung der Abteilungsnummer in Kombination mit der Personalnummer eine gute Wahl ist noch berücksichtigt er die RCAC-Bedingung! Dieses Verhalten mag vielleicht meinem TR-Stand geschuldet sein; seien Sie aber auf alle Fälle skeptisch und hinterfragen Sie die Empfehlungen des Advisors.



Die Indexierungsempfehlungen

FAZIT:

Wenn Sie RCAC-Zeilenbedingungen verwenden, müssen Sie Ihre Indizes ggf. überarbeiten. Dieses Problem haben Sie nicht, wenn Sie mit Masken arbeiten. Der Grund hierfür? Die Maskierung der Spalten erfolgt erst unmittelbar vor der Präsentation des Result Sets, spielt also für den Datenabruf überhaupt keine Rolle.

Immer wieder werde ich in diesem Zusammenhang gefragt, wie sich RCAC mit nativen SQL-Interfaces wie beispielsweise Query/400 oder OPNQRYF verhält?

Das Interface spielt seit i7.2 für die Wirkung von RCAC keine Rolle, wie das nachfolgende Beispiel zeigt:

```
RUNQRY  EPSW07/PERSONAL

                          Bericht anzeigen
Abfrage . . :   EPSW07/PERSONAL              Breite des Berichts
Anfang auf Zeile . . . . . .  _____     Anfang in Spalte . . . .
Zeile  ....+....1....+....2....+....3....+....4....+....5....+...
       PERSONALNUMMER  VORNAME                          NACHNAME
000001     993400      Carmen                           RASCHE
000002     999999      Carmen                           Rasche
****** ********  Ende des Berichts  ********
```

RUNQRY und RCAC-Bedingungen

7.3.9
Seite 34

Allerdings hat IBM – wie bereits in früheren Releases – auch in i7.2 Erweiterungen an der SQE vorgenommen. Ab Release 7.2 wird unter anderem bei den Befehlen

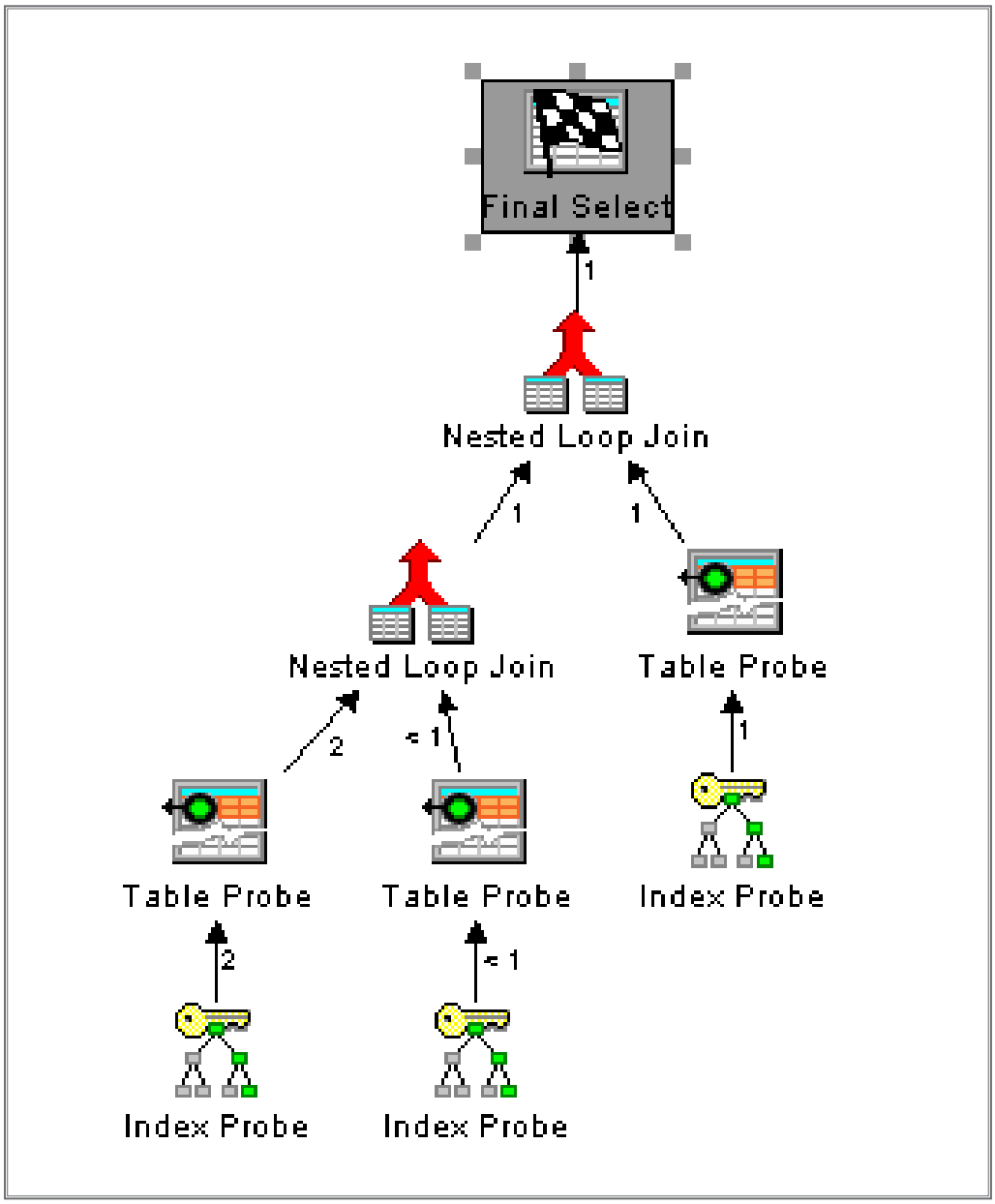
- WRKQRY (Mit Abfragen arbeiten),
- RUNQRY (Abfrage ausführen),
- OPNQRYF (Abfragedatei öffnen) und
- OPNDBF (Datenbankdatei öffnen)

für den Zugriff auf die Basisdatenbank standardmäßig die SQE verwendet. Dazu gehören auch alle Öffnungsvorgänge einer Datenbankdatei, für die Zeilenberechtigungen oder Spaltenmasken definiert und aktiviert sind. Ausnahme: Die Datei wird als reine Ausgabedatei geöffnet. Allerdings gilt auch hier: Achten Sie auf die Indizierung!

Die QAQQINI-Option SQE_NATIVE_ACCESS
Diese Änderungen in Release i7.2 können übrigens ein verändertes Ergebnis bei den Query/400-Reports bewirken, denn die SQE-Implementierung kann dazu führen, dass die Ergebnisliste der Befehle WRKQRY (Work Query), RUNQRY (Run Query) oder OPNQRYF (Open Query File) eine andere Sortierung aufweist. Wenn eine Abfrage ausgeführt wird, ohne explizit anzugeben, dass die Ergebnisse in einer bestimmten Reihenfolge zurückgegeben werden sollen, wählen sowohl die SQE als auch die CQE den am besten geeigneten Plan aus. Das bedeutet, dass sowohl SQE als auch CQE die Ergebnisse möglicherweise nicht in der Reihenfolge der Schlüsseldatei ausgeben. Da die CQE aber über weniger Funktionalität verfügt als die SQE, ist die Wahrscheinlichkeit, dass die Ergebnisse von der CQE in der Reihenfolge der Schlüsseldatei ausgegeben werden, größer als bei der SQE. Wenn bei WRKQRY, RUNQRY oder OPNQRYF die Zeilensortierung wichtig ist, müssen Sie daher die Sortierfolge **explizit** angegeben.
Falls es Ihnen nicht möglich ist, kurzfristig die Querys zu überarbeiten, könnten Sie vorübergehend diese Neuerung deaktivieren, indem Sie in der Abfrageoptionsdatei QAQQINI die Standardeinstellung der Werts SQE_NATIVE_ACCESS auf *NO ändern. Auf die RCAC-Bedingung hat diese Änderung keinen Einfluss, aber die Umstellung kann auf Ihre Ausführungspläne und damit auch auf die Laufzeiten Ihrer sonstigen SQL-Abfragen erheblichen Einfluss haben. Standardmäßig wird für das Attribut SQE_NATIVE_ACCESS der Wert *DEFAULT verwendet, so dass durch den Optimizer festgelegt wird, ob der Wert *YES oder *NO verwendet wird. Ändern Sie nun diese Einstellung auf *NO, damit Ihre Query/400-Reports wieder richtig sortiert werden, ändern sich ggf. auch andere SQL-Ausführungspläne. Schauen Sie sich die Wirkung am nachfolgenden Beispiel an:

Beispiel 1: SQE_NATIVE_ACCESS = *YES

```
SELECT * FROM ABTEILUNGEN A LEFT JOIN PERSONAL P
ON A.ABTEILUNGSNUMMER = P.ABTEILUNG
JOIN VERTRIEBSMITARBEITER V ON P.PERSONALNUMMER =
V.PERSONALNUMMER
WHERE ABTEILUNGSNUMMER IN ('A00', 'B01', 'C01') ;
```

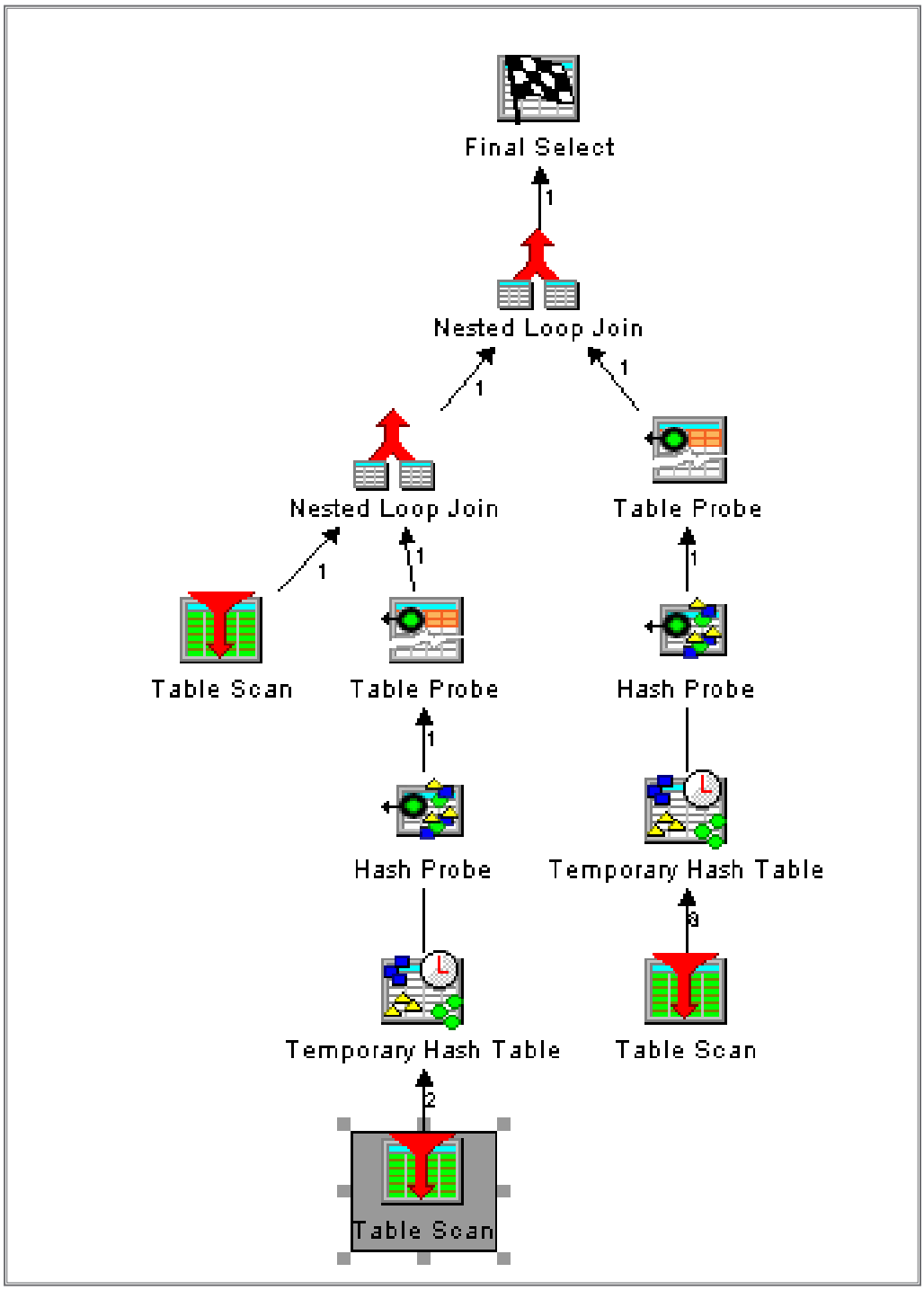


*SQE_NATIVE_ACCESS *YES*

7.3.9
Seite 36

Beispiel 2: SQE_NATIVE_ACCESS = *NO

Jetzt ändern wir die QAQQINI-Einstellungen auf SQE_NATIVE_ACCESS = *NO. Obwohl weder am SQL-Statement noch an den Indizes Änderungen vorgenommen wurden und obwohl nach wie vor die SQE das SQL-Statement ausführt, ist der Ausführungsplan vollständig geändert worden und die Laufzeiten brechen aufgrund der verwendeten Table Scans drastisch ein:



*SQE_NATIVE_ACCESS = *NO*

Gehen Sie also mit Änderungen der QAQQINI-Einstellungen sehr vorsichtig um und versuchen Sie – wenn möglich – mindestens die Einstellung *DEFAULT beizubehalten, wenn Sie keine allgemeinen SQL-Performance-Einbußen in Kauf nehmen wollen.

7.3.9.5.7 RCAC und Materialized Query Tables (MQT)

Nicht selten werden heute MQTs im Datawarehouse-Umfeld eingesetzt, um bessere Abfrageergebnisse zu erzielen. Da stellt sich natürlich die Frage, welche Auswirkungen haben die RCAC-Bedingungen in diesem Umfeld?

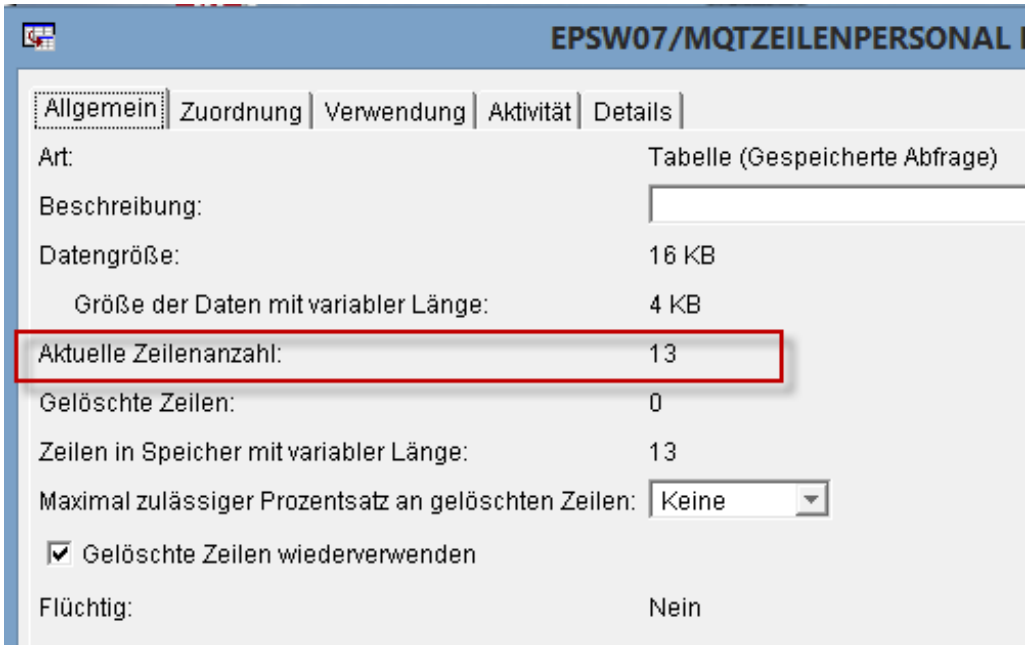
Die gute Nachricht vorweg: MQTs und RCAC-Bedingungen können friedlich koexistieren, wenn man einige Dinge berücksichtigt.

Schauen wir uns auch hierzu ein Beispiel an:

```
CREATE OR REPLACE TABLE /MQTZEILENPERSONAL FOR SYSTEM NAME MQT1
AS (
SELECT P.PERSONALNUMMER, VORNAME, NACHNAME, ABTEILUNG,
MONATSGEHALT, SOZIALVERSICHERUNGSNUMMER
FROM ABTEILUNGEN A LEFT JOIN PERSONAL P
ON A.ABTEILUNGSNUMMER = P.ABTEILUNG
LEFT JOIN VERTRIEBSMITARBEITER V
ON P.PERSONALNUMMER = V.PERSONALNUMMER
WHERE ABTEILUNGSNUMMER IN ('A00', 'B01', 'C01')
)
DATA INITIALLY IMMEDIATE
REFRESH DEFERRED
DISABLE QUERY OPTIMIZATION
MAINTAINED BY USER;
```

7.3.9
Seite 38

Die Initialisierung der MQT erfolgt problemlos und wird durch etwaige Zeilenbedingungen nicht beeinflusst. Das lässt sich sehr schnell überprüfen, wenn man sich die Dateibeschreibung dieser MQT anschaut:



Dateibeschreibung MQT1

Auch die Anweisung REFRESH TABLE ist von der Zeilenberechtigung nicht betroffen. Das heißt, der Benutzer RASCHE kann die Aktualisierung der MQT durchführen, obwohl er selbst nicht für alle Zeilen der Tabelle berechtigt ist.

Überraschend fällt allerdings anschließend die Abfrage der MQT aus:

```
SELECT *
FROM MqtZeilenPersonal;
```

PERSONAL...	VORNAME	NACHNAME	ABTEILUNG

Abfrage der Tabelle MQT1

Das Result Set ist leer. Wie kommt das? Wenn die MQT erstellt wird, wird die Zeilenzugriffssteuerung für die MQT implizit aktiviert. Es wird eine Default-Berechtigung generiert, die mit der Zeichenfolge QIBM beginnt. Diese Defaultzeilenberechtigung ist allerdings leer, das heißt, die Zeilenberechtigungen aus den Basistabellen werden nicht übernommen. Sie können diese Standardberechtigungen nicht löschen. Erst wenn Sie die entsprechende RCAC-Bedingung in der MQT deaktivieren, wird auch die QIBM-Berechtigung gelöscht.

172.16.111.24: Zeilenberechtigungen Datenbank: Idev Schema: EPSW07

Name	Tabellenname	Aktiviert	Definieren... Benutzer	Erstellungsdatum
QIBM_DEFAULT_MQT1_EPSW07	EPSW07/MQTPERSONAL	Ja	RASCHE	22.03.16 13:40:19
ZEILENZUGRIFF_PERSONALDATEN	EPSW07/PERSONAL	Ja	RASCHE	23.03.16 09:29:32

IBM Standardberechtigung

Warum werden die Bedingungen nicht mitkopiert? Stellen Sie sich vor, dass die Basistabellen in der MQT unterschiedliche RCAC-Zeilen- und Spaltenbedingungen aufweisen. Wie soll in diesen Fällen eine implizite Zeilenbedingung generiert werden?

Wir haben zwei Möglichkeiten, mit diesem Problem umzugehen:

```
ALTER TABLE MQTZeilenPersonal DEACTIVATE  ROW  ACCESS CONTROL;

SELECT *
FROM MqtZeilenPersonal;
```

PERSONAL...	VORNAME	NACHNAME	ABTEILUNG	MONATSG...
000010	CHRISTINE	HAAS	A00	24900.16
000110	VINCENZO	LUCCHESSI	A00	26333.52
000120	SEAN	O'CONNELL	A00	26333.52
200010	DIAN	HEMMINGER	A00	26333.52
200120	GREG	ORLANDO	A00	26333.52
100000	Katrin	MUELLER	A00	24660.16
993400	Carmen	RASCHE	A00	24660.16
000020	MICHAEL	MAYER	B01	30333.52
000030	SALLY	KWAN-CHANG	C01	25660.16

Deaktivierung der entsprechenden RCAC-Bedingungen

Wir können die RCAC-Bedingungen deaktivieren. Falls Sie Ihre MQTs für den Optimierungsprozess freigegeben haben, wird Ihnen vermutlich nichts anderes übrig bleiben. Anderenfalls laufen Sie Gefahr, dass Ihre SQL-Abfragen zukünftig falsche oder deutlich veränderte Ergebnisse liefern.

Falls Sie die MQTs direkt in den Abfragen verwenden oder die unterschiedliche Satzauswahl auch im Optimierungsprozess gewünscht ist, können eigene RCAC-Zugriffsberechtigungen erstellt werden:

7.3.9
Seite 40

```
ALTER TABLE MQT1 DEACTIVATE ROW ACCESS CONTROL;


CREATE or REPLACE PERMISSION EPSW07/ZEILENZUGRIFF_MQT1
ON MQT1 AS MQTPERSONAL
FOR ROWS WHERE (
VERIFY_GROUP_FOR_USER ( SESSION_USER , 'RASCHE' ) = 1
AND UPPER ( NACHNAME ) = SESSION_USER
)
OR
(
VERIFY_GROUP_FOR_USER ( SESSION_USER , 'PERSONAL' ) = 1
)
ENFORCED FOR ALL ACCESS
ENABLE ;


ALTER TABLE MQT1 ACTIVATE ROW ACCESS CONTROL;
```

Fazit:

Es ist sicherlich einiges an Planung notwendig, wenn Sie bereits MQTs einsetzen und die neuen RCAC-Bedingungen implementieren wollen.

7.3.9.5.8 Datenbankadministration und RCAC

Datenbankdateien duplizieren (CRTDUPOBJ / CPYLIB)
Auch bei der Administration der Tabellen müssen Sie einige Dinge beachten. Was passiert, wenn Sie zum Beispiel Daten mit den Befehlen CPYF, CRTDUPOBJ oder CPYLIB kopieren?

Abhängig von den implementierten Regeln sind hier unterschiedliche Dinge zu beachten. Betrachten wir zunächst einmal das Verhalten, wenn in der Tabelle Zeilenberechtigungen implementiert wurden.

Die Ausgangsdatei PERSONAL enthält die schon mehrfach gezeigte Zeilenberechtigung. Nun verwendet der Benutzer RASCHE den Befehl CRTDUPOBJ (Create Duplicate Object), um die Datei zu kopieren:

```
                    Doppeltes Objekt erstellen (CRTDUPOBJ)

Auswahl eingeben und Eingabetaste drücken.

Von Objekt . . . . . > PERSONAL    Name, generisch*, *ALL
Von Bibliothek . . . . . > EPSW07     Name, *LIBL, *CURLIB
Objektart . . . . . > *FILE     *ALL, *ALRTBL, *AUTL...
     + für weitere Werte          _______
Nach Bibliothek . . . . . > EPSW07     Name, *FROMLIB, *SAME...
Neues Objekt . . . . . > PERSONAL2    Name, *OBJ, *SAME
Von ASP-Einheit . . . . . *          Name, *, *CURASPGRP, *SYSBAS
In ASP-Einheit . . . . . *ASPDEV     Name, *ASPDEV, *...
Daten duplizieren . . . . . *NO          *NO, *YES
Einschränkungen duplizieren . . *YES         *YES, *NO
Auslöser duplizieren . . . . . *YES         *YES, *NO
Doppelt vorhandene Datei-IDs . . *NO          *NO, *YES
Doppelte Zugriffssteuerung . . . *NONE        *ALL, *ROW, *COL, *NONE
                                  Der Parameter ACCCTL i7.2
```

Neuerung CRTDUPOBJ

Sie können in der Abbildung oben sehen, dass die Daten nicht mitkopiert werden sollen. Mit Hilfe des neu eingeführten Parameters ACCCTL steuern Sie jetzt, ob alle RCAC-Berechtigungen, nur die Zeilen- oder nur die Spaltenberechtigungen kopiert werden sollen. Es ist auch möglich, das Duplikat vollständig ohne diese Berechtigungen zu erstellen. Letzteres ist allerdings nur möglich, solange Sie keine Daten kopieren.

Anders verhält es sich, wenn Sie die Daten ebenfalls kopieren. Jetzt müssen die Berechtigungen übernommen werden. Hat die Datei – wie in meinem Beispiel – nur Zeilenberechtigungen, so ist es ausreichend, diese zu übernehmen. Gleiches gilt, wenn nur Spaltenberechtigungen implementiert wurden. Sind sowohl Zeilen- als auch Spaltenberechtigungen vorhanden, müssen Sie den Wert *ALL im Kopiervorgang verwenden, anderenfalls erhalten Sie stets die folgende Fehlermeldung:

```
5>> CRTDUPOBJ OBJ(PERSONAL) FROMLIB(EPSW07) OBJTYPE(*FILE) TOLIB(EPSW07) NEWOB
    J(PERSONAL2) DATA(*YES) ACCCTL(*NONE)
    Integritätsbedingung wurde hinzugefügt.
    Integritätsbedingung wurde hinzugefügt.
    Integritätsbedingung wurde hinzugefügt.
    3 Integritätsbedingung(en) der Datei PERSONAL2 hinzugefügt.
    Fehler bei DataLink-, UDT-, Zugriffssteuerungs-, LOB- oder XML-Feld,
      Ursachencode 11.
    Datei PERSONAL2 in EPSW07 kann nicht dupliziert werden.
    0 Objekte dupliziert. 1 Objekte nicht dupliziert.
```

Fehler beim Kopieren der Objekte mit RCAC-Bedingung

Analog zum Befehl CRTDUPOBJ verhält sich auch der Befehl CPYLIB (Copy Library). Auch hier finden wir seit Release i7.2 den Parameter ACCCTL, um die Zeilen- und Spaltenberechtigungen zu verwalten.

Das Handling scheint einfach und unkompliziert zu sein, birgt aber eine Gefahr: Stellen Sie sich vor, Sie verwenden innerhalb der RCAC-Bedingungen andere Tabellen, um den Zugriff zu verifizieren. In diesem Fall würden diese Tabellennamen qualifiziert – also inklusive Schemanamen – als RCAC-Bedingung im neuen Dateiobjekt hinterlegt werden. Leider werden die Schemanamen im Kopiervorgang nicht modifiziert, so dass das Duplikat ggf. auf falsche Objekte referenziert. Um diesem Phänomen vorzubeugen, hilft nur eins: **Nach dem Kopiervorgängen müssen die RCAC-Regeln neu erstellt werden.**

Anders verhält es sich bislang, wenn Sie die Anweisung CREATE TABLE … LIKE verwenden, um das Dateiobjekt zu duplizieren. Bislang gibt es keine Copy-Optionen, die die RCAC-Bedingungen zu steuern helfen. Das Duplikat wird ohne Zeilenzugriffsrechte oder Spaltenmasken erstellt. Diese Attribute müssen daher nachträglich eingepflegt werden.

Datensätze duplizieren (CPYF)

Grundsätzlich anders verhält es sich, wenn Sie nicht auf die Objekte, sondern direkt auf die Daten zugreifen, beispielsweise mit dem Befehl CPYF (Copy File) oder mit einem entsprechenden SQL-Befehl wie CREATE TABLE … AS (SELECT …) oder einem entsprechenden INSERT-Befehl. Hier sind folgende Szenarien zu unterscheiden:

a) Die Zieldatei verfügt über RCAC-Bedingungen:

Fügen Sie Daten in die Tabelle PERSONAL ein, so müssen Sie für diese Daten auch berechtigt sein. Der Benutzer RASCHE kann zum Beispiel das folgende SQL-Statement nicht fehlerfrei ausführen:

```
INSERT INTO PERSONAL(Persnr, Vorname, Nachname)
VALUES('993400', 'Bertha', 'Rusch')   ;
```

Zeilenberechtigung erlaubt nur Zugriff auf den Nachnamen RASCHE

```
SQL-Status: 22542
Vendorencode: -20471
Nachricht: [SQ20471] INSERT oder UPDATE entspricht nicht den Zeilenberechtigungen. Ursache . . . . : Für
PERSONAL in EPSW07 wird zeilenbezogene Zugriffssteuerung erzwungen. Daher werden alle Versuche,
Zeilen in der betreffenden Tabelle einzufügen oder zu aktualisieren, geprüft, um sicherzustellen, dass die
daraus resultierenden Zeilen den für die Tabelle definierten Zeilenberechtigungen entsprechen. INSERT oder
UPDATE konnte nicht ausgeführt werden, da eine daraus resultierende Zeile eine oder mehrere
Zeilenberechtigungen für PERSONAL in EPSW07 nicht erfüllt hat. Fehlerbeseitigung:  Die Daten, die
eingefügt oder aktualisiert werden sollen, so ändern, dass sie den für die Zeilenberechtigungen definierten
Regeln entsprechen.
```

Vorsicht bei INSERT

Die RCAC-Berechtigungen müssen sich mit den Inhalten der einzufügenden Zeilen decken.

b) Die Ausgangsdatei verfügt über RCAC-Bedingungen: Werden die Daten aus einer Datei kopiert, die RCAC-Berechtigungen aufweist, werden nur die Datensätze kopiert, für die der Anwender berechtigt ist.

```
                          Datei kopieren (CPYF)

Auswahl eingeben und Eingabetaste drücken.

Ausgangsdatei  . . . . . > PERSONAL     Name
  Bibliothek . . . . . . >   EPSW07     Name, *LIBL, *CURLIB
Zieldatei  . . . . . . . > PERSONAL2    Name, *PRINT
  Bibliothek . . . . . . >   EPSW07     Name, *LIBL, *CURLIB
Von Teildatei  . . . . . .   *FIRST     Name, generisch*, *FIRST, *ALL
Nach Teildatei oder Kennsatz .  *FIRST     Name, *FIRST, *FROMMBR, *ALL
Sätze ersetzen od. hinzufügen  > *REPLACE   *NONE, *ADD, *REPLACE...
Datei erstellen  . . . . . > *YES       *NO, *YES
Druckformat  . . . . . .   *CHAR      *CHAR, *HEX
```

Der Befehl CPYF

7.3.9
Seite 44

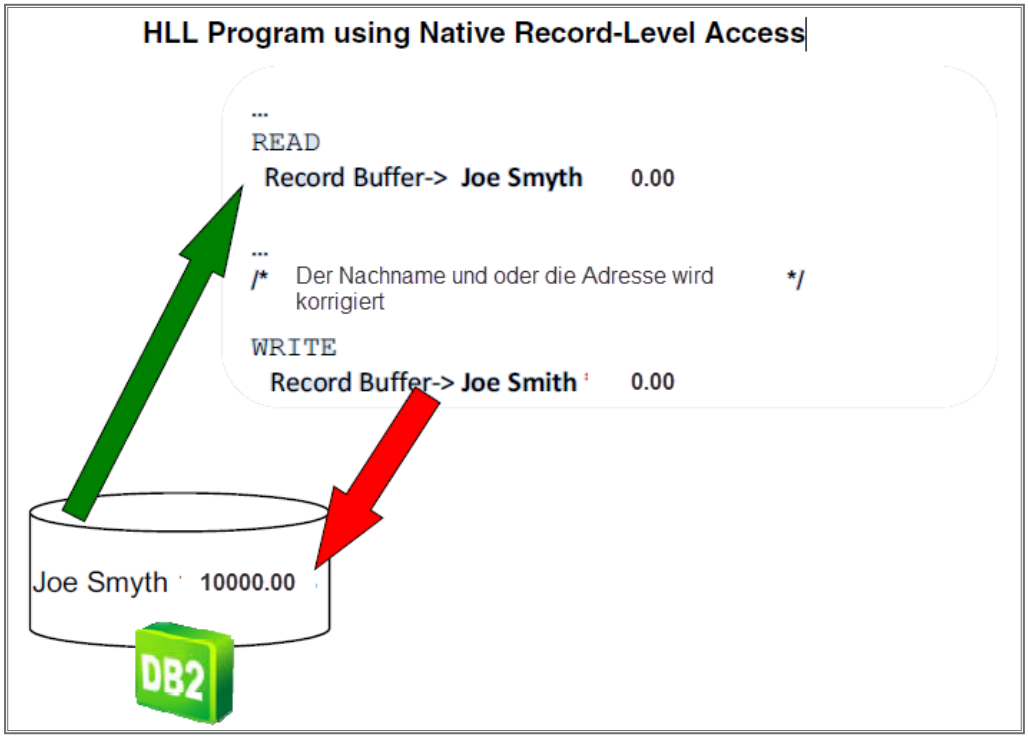
In meinem Beispiel würde die Zieldatei PERSONAL2 für diesen Fall also nur einen Datensatz enthalten. Dieses Verhalten gilt im Übrigen nicht nur für den Befehl CPYF (Copy File), sondern auch für den Exportbefehl CPYTOIMPF (Copy to Importfile)!

Bedenken Sie, dass Sie ggf. Datenexporte und Batch-Programme als Scheduler-Jobs im System haben, um Daten zu archivieren oder zu verdichten. Für diese Programme (= Jobs) ist der Benutzername zukünftig entscheidend. Und vergessen Sie bitte nicht, dass es bei Ihnen ggf. Berechtigungsübernahmen geben könnte. Auch in diesen Fällen ist der Benutzername zukünftig entscheidend für die Ergebnisse.

7.3.9.5.9 RCAC in der Programmierung

Auch für die Programmierung sind zukünftig einige Dinge zu berücksichtigen: Stellen Sie sich vor, die Gehaltsspalte ist in der Personaldatei maskiert – so wie ich es in den vorangegangenen Kapiteln gezeigt habe. In meinen Spaltenberechtigungen hatte ich festgelegt, dass das Monatsgehalt stets den Wert 0 enthält, wenn der Benutzer fremde Personaldaten liest. Was könnte jetzt passieren, wenn der Benutzer innerhalb eines RPG-Programms berechtigt ist, die Adressdaten der Mitarbeiter zu bearbeiten?

Ein RPG-Programm, das den zu ändernden Satz nativ liest (Native Record-Level Access), füllt den vollständigen Satzpuffer. Im Puffer steht somit der maskierte Wert, also die 0.



Satzverarbeitung im HLL-Programm bei native Record-Access

Anschließend wird der Name oder die Adressdaten korrigiert und der Satz wird in der Datei aktualisiert. Da wir aber im RPG-Programm mit Nativ Record-Level-Access stets ganze Sätze lesen und schreiben, könnte auf diesem Weg der ursprüngliche Gehaltsbetrag verloren gehen.

7.3.9
Seite 46

Um diese „Unfälle“ zu vermeiden, hat IBM neue Prüfbedingungen (CHECK-Constraints) eingeführt, die Sie entweder nachträglich mit dem Befehl ALTER TABLE oder direkt im CREATE TABLE-Statement verwenden können.

```
ALTER TABLE PERSONAL
ADD CONSTRAINT  mask_Gehalt
CHECK(Monatsgehalt <> 0)
ON UPDATE VIOLATION PRESERVE Monatsgehalt
ON INSERT VIOLATION SET Monatsgehalt = DEFAULT;
```

Mit Hilfe dieser Bedingungen ist sichergestellt, dass bei einer UPDATE-Anweisung der ursprüngliche Wert der Spalte MONATSGEHALT erhalten bleibt und bei einer INSERT-Anweisung anstelle der Werts 0 der Standardwert der Spalte MONATSGEHALT genutzt wird.

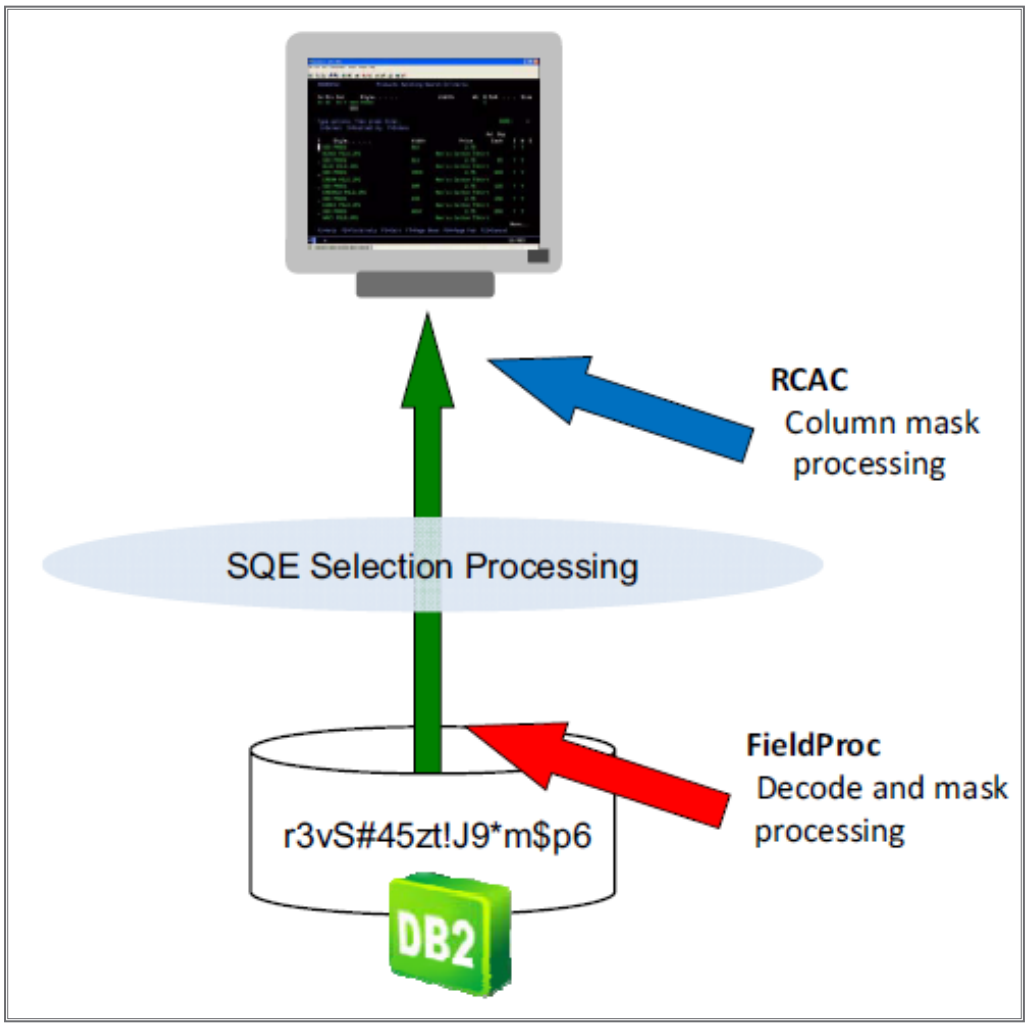
Alternativ könnten auch Before-Trigger helfen, das Problem in den Griff zu bekommen:

```
CREATE TRIGGER MASK_GEHALT BEFORE INSERT OR UPDATE ON PERSONAL
REFERENCING NEW ROW AS N OLD ROW AS O
FOR EACH ROW
MODE DB2ROW
SECURED
WHEN Monatsgehalt = 0
BEGIN
IF INSERTING THEN SET N.Monatsgehalt = DEFAULT;
ELSEIF UPDATING THEN SET N.Monatsgehalt = O.Monatsgehalt;
END IF;
END
```

Bei dem oben gezeigten Beispiel handelt es sich um einen Before-Trigger, der im Falle eines Insert-Befehls auch den Standardwert für die Initialisierung der Spalte „Monatsgehalt“ verwendet. Wird die Zeile hingegen aktualisiert, wird der Ursprungswert im Update verwendet. Damit der Trigger nicht grundsätzlich aktiviert wird, dürfen Sie die Bedingung (WHEN) nicht vergessen; und außerdem benötigt der Trigger die Eigenschaft SECURED, wenn er für eine Datei mit RCAC-Bedingungen implementiert wird. Sie sehen, auch für die zukünftige Programmierung ist einiges an Überlegungen notwendig, bevor man die RCAC-Bedingungen implementieren kann.

Aber nicht nur auf die RPG-Programme haben die RCAC-Bedingungen Einfluss – auch bestehende SQL-Anweisungen könnten plötzlich andere Ergebnisse liefern. Um diesen Sachverhalt zu verstehen, ist es wichtig, sich zu verdeutlichen, wann die RCAC-Bedingungen greifen.

Maskierungen
Anders als bei Feldprozeduren (Field Procedures) werden bei Spaltenbedingungen alle lokalen Selektionen (WHERE), alle Verknüpfungen (JOIN) und alle Gruppierungen (GROUP BY) auf die Originaldaten ausgeführt. Erst für die Präsentation des Result Sets werden die Daten maskiert:



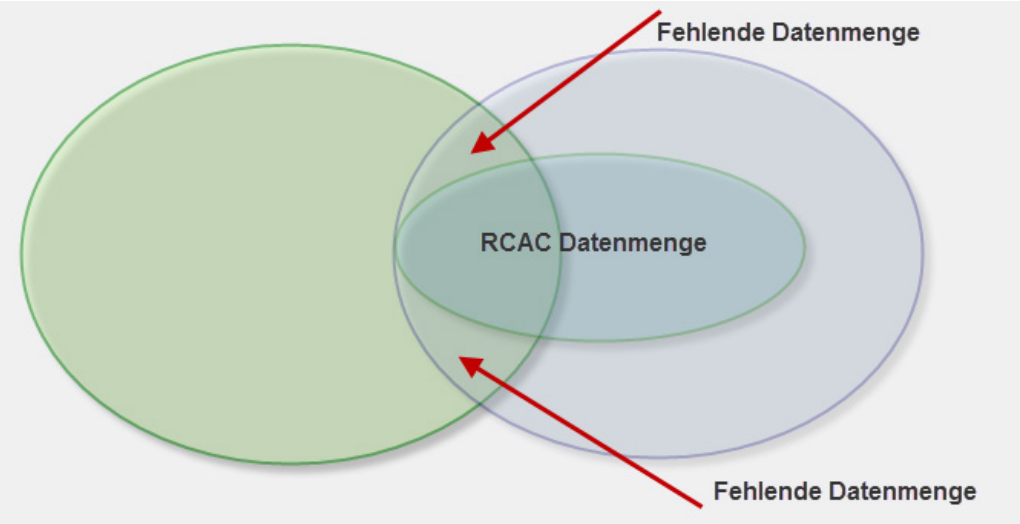
Quelle IBM Redbook „Row and Column Access Control Support in IBM DB2 for i“

Somit haben die Spaltenmasken keinen negativen Einfluss auf die Ergebnisse von Gruppierungen, Aggregatfunktionen oder JOIN-Statements.

7.3.9
Seite 48

Zeilenbedingungen

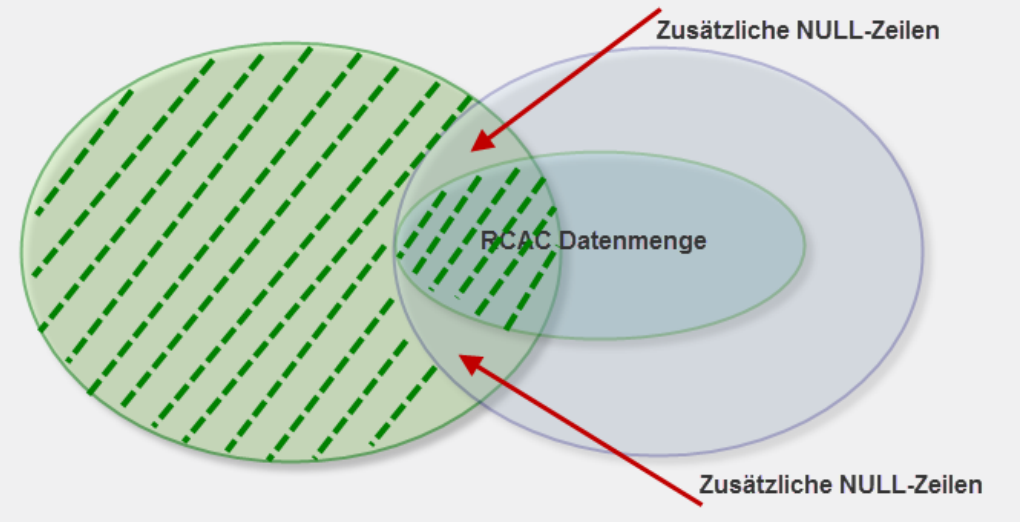
Anders verhält es sich mit Zeilenbedingungen. Zeilenbedingungen werden bereits im Datenzugriff realisiert und insbesondere für die JOIN-Statements kann es dabei zu veränderten Ausgaben kommen. So wird die Datenmenge beim INNER JOIN ggf. deutlich reduziert:



INNER JOIN und RCAC

Dadurch können sich natürlich auch die Ergebnisse der Aggregatfunktionen deutlich verändern.

Beim OUTER-JOIN hingegen wird die Anzahl der Zeilen mit NULL-Werten größer:



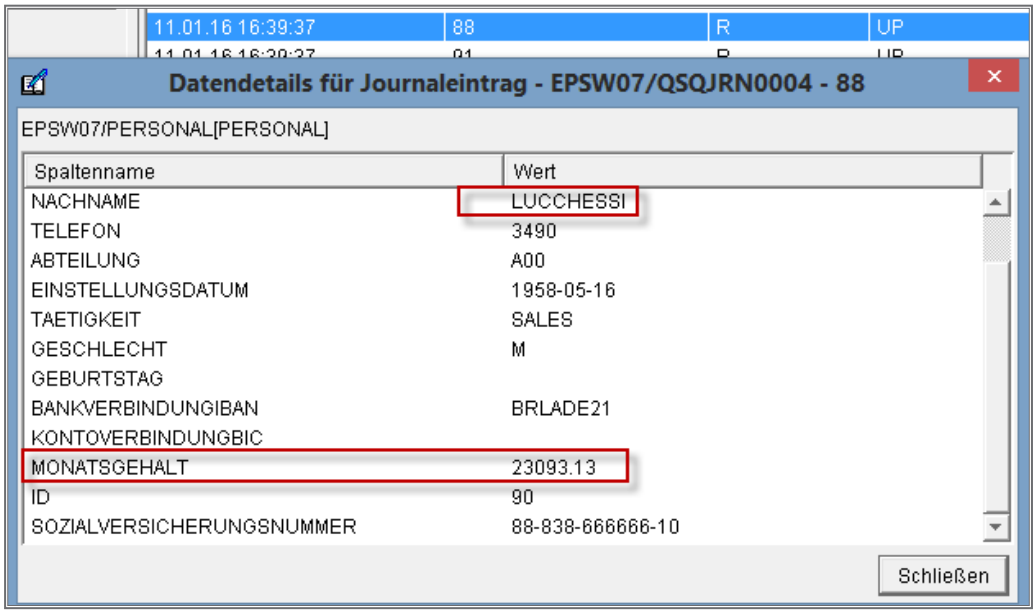
OUTER JOIN und RCAC

7.3.9.5.10 Sonstige Überlegungen

Neben den bereits genannten Überlegungen gibt es auch einige grundsätzliche Einschränkungen, die beachtet werden müssen:

- RCAC verhindert, dass diese Dateien weiterhin als programmbeschriebene Dateien im RPG-Umfeld verwendet werden können. Sie erhalten keine Fehlermeldung, wenn Sie die RCAC-Bedingung implementieren. Vielmehr wird das RPG-Programm eine Fehlermeldung senden, wenn auf die programmbeschriebene Datei zugegriffen wird und diese Datei RCAC-Bedingungen erhält. Gleiches gilt im Übrigen auch für DDM-Dateien.
- Zu Programmabstürzen kommt es auch, wenn auf die physische Datei über eine Mehrformatdatei zugriffen wird. Auch hier ist die Existenz der Mehrformatdatei zunächst kein Hinderungsgrund für die Implementierung der RCAC-Bedingungen. Erst der Zugriff über die Mehrformatdatei führt zur Fehlersituation.
- Wie zuvor beschrieben, bereiten SQL-Funktionen und SQL-Trigger keinerlei Probleme im Zusammenhang mit den RCAC-Bedingungen. Sie müssen lediglich mit der Eigenschaft **SECURED** bzw. **NOT SECURED** neu erstellt werden. READ-Trigger verhindern hingegen eine erfolgreiche RCAC-Implementierung.
- Außerdem sind keine Dateien und Abfragen erlaubt, die die Sortierfolge ICU 2.6.1 verwenden.
- Wenn Sie mit RCAC personen- oder firmenbezogene Daten schützen möchten, müssen Sie bedenken, dass die Dateien ggf. journalisiert werden. In den Journalreceivern werden die Informationen nicht maskiert!

7.3.9
Seite 50



Die Einträge in den Receivern sind nicht maskiert

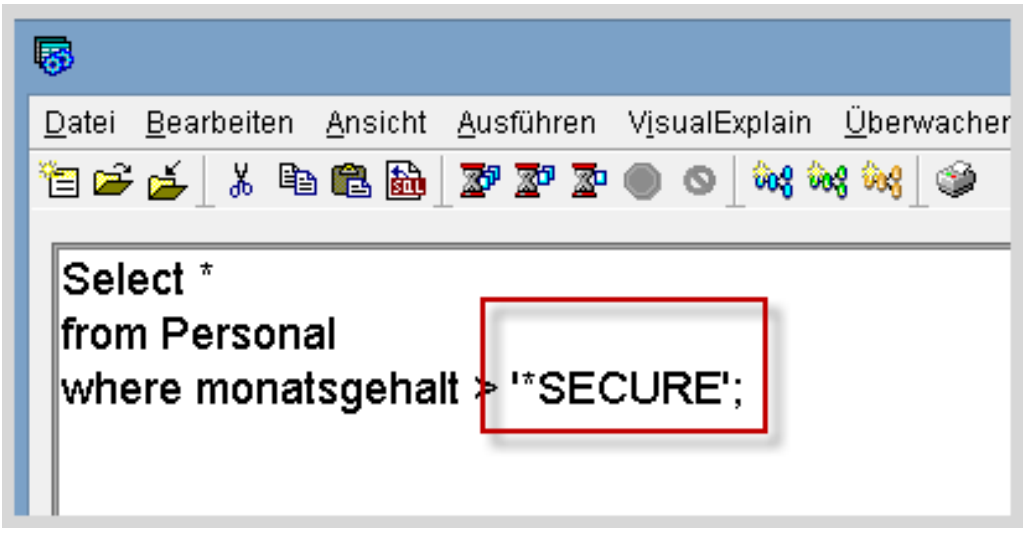
Und natürlich sind auch Konstanten, die in SQL-Abfragen verwendet werden, nicht maskiert, wenn Sie die entsprechenden Analysetools, wie Plan-Cache-Analysen oder Datenbankmonitore, einsetzen. Aus diesen Konstanten könnten demnach ebenfalls personen- und firmenbezogene Informationen von unberechtigten Personen ausgelesen werden. Sollen auch diese Konstanten „verschlüsselt" werden, können Sie mit Hilfe der Prozedur SET_COLUMN_ATTRIBUTE einigen Spalten die Eigenschaft SECURED zuweisen. Die Prozedur finden Sie in der Bibliothek SYSPROC.

```
CALL SYSPROC/SET_COLUMN_ATTRIBUTE('EPSW07', 'PERSONAL',
'MONGEHALT', 'SECURE YES');
```

Die ersten drei Argumente der Prozedur sind selbsterklärend. Im vierten Parameter können Sie die Werte ‚SECURE YES' oder ‚SECURE NO' verwenden, um das Attribut zu setzen oder zu entfernen. Ob eine Spalte dieses Attribut aufweist, können Sie im Datenbankkatalog QSYS2/SYSCOLUMNS2 überprüfen.

```
SELECT *
FROM QSYS2/SYSCOLUMNS2
WHERE SECURE = '1';
```

Schauen wir uns jetzt die Wirkung an, indem wir ein Statement aus dem Plan Cache anzeigen:



Alle Konstanten für die Spalte MONATSGEHALT sind geschützt

Fazit:

Die Implementierung der RCAC-Berechtigungen ist sicherlich einfach durchzuführen. Aber insbesondere dann, wenn Sie eine bestehende Programmumgebung haben, müssen Sie vor der Implementierung sehr genau planen, welche Auswirkungen die neuen Regeln auf

- bestehende Applikationen,
- vorhandene Sceduler-Jobs,
- vorhandene Batch-Jobs,
- Berechtigungsübernahmen,
- MQTs,
- Laufzeitverhalten und
- zukünftige Programmierung

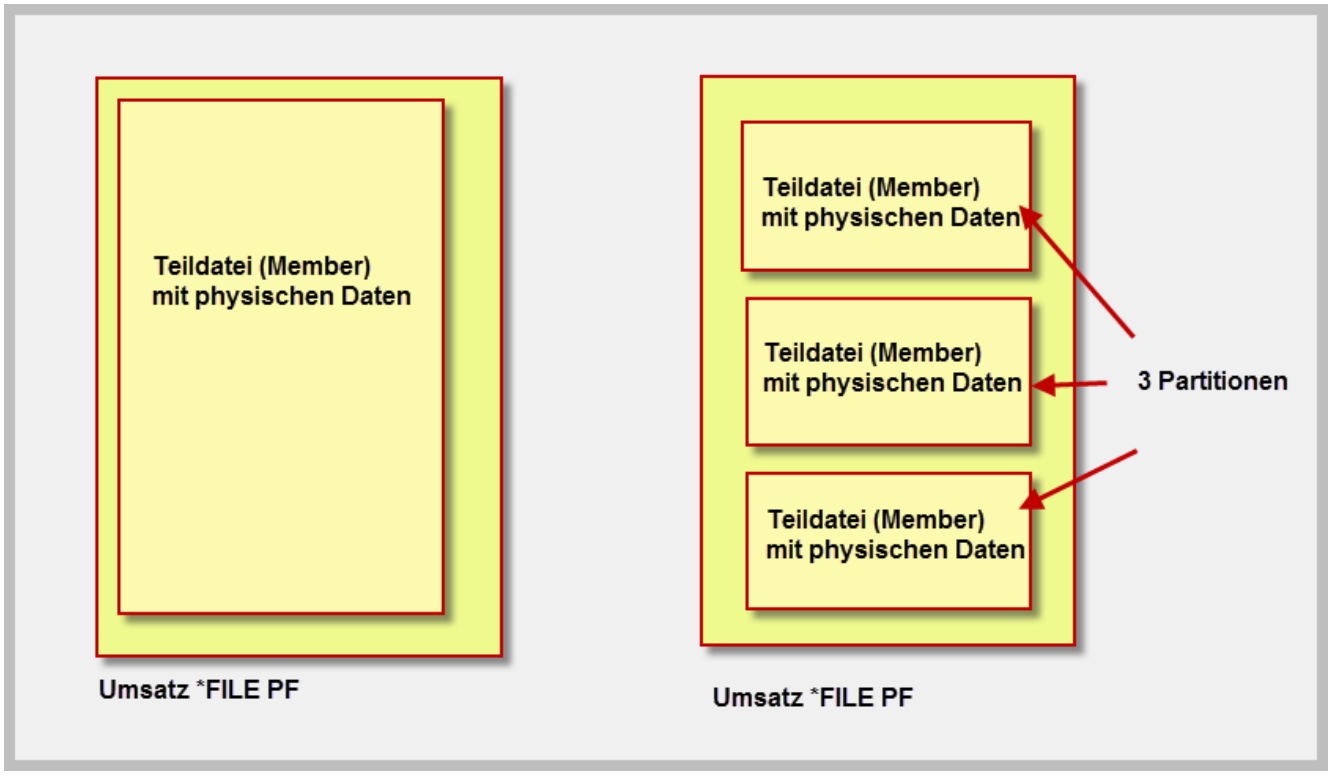
haben kann. Wahrscheinlich wird Ihnen nicht erspart bleiben, Ihre Applikationen und automatisierten Jobs zu überarbeiten, wenn Sie RCAC-Berechtigungen einführen wollen, um personen- und firmenbezogene Daten zu schützen. Aber sicherlich wird es über kurz oder lang auch erforderlich sein, personell zwischen Systemadministratoren und Datenbankadministratoren zu unterscheiden.

7.3.9
Seite 52

7.3.10 Partitionierte Tabellen

In diesem Kapitel wollen wir das Thema der partitionierten Tabellen ein wenig differenzierter betrachten. Dabei stehen die Fragen „Wann sollten partitionierte Tabellen verwendet werden?“ und „Welche Aspekte spielen für das Design eine Rolle?“ im Vordergrund.

Voraussetzung, um mit partitionierten Tabellen zu arbeiten, ist das Lizenzprogramm DB2 Multisystem (5770SS1 Option 27). Falls Sie sich jetzt fragen, was partitionierte Tabellen sind, ist die Antwort eigentlich sehr einfach:
Partitionierte Tabellen sind „moderne“ Multimember-Dateien, die wir bereits aus vergangenen Zeiten kennen. Der entscheidende Unterschied ist allerdings im Handling zu sehen. Mussten die einzelnen Member einer physischen Multimember-Datei früher mit Hilfe von logischen Dateien angesprochen werden, verhält sich die partitionierte Tabelle wie ein Objekt. Sie müssen also beim SELECT, INSERT, UPDATE oder DELETE nicht wissen, in welchem Member die Daten liegen!



Konzept partitionierter Tabellen

7.3.10
Seite 2

Allerdings müssen Sie bei partitionierten Tabellen einige Restriktionen in Kauf nehmen:

- Vor Release i7.1 waren referenzielle Integritätsbedingungen und Autoinkrement-Spalten in partitionierten Tabellen nicht erlaubt.
- Vor Release i7.2 konnte der Partitionsschlüssel nicht durch Updates manipuliert werden. Um den Partitionsschlüssel eines Satzes zu verändern, musste die Zeile gelöscht und neu eingefügt werden. Eine vollständige Liste der Restriktionen finden Sie im IBM i Knowledge Center unter folgendem Link: https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/dbmult/partrestrict.htm

Außerdem müssen Sie ggf. einige CL-Programme und Befehle anpassen und zukünftig die Option MEMBER *ALL verwenden. Dies gilt beispielsweise für Programme, die mit folgenden CL-Commands arbeiten:

- Clear Physical File Member (CLRPFM)
- Copy from Importfile – CPYFRMIMPF
- Copy to Importfile – CPYTOIMPF
- Open Query File – OPNQRYF
- Run Query – RUNQRY
- Work with Object Locks – WRKOBJLCK
- Apply Jpurnal Changes – APYJRNCHG
- Display Journal – DSPJRN
- Receive Journal Entrys – RCVJRNE
- Removed Journal Changes – RMVJRNCHG
- Retrieve Journal Entrys – RTVJRNE
- Restore Object – RSTOBJ
- Save Object – SAVOBJ
- Save restore Object – SAVRSTOBJ

und einige mehr.

7.3.10.1 Wofür werden partitionierte Tabellen benötigt?

Wann und warum werden partitionierte Tabellen wichtig? In relationalen Datenbanksystemen ist es häufig notwendig mit partitionierten Tabellen zu arbeiten, wenn für die Datenverarbeitung parallele Prozesse genutzt werden sollen. Dies ist auf einem IBM i Server allerdings nicht der Fall. Ihr Server kann parallele Verarbeitungsmethoden nutzen, egal ob die Tabellen partitioniert oder nicht partitioniert sind. Entscheidend für die Parallelverarbeitung auf einem IBM i Server ist das Feature DB2 Symmetric Multiprozessing (5770SS1 Option 26).

Wofür also benötigen wir dann partitionierte Tabellen? Für die Massendatenverarbeitung! Wenn Sie das Größenlimit einer Datei erreichen und das Datenbankdesign nicht ändern wollen oder können, kann das Problem mit Hilfe partitionierter Tabellen gelöst werden. Für nicht partitionierte Tabellen existiert ein Größenlimit für die Zeilenanzahl und für die Gesamtgröße einer Tabelle. Welche Limits existieren, zeigt die nachfolgende Tabelle.

Satzlänge (Byte)	Anzahl der Zeilen
32766	57 Millionen
8192	228 Millionen
2048	912 Millionen
1024	1,8 Milliarden

Da in einer partitionierten Tabelle bis zu 256 Member liegen können, ist es mit partitionierten Tabellen möglich 256 mal 57 Millionen Sätze à 32766 Byte zu speichern. Theoretisch können Sie also bis zu 435 TB an Daten in einer Tabelle speichern.

Effizientere Massendatenverarbeitung kann eventuell auch ein Grund für die Einführung von partitionierten Tabellen sein. Stellen Sie sich vor, es gibt eine Umsatzdatei, in der die Daten mehrerer Jahre verwaltet werden. Schnell können hier Millionen von Datensätzen entstehen und im „Online Transaction Processing (OLTP) " können sich die Antwortzeiten mit der Zeit deutlich verschlechtern. In den meisten Fällen werden Sie vermutlich auf die aktuellen Umsatzdaten des laufenden Jahres zugreifen müssen. Hier könnte es sinnvoll sein, die Umsatztabelle anhand des Auftragsdatums zu partitionieren. Sie haben hier zwei Vorteile:

- Es muss ggf. nur noch das Member mit den aktuellen Umsatzdaten gesichert werden. Die Datensicherung wird auf diese Weise schneller.
- Vielleicht wollen Sie die Umsatzdaten auch nur drei Jahre auf dem Server vorhalten und ältere Daten stets archivieren? Auch dieser Prozess ist einfach, da sie das entsprechende Member nach der Archivierung einfach löschen können. Insbesondere in Hochverfügbarkeitsumgebungen kann dieses Vorgehen Vorteile haben, denn es müssen nicht mehr tausende von Löschoperationen gespiegelt werden, sondern lediglich eine Löschanweisung für die entsprechende Partition ausgegeben werden.
- Damit aber auch Ihre Datenbankzugriffe weiterhin effizient sind, ist es bei einem solchen Szenario wichtig, dass das Auftragsdatum Primärschlüssel der Datei wird und in den SQL-Zugriffen stets das Datum als lokales Selektionskriterium verwendet wird, damit der Optimizer sich auf eine Partition beim Datenzugriff fokussieren kann..

Auch ein anderes Szenario könnte von partitionierten Tabellen profitieren. Stellen Sie sich vor, dass nachts regelmäßig neue Daten in eine Tabelle übernommen werden. Nur die neuen Daten müssen anschließend weiterverarbeitet werden. Anstatt anhand eines Timestamp-Felds die neuen Datensätze zu lokalisieren, könnten auch partitionierte Tabellen eingesetzt werden. Jede Nacht werden die Daten in einer neuen Partition übernommen und anschließend wird lediglich diese Partition weiterverarbeitet.

Prüfen Sie vorher aber stets genau, ob die Vorteile tatsächlich die Restriktionen und Nachteile aufwiegen. So wird beispielsweise die gesamte Tabelle exklusiv gesperrt, wenn ein Member gelöscht wird. Während dieser Zeit könnte sich der Zugriff auf die Tabelle deutlich verlangsamen. Bedenken Sie auch, dass Indizes, die nicht partitioniert sind, neu aufgebaut werden, wenn eine Partition der Tabelle hinzugefügt wird oder wenn Sie ein Member in der Tabelle löschen.

7.3.10.2 Wie werden partitionierte Tabellen erstellt?

Partitionierte Tabellen können Sie nur mit SQL erstellen. Im DDS gibt es dazu keine Möglichkeiten! Grundsätzlich unterscheidet man bei der Partitionierung zwei Verfahren:

a) Systemdefinierte Partitionen mittels Hash-Algorithmus
b) Benutzerdefinierte Partitionen anhand von Schlüsselwerten

```
CREATE OR REPLACE TABLE KUMULIERTER_UMSATZ FOR System NAME KUMUMS
(
ID bigint,
Jahr smallint,
Quartal smallint,
Monat smallint,
Filiale char(5),
Erloese dec(22, 2),
Kosten dec(22, 2),
Gewinn dec(22, 2)
)
PARTITION BY HASH(ID) INTO 5 PARTITIONS;
```

Bei der Hash-Partitionierung geben Sie lediglich die Anzahl der Partitionen an. Anschließend verteilt das DBMS mit Hilfe eines Hash-Algorithmus die Daten eigenständig auf die verschiedenen Partitionen.

7.3.10
Seite 6

Es ist aber auch möglich, selber Bereiche für die Partitionen vorzugeben:

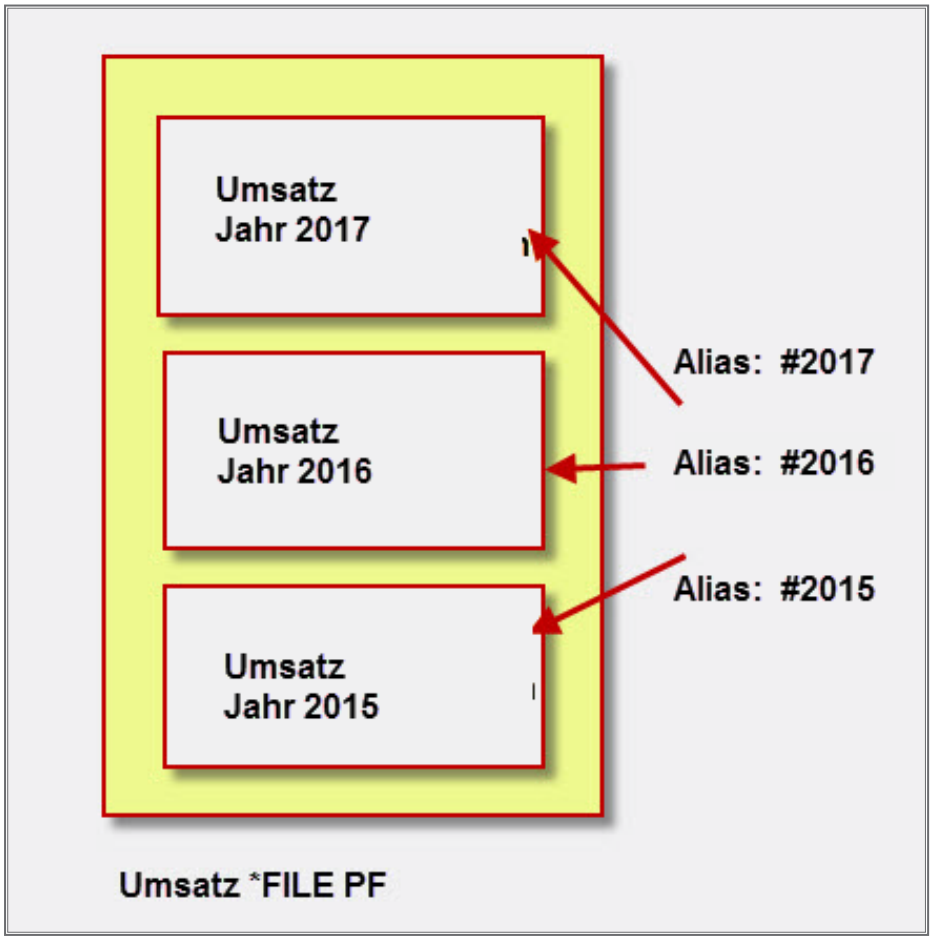
```
CREATE OR REPLACE TABLE KUMULIERTER_UMSATZ FOR System NAME KUMUMS
(ID BIGINT,
JAHR SMALLINT,
QUARTAL SMALLINT,
MONAT SMALLINT,
FILIALE CHAR(5),
ERLOESE DEC(22, 2),
KOSTEN DEC(22, 2),
GEWINN DEC(22, 2)
)
PARTITION BY RANGE(JAHR NULLS FIRST)
(STARTING FROM (MINVALUE) ENDING AT (2013) INCLUSIVE,
STARTING FROM (2014) ENDING AT (2014) INCLUSIVE,
STARTING FROM (2015) ENDING AT (2015) INCLUSIVE,
STARTING FROM (2016) ENDING AT (2016) INCLUSIVE,
STARTING FROM (2017) ENDING AT (2017) INCLUSIVE,
STARTING FROM (2018) ENDING AT (2018) INCLUSIVE,
STARTING FROM (2019) ENDING AT (MAXVALUE) INCLUSIVE
);
```

In diesem Fall werden die Daten anhand eines „Schlüsselwerts“ auf die einzelnen Partitionen verteilt. Im Beispiel ist dies die Spalte JAHR. Der Partitionsschlüssel muss nicht explizit Schlüssel der Tabelle sein, aber Sie müssen darauf achten, dass die Daten den definierten Bereichen entsprechen, anderenfalls erhalten Sie eine Fehlermeldung:

```
SQL-Status: 22525
Vendorencode: -327
Nachricht: [SQL0327] Partitionierungsschlüssel stimmt nicht mit einer definierten Partition
überein. Ursache . . . . : Es wurde versucht, eine Anweisung INSERT, UPDATE, CREATE
TABLE oder ALTER TABLE für eine partitionierte Tabelle zu verarbeiten, aber durch die für
den Partitionierungsschlüssel angegebenen Werte wurde die Zeile nicht für Partitionierung
qualifiziert. Fehlerbeseitigung:  Trat der Fehler bei einer Anweisung INSERT oder UPDATE
auf, die für die einzufügende oder zu aktualisierende Zeile angegebenen Werte so ändern,
dass sie einer Partition der Tabelle entsprechen, oder die Partitionsdefinition ändern, damit
dieser Schlüsselwert zulässig ist. Trat der Fehler bei einer Anweisung CREATE TABLE oder
ALTER TABLE auf, die Partitionsdefinitionen so ändern, dass alle vorhandenen Zeilen in die
Tabelle passen, oder die Zeilen löschen, die nicht passen. Die Anforderung wiederholen.
```

Datensätze entsprechen nicht dem Partitionierungsschlüssel

In diesem Umfeld spielen ALIAS-Namen eine gewisse Rolle, mittels derer Programmierer eine Partition mit SQL direkt ansprechen können. Schauen Sie sich die folgende Abbildung an:



Die partitionierte Tabelle KUMULIERTER_UMSATZ

Mit Hilfe eines ALIAS-Namens können wir gezielt die einzelnen Partitionen ansprechen:

```
CREATE OR REPLACE ALIAS #2017 FOR KUMULIERTER_UMSATZ
(PART000001);
CREATE OR REPLACE ALIAS #2016 FOR KUMULIERTER_UMSATZ
(PART000002);
CREATE OR REPLACE ALIAS #2015 FOR KUMULIERTER_UMSATZ
(PART000003);


SELECT ….
FROM #2017;
```

7.3.10
Seite 8

JAHR	QUA...	MO...	FILIA...	ERLOESE	KOSTEN	GEWINN
2017	2	0	00007	113451432.62	108409370.56	5046549.06
2017	2	6	00007	38333789.22	36772802.08	1565474.14
2017	3	7	00007	32540439.62	30831461.80	1713464.82
2017	3	8	00007	23563916.98	22027451.92	1540952.06
2017	3	0	00007	80195945.54	75284200.24	4916232.30
2017	3	9	00007	23935305.06	22279695.64	1660096.42
2017	4	10	00007	24132742.52	23055069.76	1082159.76
2017	4	11	00007	24922323.06	23438170.60	1488639.46
2017	0	0	00007	378337712.34	358881218.80	19460980.54

Zugriff mit ALIAS-Namen

Anmerkung:

Sie erzeugen mit dem Befehl CREATE ALIAS ein Objekt der Art *FILE DDMF

Im RPG-Programm haben Sie verschiedene Möglichkeiten, ein spezifisches Member einer partitionierten Tabelle anzusprechen:

1. Sie nutzen eine logische Datei, die für ein Member der Tabelle erzeugt wurde.
2. Sie könnten auch den Alias-Namen verwenden. Allerdings muss der Alias-Name in diesem Fall zwingend auf den Systemnamen (Kurzname) der Tabelle verweisen.
3. Seit einiger Zeit ist es im RPG-Programm möglich, mit den Schlüsselwörtern EXTFILE(XXXX) und EXTMBR(PART00001) ein Member direkt anzusprechen. Sollen mehrere Member im Programm verarbeitet werden, können Sie Variablen für die Definition verwenden. Sie müssen in diesen Fällen lediglich bedenken, dass Sie das Member schließen, bevor die nächste Teildatei geöffnet werden kann.
4. Natürlich können Sie auch den Befehl OVRDBF verwenden.

```
OVRDBF FILE(KUMUMS) TOFILE(UMSATZ) MEMBER(PART000001)
CALL RPG_Programm
```

Falls es erforderlich ist, mittels RPG die gesamte Datei zu verarbeiten, brauchen Sie eine logische DDS-Datei oder einen nicht-partitionierten SQL-Index.

Bedenken Sie bei der Verarbeitung bitte, dass jede Partition eigene relative Satznummern hat, das heißt, die Partition 1 und die Partition 2 haben beide die relative Satznummer (RSN) 1. Eine Verarbeitung der Daten anhand der RSN ist somit schwierig.

7.3.10.3 Der Migrationsprozess

Sicherlich wäre es wünschenswert, partitionierte Tabellen bereits beim Datenmodell zu berücksichtigen. Doch die Realität ist häufig eine andere. In der Regel sollen bestehende Tabellen nachträglich partitioniert werden. Die Konvertierung ist einfach: Sie können den Befehl ALTER TABLE verwenden. Hierbei wird eine neue partitionierte Tabelle erstellt, die Daten werden in die Tabelle übernommen und alle bestehenden Indizes werden als partitionierte Indizes neu erstellt. Dieser Vorgang kann sehr zeitintensiv sein und die Datenübernahme erfolgt auch nicht parallel. Aus diesem Grund könnte es sinnvoll sein, dass die Datenübernahme manuell ausgeführt wird. Sinnvollerweise sollten hierbei ALIAS-Namen verwendet werden:

```
INSERT INTO #2017          --Alias-Name
Select *
From Umsatz_kumuliert_nicht_Partitioniert
Where jahr = '2017';


INSERT INTO #2016          --Alias-Name
Select *
From Umsatz_kumuliert_nicht_Partitioniert
Where jahr = '2016';


INSERT INTO #2015          Alias-Name
Select *
From Umsatz_kumuliert_nicht_Partitioniert
Where jahr = '2015';
```

7.3.10
Seite 10

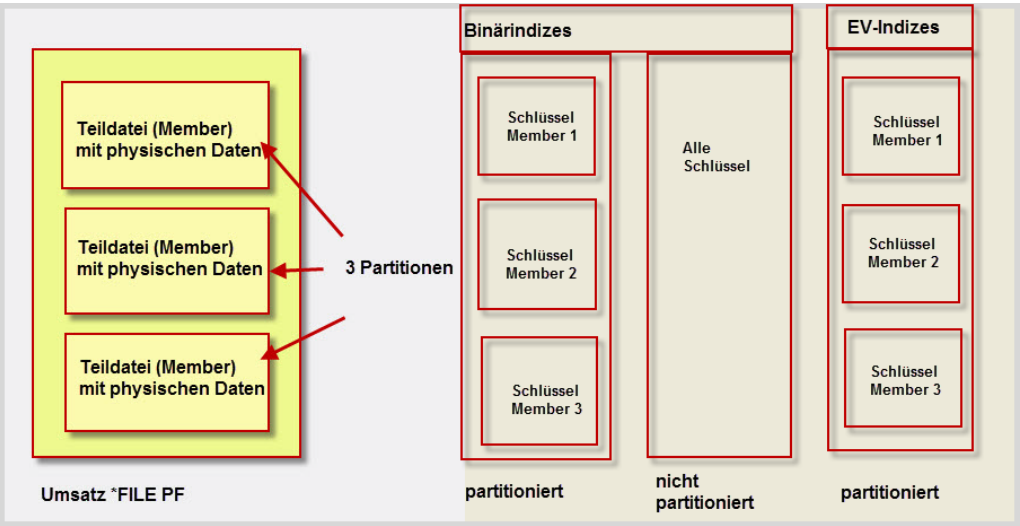
Wie gehen Sie bei der Migration vor?

1. Stellen Sie sicher, dass Sie genügend Plattenspeicher haben, um die partitionierte und die nicht partitionierte Tabelle zu speichern.
2. Vor der Migration sollten Sie die nicht partitionierte Tabelle und deren Indizes sichern.
3. Erstellen Sie anschließend die partitionierte Tabelle.
4. Ermitteln Sie vor der Migration die Anzahl Zeilen, die jede Partition erhalten soll.
5. Erstellen Sie die ALIAS-Namen für den Migrationsprozess.
6. Nutzen Sie den Befehl OVRDBF, um die Blockgröße für die Quell- und die Zieldatei auf 256 KB zu erhöhen.
7. Übernehmen Sie die Daten. Nutzen Sie hierfür konkurrierende, parallel laufende INSERT-Anweisungen mit SUBSELECT wie gezeigt.
8. Prüfen Sie nach der Datenübernahme die Anzahl Zeilen je Partition.
9. Erstellen Sie jetzt die notwendigen Indizes.
10. Löschen Sie abschließend die nicht partitionierte Tabelle und deren Indizes.
11. Falls erforderlich, müssen sie abschließend die neue Tabelle und deren Indizes umbenennen.

7.3.10.4 Indizes und partitionierte Tabellen

Für partitionierte Tabellen können sowohl Binärindizes als auch Encodded Vector Indizes erstellt werden. Zu beachten ist, dass EVIs immer als partitionierte Indizes erstellt werden. Bei den Binärindizes haben Sie die Wahl: Der Binärindex kann partitioniert werden, muss es aber nicht. Falls Sie den Index nicht partitionieren, umspannt der Index die gesamte Datei und wird aus diesem Grund auch „SPANNING INDEX“ genannt.

```
-- Spanning Index
CREATE INDEX KUMULIERTER_UMSATZ_I1_SPANNING
ON KUMULIERTER_UMSATZ(FILIALE) NOT PARTITIONED;

CREATE INDEX KUMULIERTER_UMSATZ_I1_PARTITIONED
ON KUMULIERTER_UMSATZ(FILIALE) PARTITIONED;

CREATE ENCODED VECTOR INDEX KUMULIERTER_UMSATZ_E1_PARTITIONED
ON KUMULIERTER_UMSATZ(FILIALE) INCLUDE (SUM(ERLOESE), SUM(KOSTEN));
```



Indizes für partitionierte Tabellen

7.3.10
Seite 12

Was müssen Sie bei Primär- und UNIQUE-Keys beachten?
Ein eindeutiger Index kann nur partitioniert werden, wenn die Schlüssel identisch oder eine Teilmenge des Partitionierungsschlüssels sind.

Wenn ein eindeutiger Index eine Teilmenge des Partitionsschlüssels ist, wird dieser Index standardmäßig als partitionierter Index implementiert. Nur wenn in der Indexdefinition explizit die Klausel NON PARTITIONED verwendet wird, wird ein Spanning-Index erstellt.

Bedenken müssen Sie zudem, dass ein nicht partitionierter Index die gesamte Struktur der physischen Tabelle enthält. Wenn zu einem späteren Zeitpunkt in der Tabelle eine Partition hinzugefügt oder gelöscht wird, muss der nicht partitionierte Index neu erstellt werden; und da partitionierte Tabellen in der Regel keine kleinen Tabellen sind, kann dieser Prozess sehr viel Zeit beanspruchen und entsprechend viele Systemressourcen binden. Nur wenn die Partitionen an das Ende der Tabelle angehängt oder am Ende der Tabelle gelöscht werden, kann die Neuerstellung der Indizes verhindert werden.

7.3.10.5 Partitionierte Tabellen und der SQL-Optimizer

Die Optimierung der SQL-Statements für partitionierte Tabellen ist vergleichbar mit der Optimierung eines UNION ALL-Statements.

Die Abfrage:

```
SELECT *
FROM KUMULIERTER_UMSATZ
WHERE FILIALE = '00011';
```

entspricht im Wesentlichen der Abfrage:

```
SELECT *
FROM   1. member
WHERE FILIALE = '00011'

UNION ALL

SELECT *
FROM   2. member
WHERE FILIALE = '00011'

UNION ALL

SELECT *
FROM   3. member
WHERE FILIALE = '00011'
```

Grundsätzlich gilt, dass der Optimierungsprozess und die Ausführungspläne mit partitionierten Tabellen aufwendiger werden, denn nur wenn die lokalen Selektionskriterien der SQL-Abfrage den Partitionierungsschlüssel enthalten, kann dieser bei der Datenselektion vom Optimizer berücksichtigt werden.

Hinzu kommt, dass die Daten der partitionierten Tabellen für viele Abfragen „materialisiert“ werden müssen. Was heißt das konkret? Stellen Sie sich vor, dass bezogen auf die letzten zwei Jahre die Filiale mit den höchsten Erlösen ermittelt werden soll:

```
Select filiale, sum(Erloese) as Erloese
from Kumulierter_UMSATZ
where jahr = year(current date) - 1 or jahr = year(current date)
group by filiale
order by erloese desc
fetch first 3 rows only;
```

Um das Ergebnis zu liefern, werden die Daten in eine Hash-Tabelle gestellt und dann erst kumuliert. Die Datenselektion erfolgt trotz vorhandenem Index mittels Table Scan, das heißt, die gesamte Tabelle wird gelesen.

Der Ausführungsplan

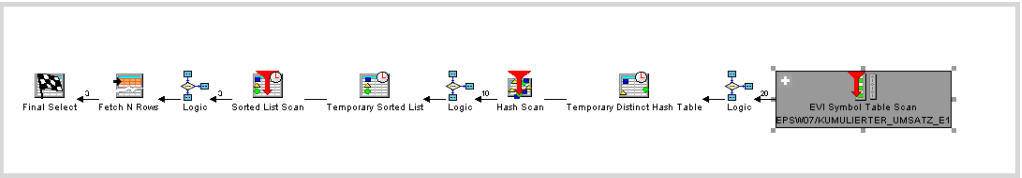
Diese temporären Objekte beanspruchen Arbeitsspeicher und temporären Speicher. Ohne eingehende Analyse der Laufzeitumgebung ist es schwer vorherzusagen, ob und wie viel mehr Ressourcen genutzt werden und ob sich die Antwortzeiten verschlechtern.

Bedenken Sie daher:
Partitionierte Tabellen erlauben zwar die Definition von 256 Membern, aber es versteht sich hoffentlich von selbst, dass man mit Augenmaß und Verantwortung die Anzahl der tatsächlich benötigten Member festlegt. Als Faustregel gilt, dass Sie – wenn möglich – nicht mehr als 36 Member verwenden.

Und partitionierte Tabellen benötigen natürlich Indizes. Erstellen wir zum Beispiel einen EVI mit inkludierten Summenspalten:

```
CREATE ENCODED VECTOR INDEX KUMULIERTER_UMSATZ_E1
FOR SYSTEM NAME KUMUMS01
ON KUMULIERTER_UMSATZ ( Jahr, FILIALE ASC )
       INCLUDE ( SUM ( ERLOESE ) , SUM ( KOSTEN ) ) ;
```

Dann kann die oben verwendete Abfrage deutlich effizienter ausgeführt werden:



Der Ausführungsplan mit EVI

Grundsätzlich sollten Sie die Indizes für eine partitionierte Tabelle ebenfalls partitionieren. Wie bereits erwähnt, sind diese Indizes deutlich pflegeleichter. Falls Sie jedoch native Zugriffe aus den HLL-Programmen realisieren müssen oder Sie wissen, dass in Ihrem Unternehmen SQL-Abfragen auch noch von der CQE ausgeführt werden, dann benötigen Sie auf alle Fälle zusätzlich auch nicht partitionierte Indizes und müssen den erhöhten Mehraufwand für die Indexpflege in Kauf nehmen. Sie haben gelesen, dass EV-Indizes immer partitioniert sind? Auch hier sollten Sie vorsorgen, wenn die CQE noch aktiv ist oder native Satzzugriffe erfolgen: In diesem Fall erstellen Sie nicht nur den EVI, sondern zusätzlich auch einen nicht partitionierten Binärindex.

Und wie sieht es mit den Statistikdaten aus? Statistiken werden für jede Partition separat erhoben. Achten Sie darauf, dass hier tatsächlich alle Statistiken für jede Partition vorhanden sind.

7.3.10
Seite 16

7.4 Grafische Datenbank-Tools

Um mit unserer Datenbank zu arbeiten, könnten wir auch weiterhin die 5250-Oberfläche nutzen. Nur: IBM entwickelt diese Oberflächen nicht mehr weiter. Stattdessen gewinnen neue grafische Oberflächen zunehmend an Bedeutung. Insbesondere im Bereich SQL und Datenbanken wird es später auch darum gehen, die Datenbank zu analysieren, die Performance einzelner SQL-Statements zu prüfen und gegebenenfalls zu optimieren. Es wird darum gehen, festzustellen, ob Indizes vorhanden sind und warum diese nicht genutzt werden. All diese Dinge sind auf der 5250-Oberfläche gar nicht oder nur sehr umständlich über APIs zu realisieren. Also warum sollten wir nicht von Beginn an mit einer modernen und komfortablen Oberfläche arbeiten? Und noch eine Anmerkung: Viele Administratoren und Entwickler glauben, dass die Optimierung von SQL überflüssig sei. Schließlich haben wir auch früher die AS/400 beziehungsweise den iSeries Server kaum optimiert – und irgendwie lief vieles von allein. Im Bereich SQL ist dies nicht der Fall. Hier wird irgendwann der Tag kommen, an dem Sie sich mit Detailfragen auseinandersetzen müssen.

Ergänzung 2/2007

7.4 Seite 2

7.4.1 iSeries Navigator

Mit Erscheinen der Client Access-Version 3.2 kam das Produkt iSeries Navigator auf den Windows Desktop. Der iSeries Navigator bietet einige gute Tools, die dem Benutzer auf attraktive Weise Datenbankinformationen liefern.

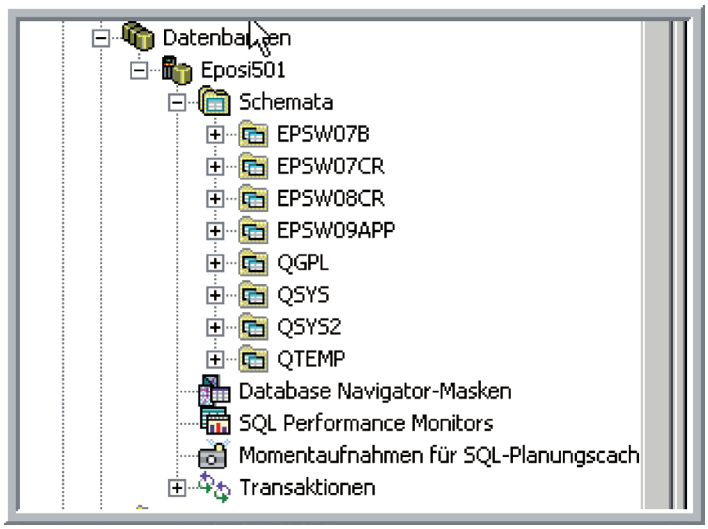
Seit langem ist die SQL Engine zwar im i5/OS integriert, um aber Ad-hoc-Abfragen mit SQL zu erstellen, musste man früher zusätzliche Lizenzen erwerben und Produkte installieren. Seit Release V4R4 hat IBM ein SQL-Scripting-Tool in den iSeries Navigator integriert, das in den folgenden Releases ständig verbessert wurde. Da die Standard-Features des iSeries Navigators dem Kunden kostenfrei zur Verfügung stehen, läutete dieses neue Feature eine neue Ära der SQL-Entwicklung auf der iSeries ein. Zum ersten Mal konnten Benutzer interaktives SQL nutzen, ohne zusätzliche Produkte kaufen zu müssen. Das Tool ist nicht nur eine kostenfreie Alternative zum STRSQL-Tool, sondern bietet Entwicklern, die keine Erfahrungen mit der iSeries haben, zudem ein komfortables Umfeld, die 5250-Oberfläche zu meiden.

iSeries Navigator ist eine leistungsstarke grafische Oberfläche, mit der Sie ohne großen Aufwand mit dem iSeries-Server arbeiten können und problemlos Informationen über Ihre bestehenden Tabellen erhalten, aber auch neue Datenbanken erstellen können. Außerdem können Sie Ad-hoc-Abfragen schreiben. Des Weiteren bietet die Oberfläche verschiedene Möglichkeiten, die Performance Ihrer Datenbankzugriffe zu optimieren. Mit jedem neuen Release kommen weitere Komponenten hinzu, so dass die grafische Oberfläche ständig erweitert und verbessert wird.

7.4.1
Seite 2

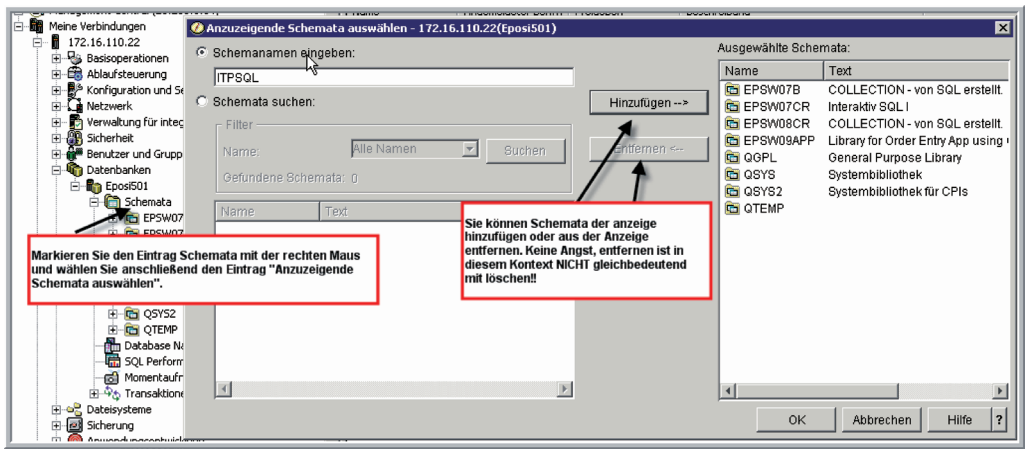
7.4.1.1 Mit Datenbanken arbeiten

Um mit Datenbanken zu arbeiten, stellen Sie zunächst im iSeries Navigator eine Verbindung zum System her. Anschließend erweitern Sie den Eintrag „Datenbanken“.



Mit Datenbanken arbeiten

Sie finden eine Reihe „Schemata“ in der Anzeige. Bitte verwechseln Sie diese Anzeige nicht mit der Bibliotheksliste. Die hier eingeblendeten Schemata haben damit nichts zu tun! Sie können die anzuzeigenden Schemata individuell bestimmen, wenn Sie den Eintrag „Schemata“ mit der rechten Maustaste markieren und anschließend „Anzuzeigende Schemata auswählen“ aufrufen. Daraufhin erscheint der folgende Dialog:



Anzuzeigende Schemata auswählen

Ich habe meine anzuzeigenden Schemata ein klein wenig verändert und mein Schema ITPSQL in die Anzeige aufgenommen. Das Schema ITPSQL hatte ich bereits im Kapitel 7.3.2.2 erstellt. Dort finden Sie auch entsprechend detaillierte Informationen.

7.4.1.2 Mit Datenbanktabellen arbeiten

Eine Tabelle ist das grundlegende Datenbankobjekt, in dem Datenbankinformationen gespeichert werden. Mit dem iSeries Navigator können Sie viele der üblichen Verwaltungsvorgänge für Tabellen ausführen. Die meisten Operationen basieren dabei auf Structured Query Language (SQL). Sie müssen SQL jedoch nicht vollständig verstehen, um die Operationen auszuführen. Häufig bietet die grafische Oberfläche entsprechende Funktionalität.

7.4.1.2.1 Datenbanktabellen über das grafische Interface erstellen

Um Tabellen mit dem grafischen Interface des Navigators zu erstellen, müssen Sie den Eintrag „Tabellen“ mit der rechten Maustaste markieren und anschließend „Neu“ wählen.

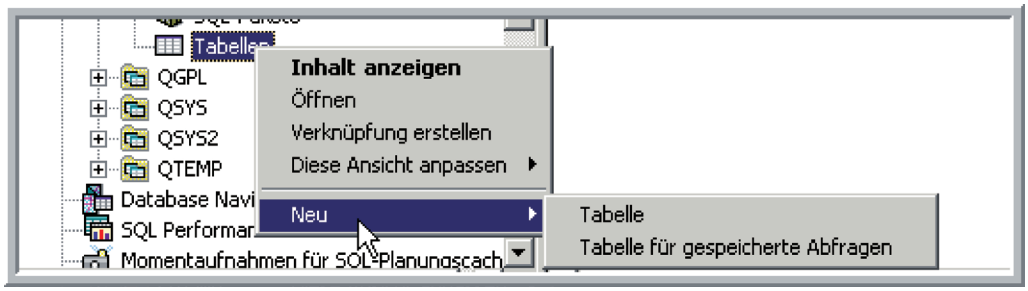
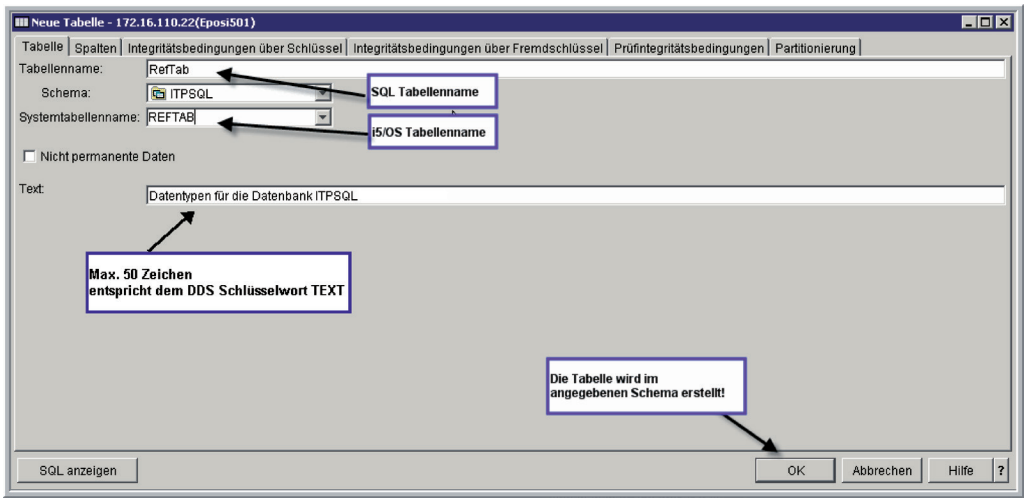


Tabelle mit grafischem Interface erstellen

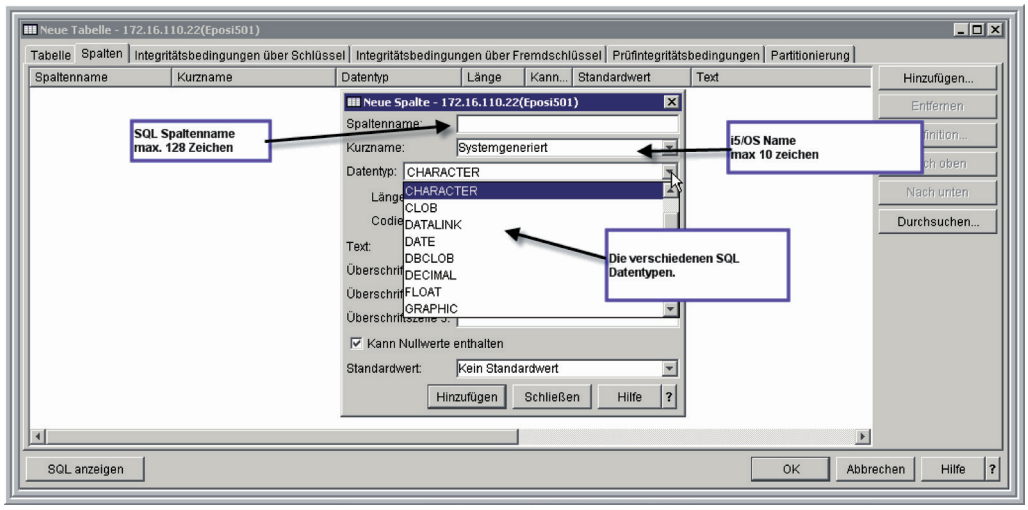
In der Abbildung sehen Sie den Eintrag „Tabelle“; wählen Sie diesen aus. Daraufhin öffnet sich folgender Dialog:

7.4.1
Seite 4



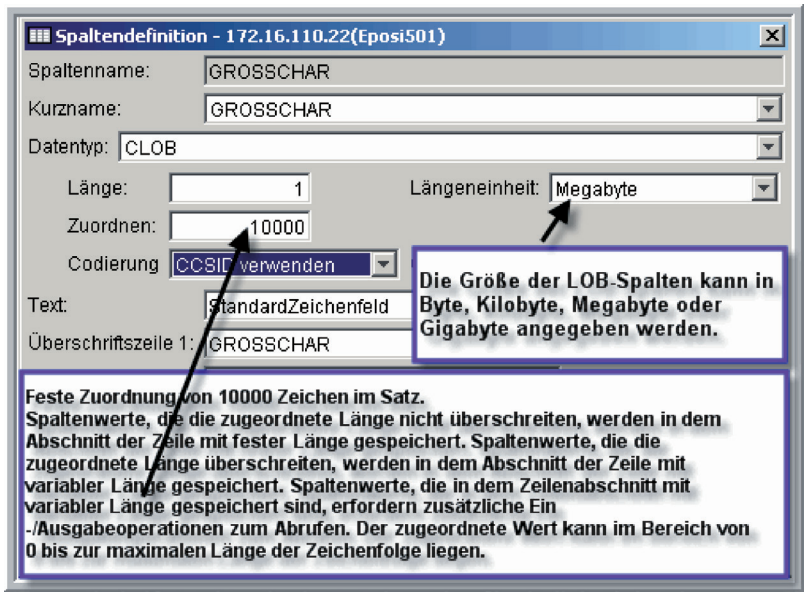
Tabellenname festlegen

Im ersten Schritt bestimmen Sie den Tabellennamen. Achten Sie darauf, dass es hier einen SQL-Tabellennamen und einen i5/OS-Namen gibt. Dies hängt mit den unterschiedlichen Namenskonventionen zusammen. Nähere Informationen entnehmen Sie bitte dem Kapitel 7.3.3.1. Sobald der Tabellenname festgelegt ist, können Sie Ihrer Tabelle neue Spalten hinzufügen. Dafür wählen Sie das Register „Spalten“. Daraufhin erscheint ein weiteres Dialogfenster:



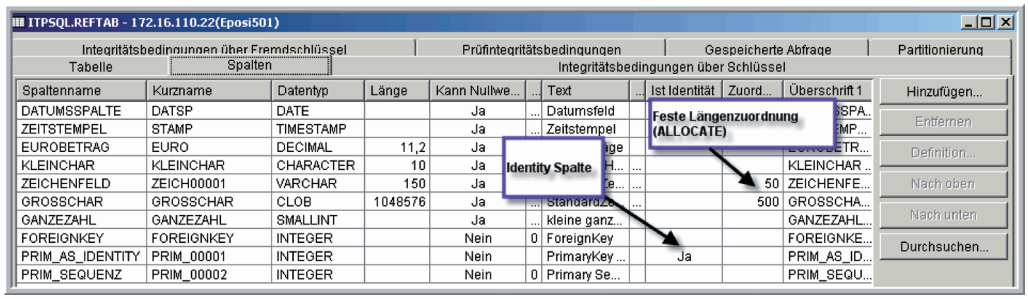
Spalten einer Tabelle hinzufügen

Ich füge jetzt meiner Referenzdatei die einzelnen Spalten hinzu. Beispielhaft sehen Sie die Definition einer CLOB-Spalte:



Definition einer CLOB-Spalte

Vieles in diesem Dialog ist selbsterklärend und muss daher nicht detailliert erläutert werden. Außerdem verfügt der iSeries Navigator im Zweifelsfall immer über eine entsprechende Hilfefunktion, die Ihnen die einzelnen Eingaben erläutert. Auf eines sollten Sie allerdings achten: Verwenden Sie für die Spaltennamen keine Umlaute oder sonstigen Sonderzeichen. SQL erlaubt es Ihnen zwar, allerdings ist die Handhabung solcher Spaltennamen später sehr umständlich. Für meine Referenztabelle erhalte ich abschließend folgendes Bild:



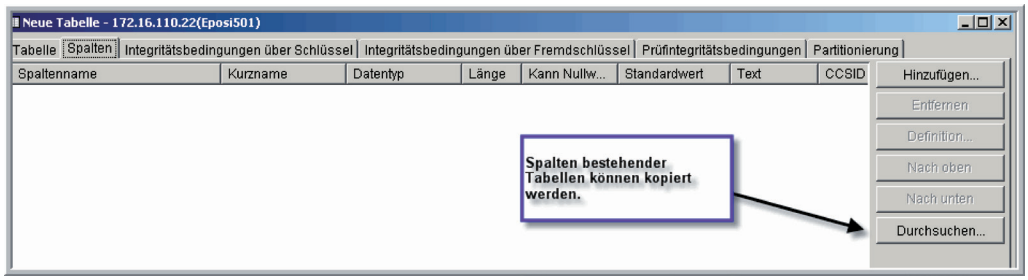
Referenztabelle

Wenn Sie jetzt auf „OK“ drücken, dann wird die Tabelle erstellt.

Einige werden sich jetzt sicherlich wundern. Was wollen wir mit einer Referenztabelle? Das kommt doch aus DDS-Zeiten! SQL kennt keine Referenztabellen. Sicher – das stimmt, dennoch kann ich gleich über das grafische Interface auf die Referenztabelle zugreifen. Außerdem werde ich in einigen Monaten nicht mehr wissen, welche Datentypen im physischen Entwurf festgelegt wurden. Die Referenztabelle dient mir somit zusätzlich zur Dokumentation. Anhand dieser Tabelle kann ich problemlos alle Datentypen identifizieren, die ich in den Tabellen des Schemata ITPSQL nutzen will.

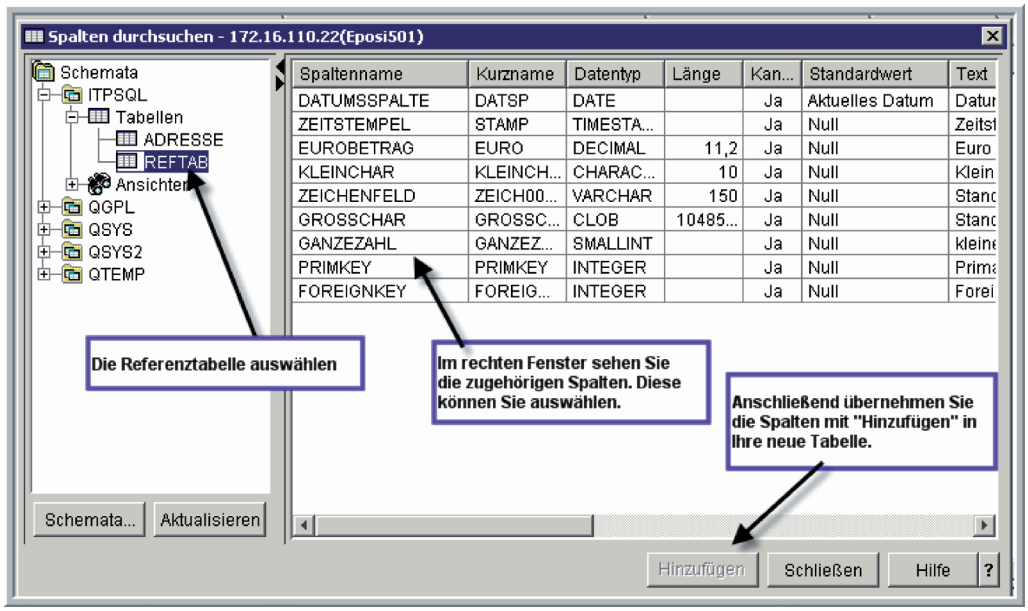
7.4.1
Seite 6

Zunächst erstelle ich aber mit Hilfe meiner „Referenztabelle“ einige weitere Tabellen. Ich rufe also wieder den Dialog „Neue Tabelle erstellen“ auf. Allerdings füge ich jetzt nicht selbst die Spalten hinzu, sondern ich verwende dieses Mal den Button „Durchsuchen…“.



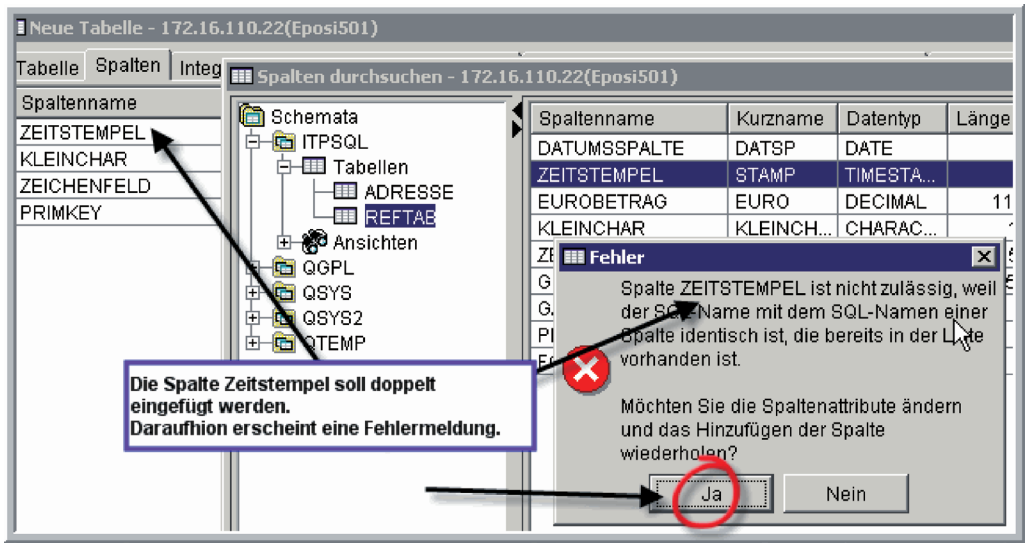
Bestehende Spalten suchen

Daraufhin erscheint folgender Dialog:



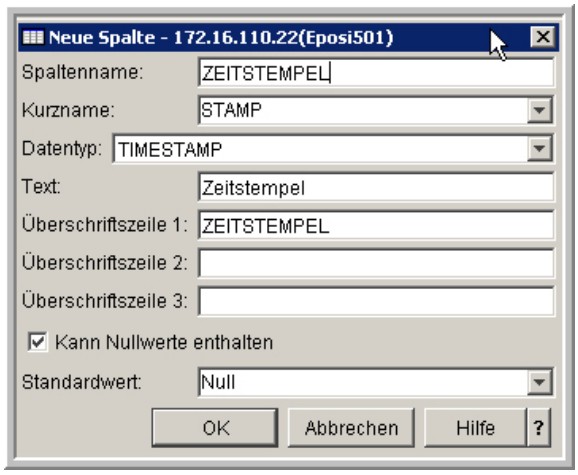
Spalten in bestehenden Tabellen wieder verwenden

Es ist sogar möglich, mehrere Spalten zu markieren und gleichzeitig zu übernehmen. Allerdings erhalten Sie eine Fehlermeldung, sobald Sie versuchen, eine Spalte mit identischem Namen mehrfach einzufügen. Schauen Sie, was passiert:



Doppelte Spaltennamen übernehmen

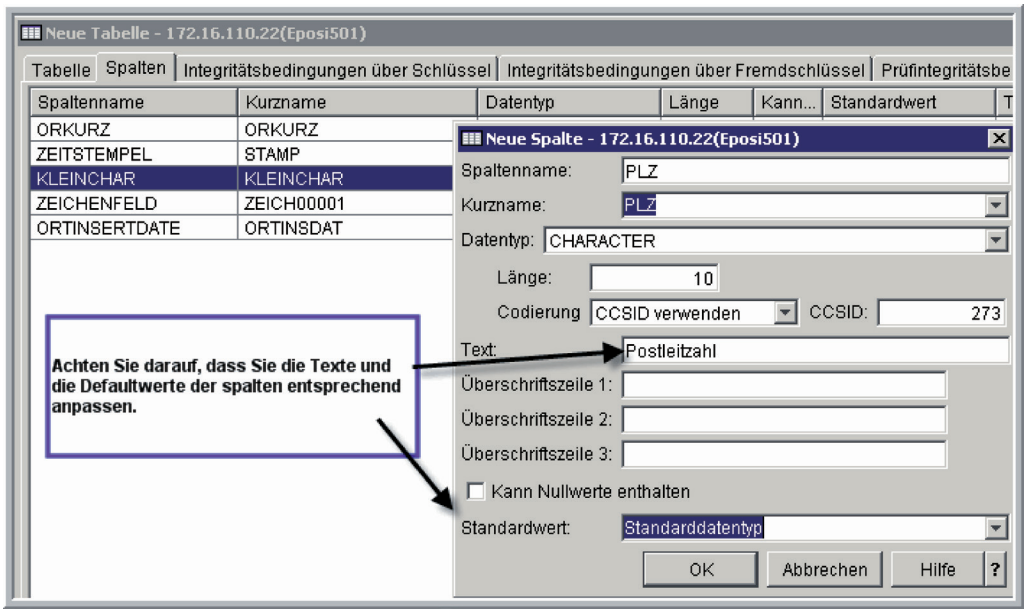
Die Fehlermeldung macht darauf aufmerksam, dass eine Spalte mit diesem SQL-Spaltennamen bereits existiert. Wenn Sie den Dialog mit „Ja“ bestätigen, erhalten Sie die Möglichkeit, die Attribute direkt zu ändern. Im folgenden Fenster können die Spaltenattribute geändert werden.



Attribute der Spalten ändern

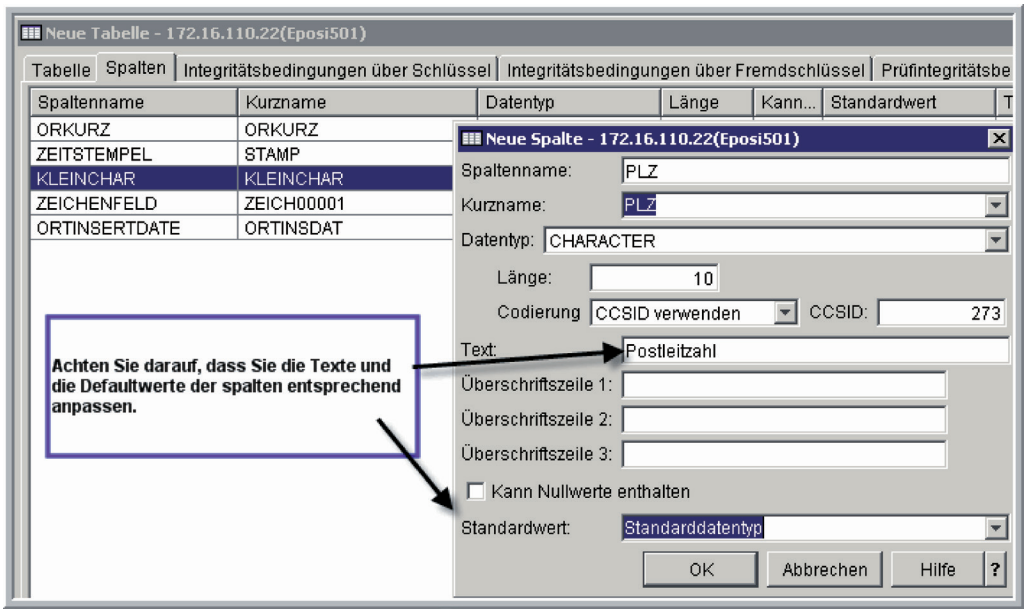
Aber auch die Attribute der Spalten, die ohne Fehlermeldung übernommen wurden, müssen nachträglich entsprechend angepasst werden. Markieren Sie einfach die zu ändernde Spalte und klicken Sie anschließend auf „Definition“:

7.4.1
Seite 8



Spaltenattribute neuer Tabellen nachträglich ändern

Sie gelangen wieder in den bereits gezeigten Dialog und können die Spaltenattribute beliebig ändern.



Attribute der Spalte „KleinChar“ ändern

Zunächst passe ich gemäß dem logischen Konzept die Spaltennamen an. Der SQL-Spaltenname kann nur verändert werden, solange die Tabelle nicht erstellt wurde. Anschließend ist die Änderung nur möglich, indem Sie die entsprechende Spalte entfernen und neu erstellen. Das ist eine Möglichkeit, die relativ einfach mit leeren Datenbanktabellen funktioniert. Sobald die Tabelle Daten enthält, ist etwas mehr Arbeit erforderlich, andernfalls verlieren Sie Daten. Achten Sie auch auf die Default-Werte Ihrer Spalten. Normalerweise enthalten leere

Spalten den Wert NULL. Wenn Sie dies unterbinden wollen, müssen Sie die Markierung im Eintrag „kann Nullwerte enthalten“ entfernen und stattdessen den gewünschten Standarddatentypen auswählen. In meinem Beispiel ist die Spalte „PLZ“ mit Leerzeichen gefüllt, wenn keine Werte erfasst wurden. Geschickter wäre es natürlich, die Referenzfelder bereits entsprechend zu definieren.

Ausgehend vom Basisfenster in Abbildung 11 stehen weitere Möglichkeiten für die Bearbeitung zur Verfügung:

- Wenn Sie eine Spalte entfernen möchten, wählen Sie die Spalte aus und klicken auf „Entfernen“.
- Sie können die Spaltenreihenfolge ändern, wenn Sie eine Spalte auswählen und auf „Nach oben“ oder „Nach unten“ klicken.

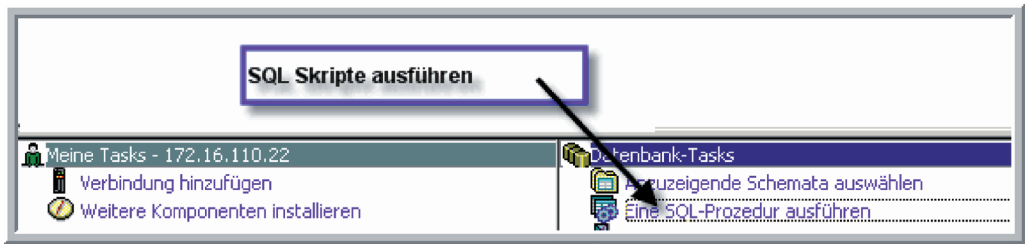
Sie sehen, dass das grafische Interface keine SQL-Syntax erfordert, sondern auf einfache Art und Weise die Erstellung von Tabellen ermöglicht. Dieses Interface ist auch für alle DDS-Dateien nutzbar. Außerdem ist es unwichtig für diese Arbeit, ob Sie eine mit CRTLIB erstellte Bibliothek nutzen oder ob Ihre Tabellen in einem SQL-Schema erstellt werden. Der iSeries Navigator unterscheidet diesbezüglich nicht. Probieren Sie es aus!

7.4.1.2.2 SQL Prozedurumgebung iSeries Navigator

Falls Sie keine Lust haben, alle Tabellen mit der grafischen Oberfläche zu erstellen, stellt Ihnen der iSeries Navigator eine separate Prozedurumgebung zur Verfügung. In dieser Skriptumgebung gibt es sogar für die Data-Manipulation-Language- (DML-) Befehle – SELECT, INSERT, UPDATE und DELETE – eine detaillierte Bedienerführung (einen SQL-Assistenten). Für komplexere Abfragen – wie Select-Statements mit Common-Table-Expressions (CTE), oder Sub-Selects oder Update-Statements mit integriertem Select-Statement – versagt diese Bedienerführung leider bislang ebenso wie die Bedienerführung unter STRSQL. Dazu später aber mehr. Grundsätzlich kann jedes SQL-Statement hier ausgeführt werden. Darüber hinaus erhalten Sie Möglichkeiten für die Performanceanalyse. Und – falls Sie in die SQL-Programmierung einsteigen – es steht Ihnen auch noch ein Debugger zur Verfügung.

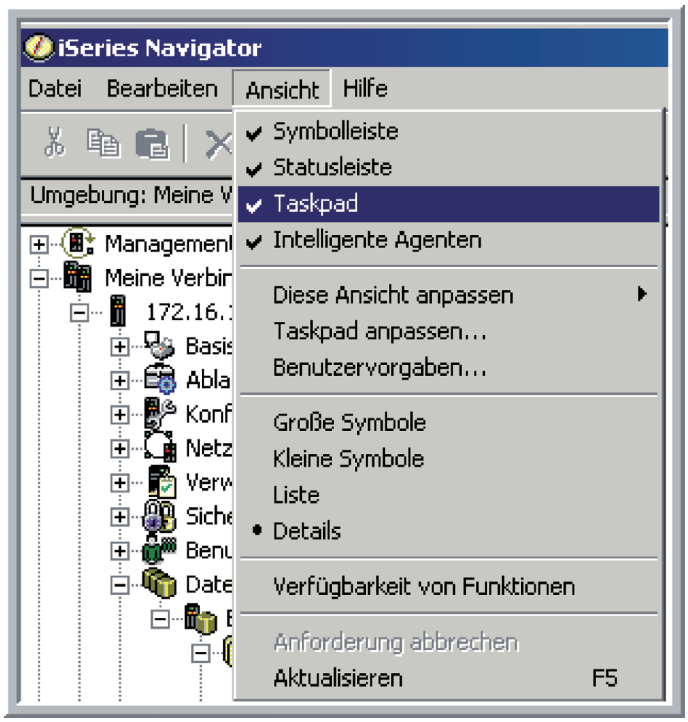
7.4.1.2.2.1 SQL-Prozedurumgebung aufrufen

Als Oberfläche bietet sich hier ebenfalls der iSeries Navigator an:



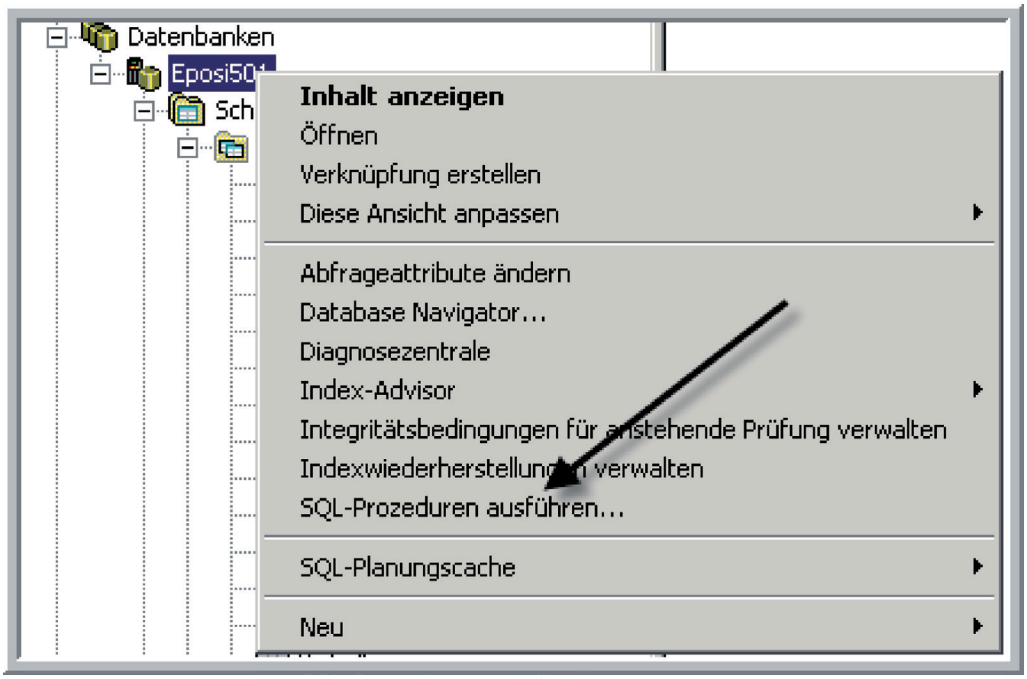
Eine SQL-Prozedur ausführen

Sie sehen, dass ich die Prozedurumgebung aufrufe, indem ich im Taskpad des iSeries Navigators den Punkt „Eine SQL-Prozedur ausführen“ doppelt klicke. Dafür müssen Sie zunächst die Verbindung zum Server hergestellt haben; außerdem muss der Datenbankbereich geöffnet sein. Falls bei Ihnen kein Taskpad angezeigt wird, rufen Sie in der Menüleiste das Menü „ Ansicht“ auf und aktivieren die Anzeige „Taskpad“.



Oberfläche des iSeries Navigator anpassen

Andernfalls können Sie die Prozedurumgebung auch aufrufen, indem Sie den Datenbanknamen mit der rechten Maustaste markieren und anschließend „SQL Prozeduren ausführen…“ auswählen.



Alternativer Aufruf der Prozedurumgebung

Jetzt öffnet sich ein neues Fenster: die Prozedurumgebung. Sie erlaubt es, einzelne oder mehrere SQL-Statements nacheinander abzuarbeiten.

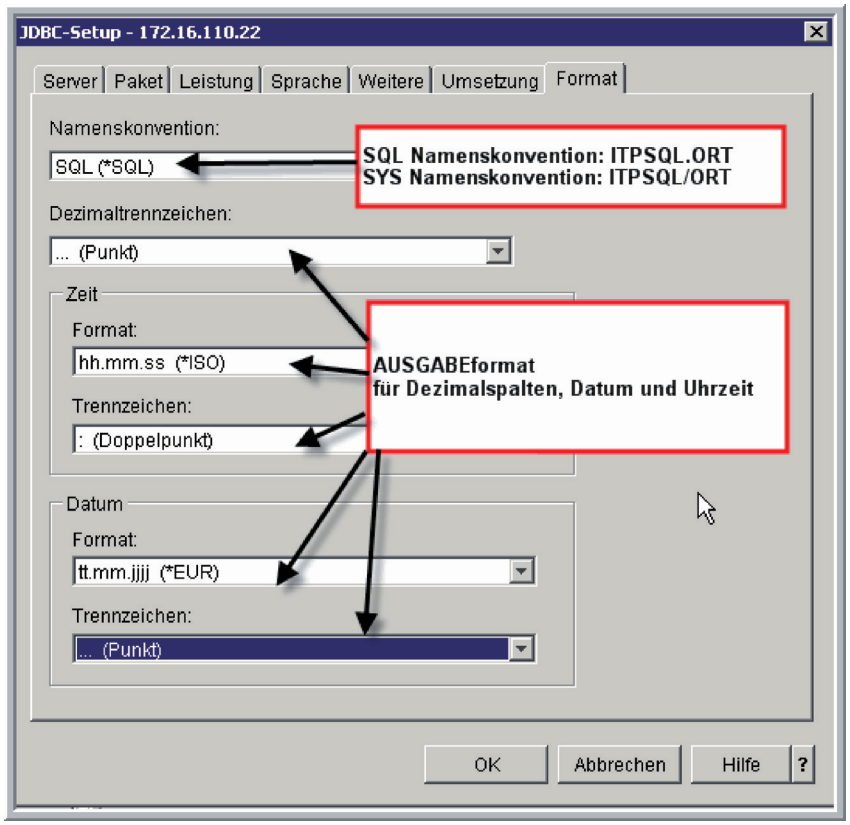
7.4.1.2.2.2 JDBC-Schnittstelle konfigurieren

Das iSeries Scripting-Tool ist in Java geschrieben und stellt die Verbindung zur iSeries über einen JDBC-Treiber her. Diese Verbindung sollten Sie zunächst über das JDBC Setup-Fenster konfigurieren. Öffnen Sie dieses Fenster, indem Sie im Verbindungsmenü „JDBC-Setup…“ wählen.

JDBC-Setup

Das Setup-Menü besteht aus verschiedenen Registern. Es reicht, wenn wir uns im Moment mit den wichtigsten Einstellungen befassen.

7.4.1
Seite 12



JDBC-Setup-Register FORMAT

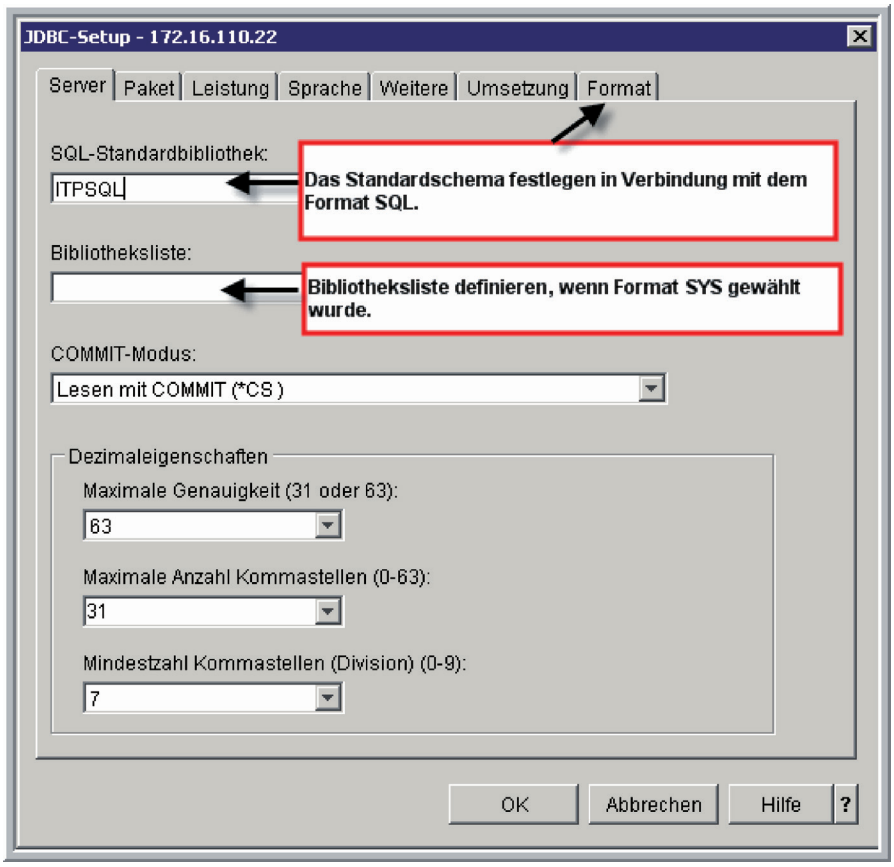
Beginnen wir zunächst mit dem Register „Format“. In diesem Register legen Sie zunächst die Namenskonventionen fest.

ANSI SQL qualifiziert Objekte mit einem Punkt; i5/OS verwendet hingegen einen Slash. Entscheiden Sie sich für ein Format, dass Sie dann durchgängig in allen Umgebungen nutzen. Zusätzlich bestimmen Sie die Zeit- und Datumsformate und legen das Dezimalzeichen fest. Diese Angaben gelten lediglich für die Ausgabe der Daten.

Ein kleiner Tipp:

Entscheiden Sie sich für den Punkt als Dezimaltrennzeichen. Es erspart Ihnen Ärger, denn SQL nutzt das Komma als Spaltentrennzeichen im SELECT-Statement.

Nachdem Sie die Formate bestimmt haben, sollten Sie in das Register „Server“ wechseln.



JDBC-Setup-Register SERVER

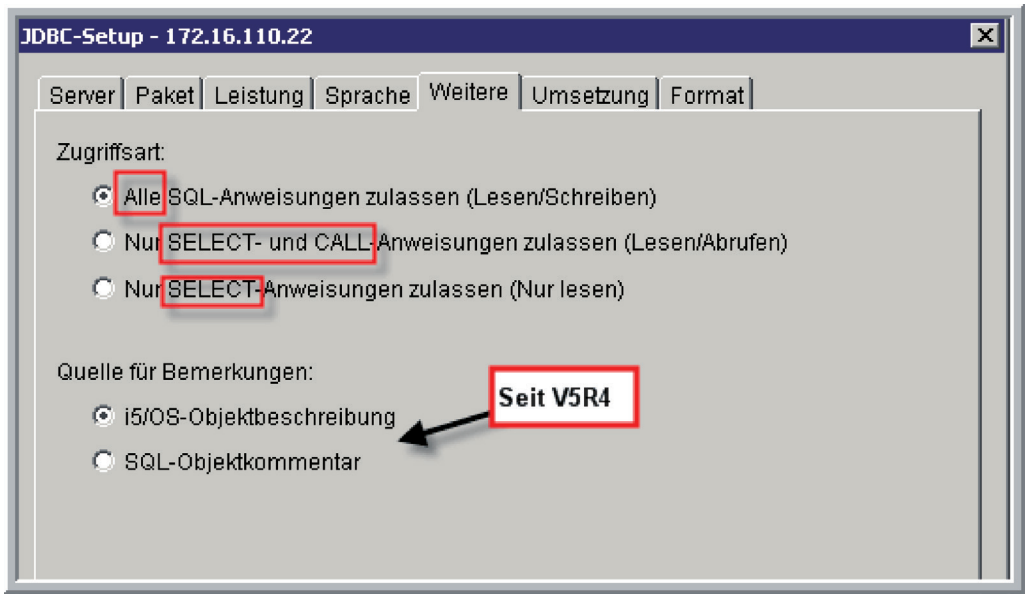
Als SQL-Standardbibliothek tragen Sie jetzt Ihre Arbeitsbibliothek (=Schema) ein. Dieses Schema wird dadurch zur „aktuellen Bibliothek“, wenn Sie im Format SQL arbeiten. Eine andere Möglichkeit haben Sie mit der Bibliothekssuchliste. Dort können Sie mehrere Benutzerbibliotheken hinterlegen – ähnlich dem Befehl ADDLIBLE. Dieser Eintrag ist allerdings nur gültig, wenn Sie sich für das Format *SYS entscheiden. Im Format *SYS haben Sie nicht die Möglichkeit, eine aktuelle Bibliothek zu benennen. Sie müssen also die CURRENT Library als erste Bibliothek in die Suchliste aufnehmen.

Des Weiteren bestimmen Sie den COMMIT-Modus. Beachten Sie, dass die Standardeinstellung „sofortiges COMMIT“ lautet. Das heißt: Ein ROLLBACK ist nach der Befehlsausführung nicht mehr möglich. Ihre Transaktionen sind somit ungesichert. Letztendlich bestimmen Sie in diesem Register auch die Dezimaleigenschaften interner SQL-Variablen.

Die Register „Paket“, „Leistung“ und „Sprache“ können Sie zunächst einmal unbeachtet lassen.

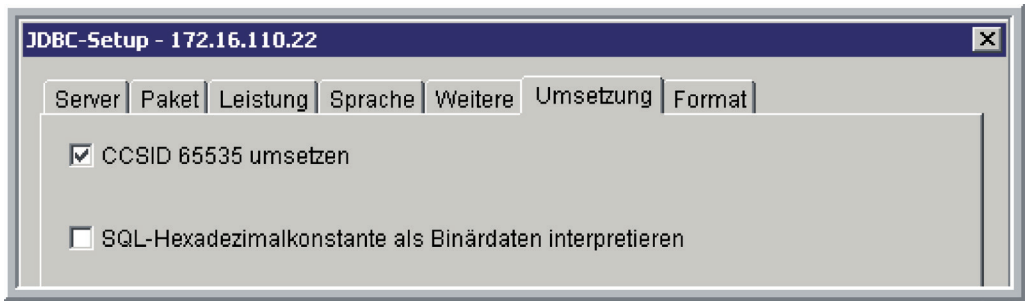
Erwähnenswert ist vielleicht noch das Register „Weitere“ und das Register „Umsetzung“.

7.4.1
Seite 14



Register WEITERE

In diesem Register können Sie die SQL-Zugriffe auf READ-Only beschränken. Außerdem steht seit Release V5R4 eine Alternative für die Quelle von Bemerkungen zur Verfügung. Wenn bisher die Objektbemerkungen aus der Objektbeschreibung abgeleitet wurden, können diese jetzt aus dem SQL-Objektkommentar abgeleitet werden.



Das Register UMSETZUNG

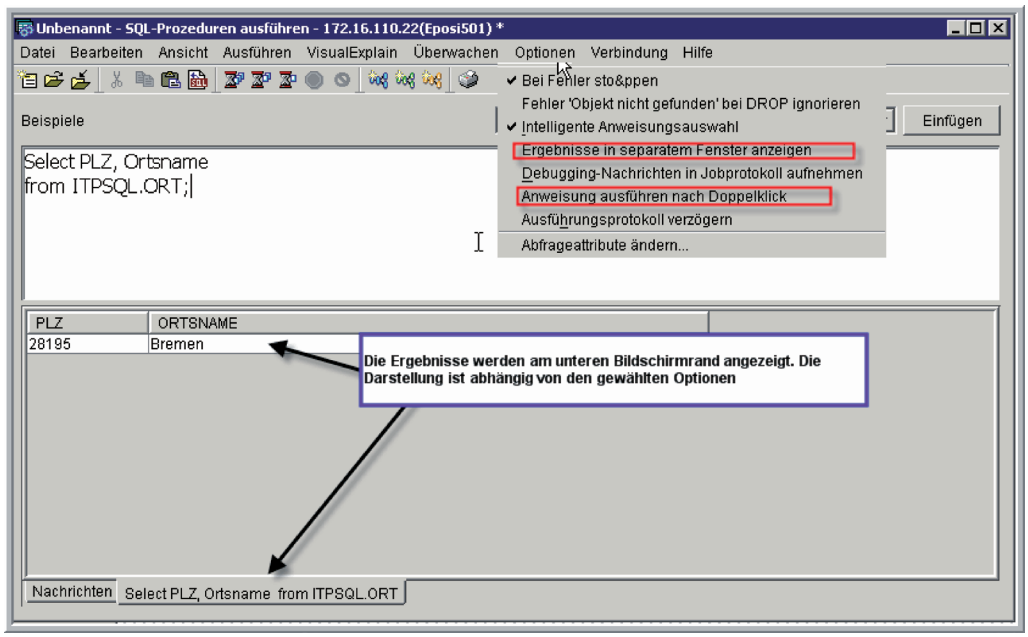
Das Register „Umsetzung“ ist dann interessant, wenn Ihre Tabellen Spalten mit CCSID 65535 enthalten. Diese Spalteninhalte müssten als Hexwerte dargestellt werden – das heißt: Es findet standardmäßig keine Umsetzung der Spalteninhalte statt. Dafür müssten Sie die Option „CCSID 65535 umsetzen“ aktivieren.

Die Option „SQL-Hexadezimalkonstante als Binärdaten interpretieren“ bestimmt den Datentyp, der von Hexadezimalkonstanten in SQL-Ausdrücken verwendet wird. Wenn sie diese Option wählen, verwenden Hexadezimalkonstanten den Datentyp BINARY, andernfalls verwenden Hexadezimalkonstanten den Datentyp CHARACTER FOR BIT DATA.

7.4.1.2.2.3 SQL-Statement in der Skript-Umgebung ausführen

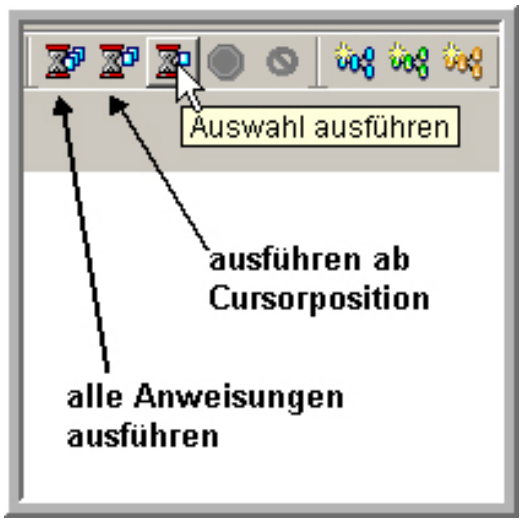
Nachdem wir die Skriptumgebung konfiguriert haben, schauen wir uns nun die Prozedurumgebung etwas genauer an. Im Prozedurfenster können alle Arten von SQL-Befehlen ausgeführt werden.

Das folgende Bild zeigt die Umgebung „SQL Prozeduren ausführen“:



Prozedurumgebung

Wenn Sie bereits wissen, welches SQL-Statement ausgeführt werden soll, tragen Sie dieses im Textfenster ein und starten es mit einem Doppelklick – oder aber nutzen Sie eines der entsprechenden Icons.



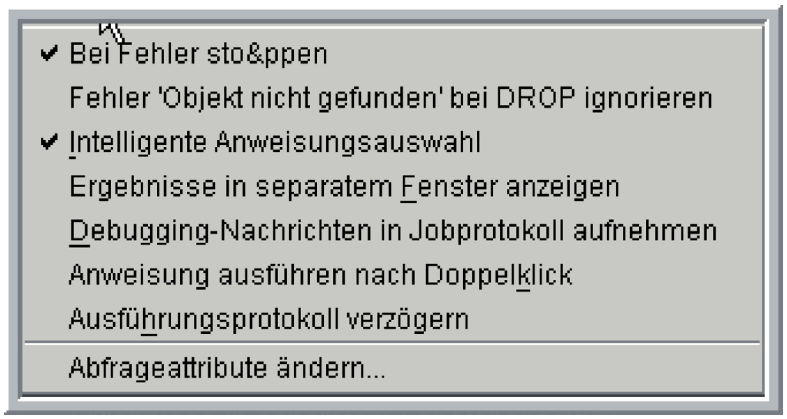
SQL-Anweisungen ausführen

7.4.1
Seite 16

Sie sehen, es stehen drei verschiedene Icons zur Ausführung des SQL-Statements zur Verfügung:

Alle SQL-Statements im Textfeld werden ausgeführt, wenn Sie den Button **„Alle ausführen“** verwenden. Bei der Erzeugung und Durchführung von SQL-Statements kann man zu dem Punkt kommen, an dem viele, aber nicht alle SQL-Statements aus dem Skript ausgeführt werden sollen. Hier bietet der Navigator die Möglichkeit, mit der Durchführung an einem bestimmten Punkt zu beginnen. Dabei werden alle SQL-Statements ignoriert, die im SQL-Skript vor diesem Punkt stehen. Mit einem Klick auf das Icon „Ausführen ab Auswahl“ werden alle SQL-Statements des Textfeldes gestartet. Begonnen wird mit dem Statement, auf dem der Cursor steht. Falls Sie hingegen nur einzelne Statements ausführen wollen, klickt man entweder auf das Icon „Auswahl“, nachdem der Cursor auf das auszuführende Statement platziert wurde, oder man löst das Statement durch einen Doppelklick aus. Hierfür müssen Sie allerdings die Option „Anweisung ausführen nach Doppelklick“ aktiviert haben.

Die Ergebnisse Ihrer Abfrage erscheinen in einem separaten Fenster oder im unteren Bildschirmbereich. Auch dies ist abhängig von der Optionswahl:



Die Optionen

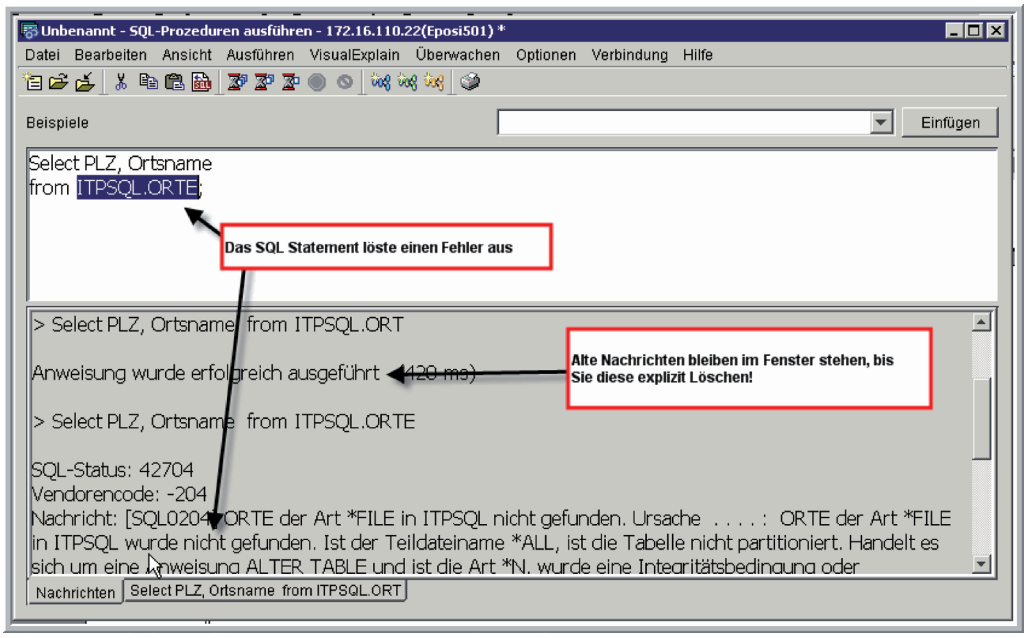
- **Bei Fehler stoppen**
  Steuert den Ablauf des Scripting-Tools, wenn mehrere SQL-Statements nacheinander durchgeführt werden. Wenn ein SQL-Statement einen Fehler aufweist, wird die Durchführung der restlichen SQL-Statements abgebrochen. Falls diese Option nicht aktiviert ist, wird jedes SQL-Statement unabhängig voneinander ausgewertet. Alle gültigen Statements werden verarbeitet.
- **Fehler ’Objekt nicht gefunden’ bei DROP ignorieren**
  Diese Option steht erst seit V5R4 zur Verfügung. Sollte in Ihrem SQL-Skript die Anweisung DROP einen Fehler verursachen, weil das Objekt nicht existiert, wird der Fehler ignoriert. Die weiteren Anweisungen werden trotzdem ausgeführt.

- **Intelligente Anweisungsauswahl**
  Diese Option stellt sicher, dass Anweisungen auch dann ausgeführt werden, wenn Sie nur teilweise markiert wurden. „Intelligente Anweisungsauswahl“ stellt somit sicher, dass Anweisungen vollständig ausgeführt werden, selbst wenn eine oder mehrere Anweisungen nur teilweise markiert sind.
- **Ergebnisse in separatem Fenster anzeigen**
  Jedes Mal, wenn ein SQL-Statement läuft und eine Ergebnistabelle erzeugt wird, erscheint diese Ergebnistabelle als separates Desktop-Fenster. Bei Nichtaktivierung dieser Option erscheinen die Ergebnisse am Ende des Fensters.
- **Debugging-Nachrichten in Jobprotokoll aufnehmen**
  Mit dieser Option werden alle auftretenden Diagnosefehler im Jobprotokoll dieser Session aufgeführt. Das Jobprotokoll kann aufgerufen werden, indem man im Menü „Ansicht“ „Jobprotokoll anzeigen“ auswählt.
- **Anweisungen ausführen nach Doppelklick**
  Mit dieser Option werden SQL-Statements durchgeführt, wenn man sie doppelt anklickt. Semikolons müssen dabei das Ende jedes SQL-Statements kennzeichnen.
- **Ausführungsprotokoll verzögern**
  Auch diese Option ist mit Release V5R4 implementiert worden. Sie bewirkt, dass die Ausgabe von Nachrichten im unteren Fensterbereich aus Leistungsgründen verzögert wird.

Welche Optionen Sie wählen, ist individuell verschieden. Ich arbeite lieber mit Ergebnissen im unteren Fensterbereich, anstatt immer neue Fenster für Result-Sets zu öffnen. Außerdem übernehme ich alle Debug-Nachrichten in das Jobprotokoll, sobald die SQL-Statements komplexer werden. Dadurch stehen mir mehr Informationen bezüglich der Ausführung zur Verfügung. Für den Anfang sollten die oben gezeigten Optionen allerdings ausreichend sein.

Eine letzte Anmerkung zu den Fehlermeldungen. Fehlermeldungen, aber auch Informationen über die korrekte Ausführung der SQL-Statements, stehen ebenfalls am Ende des SQL-Script-Fensters und werden nicht automatisch gelöscht.

7.4.1
Seite 18



SQL-Fehlermeldungen

Wenn ein SQL-Statement – wie in diesem Fall – nicht korrekt ausgeführt wurde, sehen Sie sofort die entsprechende Fehlermeldung. Die aktuelle Meldung steht dabei jeweils am Ende. Im Nachrichtenregister bleiben alle Meldungen stehen, bis Sie diese löschen. Löschen können Sie das Ausführungsprotokoll im Menü „Bearbeiten“, indem Sie „Ausführungsprotokoll löschen“ auswählen.

Im Gegensatz zum interaktiven Tool der nativen Umgebung hat das Navigator SQL-Scripting-Tool keine „Erinnerungsfunktion“. Im Gegensatz zum STRSQL-Tool in nativer Umgebung merkt sich dieses Tool nicht automatisch die vorangegangenen SQL-Statements. Wenn Sie die Skript-Umgebung verlassen und anschließend erneut aufrufen, stehen Ihnen die vorangegangenen Statements nicht mehr zur Verfügung. Sie müssen also, nachdem Sie eine Anzahl von SQL-Statements kodiert haben, diese speichern, um sie zu einem späteren Zeitpunkt wieder zu verwenden. Dafür klicken Sie auf die Option „Speichern unter“ im Dateimenü. Es erscheint ein Dialogfeld „Speichern unter“, das für Windows typisch ist.

Diese Prozedurumgebung erlaubt es uns natürlich auch, DDL auszuführen. Somit können Sie wählen, ob Sie Ihre Tabellen mittels der grafischen Benutzeroberfläche oder als DDL-Skript in der Prozedurumgebung erstellen wollen. Diese Entscheidung ist sicherlich Geschmackssache und natürlich vom konkreten Anwendungsfall abhängig. Ich habe meine Datenbank mit einem SQL-Skript erstellt, das Sie im Anhang finden und gern als Beispieldatenbank für die folgenden Beiträge nutzen können.

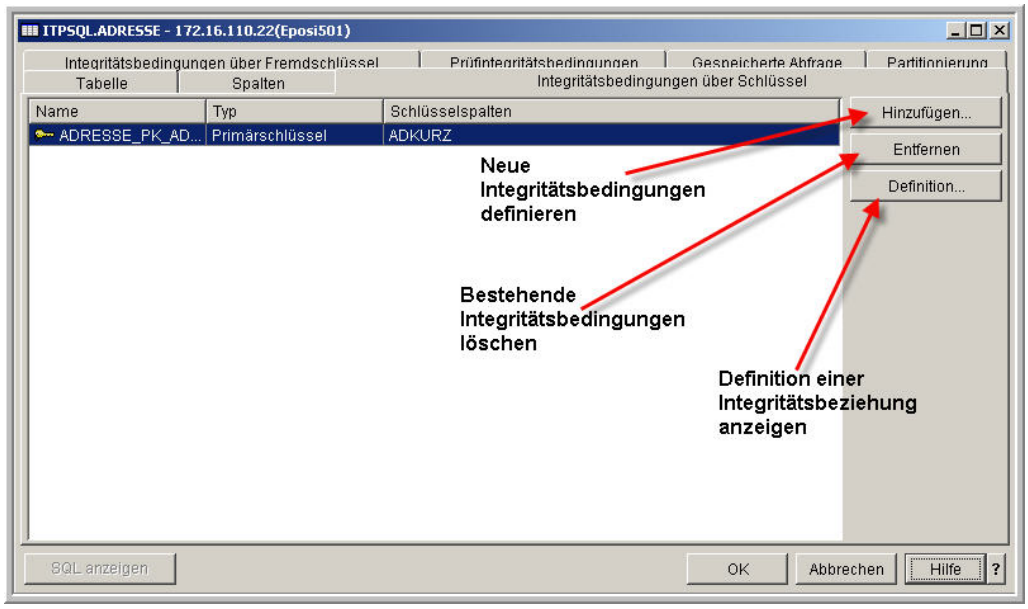
7.4.1.3 Mit Integritätsbedingungen arbeiten

Wie bereits erwähnt ist iSeries Navigator eine Grafikschnittstelle, mit der Sie viele der üblichen Verwaltungsvorgänge in einer Datenbank ausführen können. Zwar basieren die meisten Operationen in iSeries Navigator auf Structured Query Language (SQL), jedoch müssen Sie SQL nicht vollständig verstehen, um Ihre Integritätsbedingungen mit Hilfe des iSeries Navigators zu definieren bzw. bestehende Datenbank-Bedingungen zu verwalten. iSeries Navigator unterscheidet zwischen

- Integritätsbedingungen über Schlüssel,
- Integritätsbeziehungen über Fremdschlüssel
- und Prüfintegritätsbeziehungen.

Die Unterschiede, die Vor- und Nachteile der unterschiedlichen Integritätsbedingungen haben wir bereits ausführlich in den Kapiteln 7.2.4.1 und 7.3.4 behandelt. Daher werde ich in diesem Kapitel nur die iSeries Navigator-Funktionen erläutern, ohne erneut detailliert auf Bedeutung und Unterschiede der Integritätsbedingungen einzugehen.

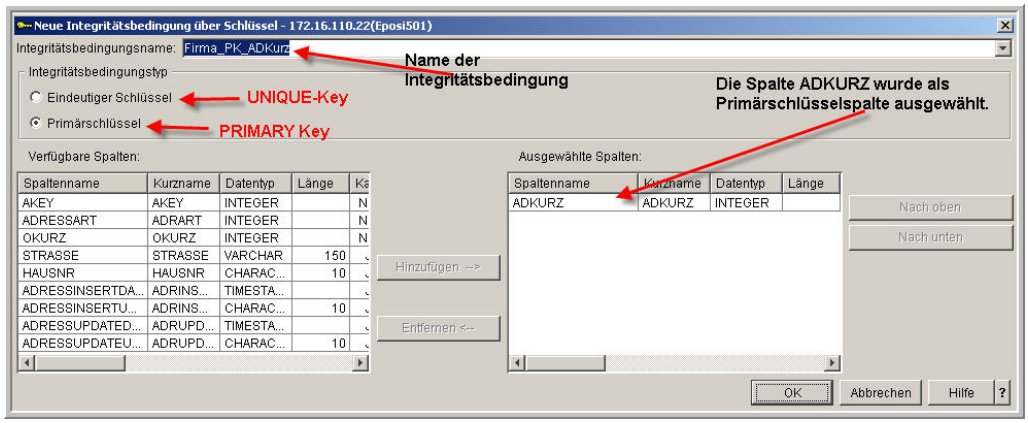
Wenn Sie in iSeries Navigator Integritätsbeziehungen über Schlüssel hinzufügen, können Sie einen Primärschlüssel oder mehrere eindeutige Schlüssel definieren. Dazu erweitern Sie den Eintrag „Datenbanken“, klicken anschließend auf den Eintrag „Tabelle“ und markieren eine Tabelle. Im Kontextmenü der Tabelle müssen Sie anschließend den Eintrag „Definition“ öffnen. Auf der Seite „Integritätsbedingungen über Schlüssel“ sind die Integritätsbedingungen über Schlüssel aufgeführt, die für diese Tabelle definiert sind.



Schlüsseldefinitionen verwalten

In meinem Beispiel verwende ich die Tabelle „Adresse“ unserer Beispieldatenbank. Für die Tabelle existiert bereits eine Integritätsbedingung. Diese Bedingung könnten Sie löschen, wenn Sie auf den Button „Entfernen“ klicken. Die Definition der Bedingung können Sie anzeigen, indem Sie den Button „Definition“ nutzen. Um neue Primär- oder Unique-Schlüssel zu definieren, verwenden Sie hingegen die Schaltfläche „Hinzufügen“.

Im Dialog „Neue Integritätsbedingung über Schlüssel“ können Sie der Integritätsbeziehung einen Namen geben. Dazu wählen Sie den Schlüsseltyp und die entsprechende Schlüsselspalte. In dem nachfolgenden Beispiel können Sie sehen, dass ich für die Tabelle „Adresse“ einen Primärschlüssel definiere:



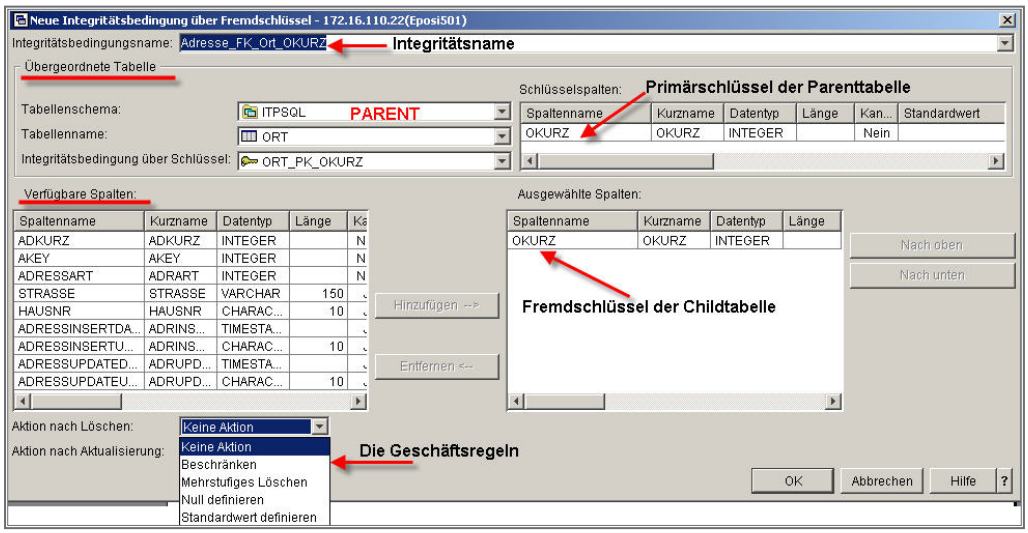
Primärschlüsseldefinition

Mit dem Integritätsbedingungstypen definieren Sie einen Primärschlüssel oder einen eindeutigen Schlüssel, der aus den Spalten besteht, die im Spaltenauswahlraster angegeben wurden. Die angegebenen Spalten dürfen nicht mit Spalten übereinstimmen, die bereits in einer anderen Integritätsbedingung für dieselbe Tabelle angegeben wurden. Beachten sollten Sie auch, dass die Spaltenauswahl A, B identisch ist mit der Auswahl B, A. Klicken Sie auf „OK“, wenn Sie alle Eingaben gemacht haben, um zum Dialog „Tabellendefinition“ zurückzukehren. Mit einem erneuten „OK“ bestätigen Sie Ihre Tabellendefinition. Die soeben erstellte Integritätsbedingung „Firma_PK_AdKurz“ können Sie jetzt nicht mehr ändern. Falls Sie die Eigenschaften der Integritätsbedingung dennoch ändern wollen, müssen Sie die bestehende Bedingung zunächst löschen und anschließend mit den gewünschten Änderungen neu erstellen. Änderungen der Eigenschaften einer Integritätsbedingung sind leider nur in der aktuellen Sitzung zulässig, d. h. während der Definition.

Wenn wir anschließend unseren Primärschlüssel mit einem Fremdschlüssel verknüpfen, entsteht eine Beziehung zwischen zwei Tabellen, die als referentielle Integritätsbedingung bezeichnet wird. Referentielle Integritätsbedingungen stellen eine Beziehung zwischen einer abhängigen Tabelle (Child) und einer übergeordneten Tabelle (Parent) her. Das System garantiert, dass es für jede Zeile der abhängigen Tabelle mit einem Wert ungleich Null eine Zeile in der

übergeordneten Tabelle mit einem entsprechenden Wert gibt. Wenn Ihre Geschäftsregel als referentielle Integritätsbedingung definiert ist, kann kein Programm, das versucht eine Zeile einzufügen, zu löschen oder zu aktualisieren, diese Integritätsbedingung verletzen.

Schauen wir uns auch hierfür ein entsprechendes Beispiel an! Ich möchte zwischen der Tabelle „Adresse“ und der Tabelle „Ort“ eine referentielle Integritätsbedingung über die Spalte „OKurz“ aufbauen. Voraussetzung hierfür ist, dass für die Tabelle „Ort“ bereits ein Primary Key oder ein Unique Key existiert. Grundsätzlich definieren Sie die referentiellen Bedingungen immer in der abhängigen Child-Tabelle. In unserem Beispiel ist dies die Tabelle „Adresse“. Öffnen Sie also erneut den Dialog „Definition“ der Tabelle „Adresse“ und wählen Sie dieses Mal die Indexzunge „Integritätsbedingungen über Fremdschlüssel“. Es erscheint der folgende Dialog, nachdem Sie auf die Schaltfläche „Hinzufügen“ geklickt haben:

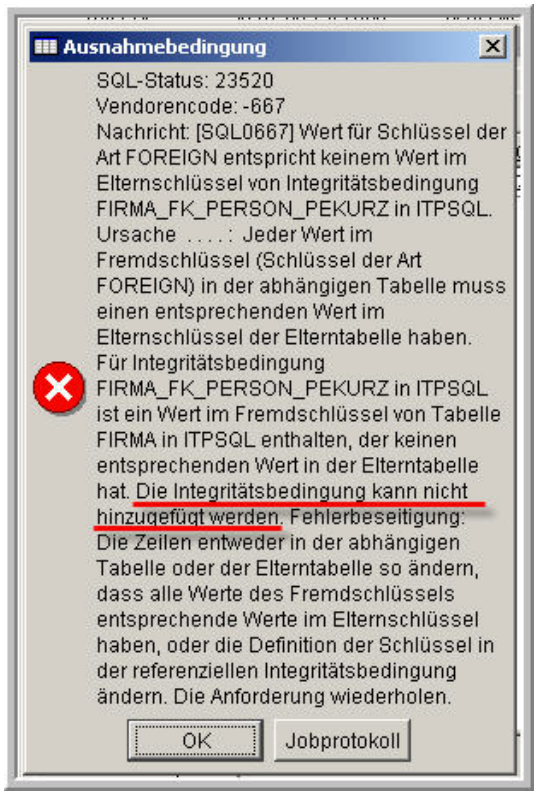


Definition einer referentiellen Integritätsbedingung

In dem Dialog müssen Sie zunächst die Elterntabelle auswählen – in unserem Beispiel die Tabelle „Ort“. Daraufhin wird automatisch die Primary Key Definition der Tabelle angezeigt. In einem weiteren Schritt wählen Sie jetzt die Fremdschlüsselspalte der Tabelle „Adresse“ aus. Zu guter Letzt bestimmen Sie die Geschäftsregeln, d.h. Sie müssen festlegen, was geschehen soll, wenn eine Applikation einen Elternsatz löschen will, zu dem noch abhängige Child-Sätze vorhanden sind. Die Erläuterung der unterschiedlichen Geschäftsregeln entnehmen Sie bitte dem Kapitel 7.2.4.1. Anschließend bestätigen Sie den Dialog wieder mit „OK“.

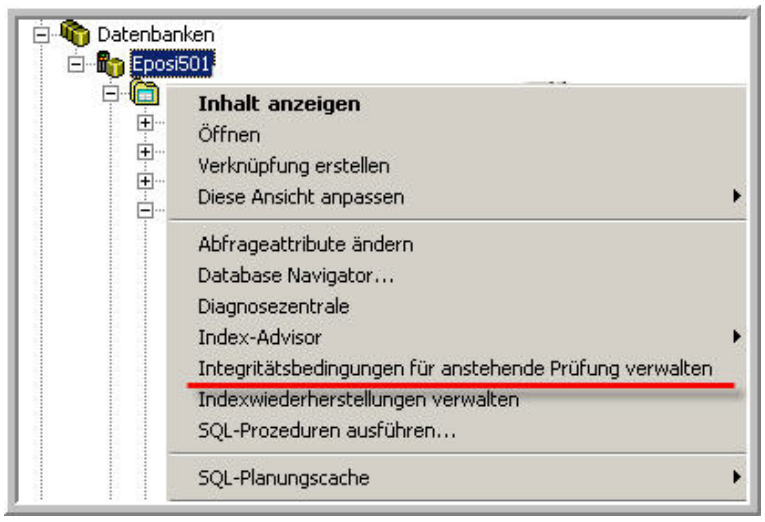
7.4.1
Seite 22

Falls die aktuelle Datenlage der Tabellen gegen die referentielle Bedingung verstößt, werden Sie die folgende Fehlermeldung erhalten:



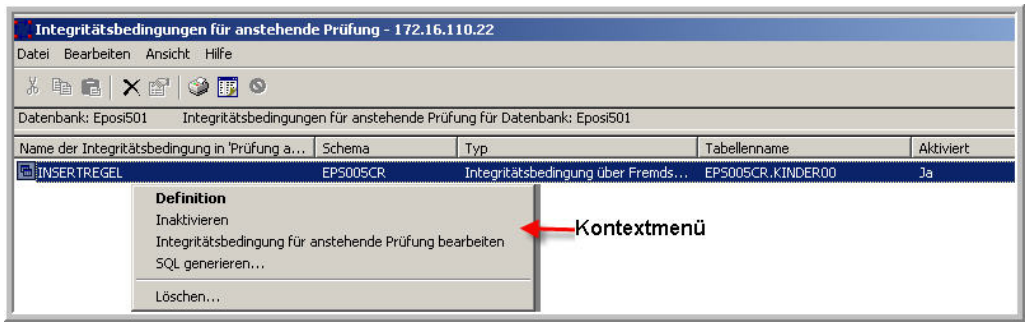
Fehlerhafte Integritätsbedingung

Anders als in der i5/OS-Umgebung entstehen keine Check Pending-Situationen, wenn Sie mit iSeries Navigator die Geschäftsregeln implementieren. Sie müssen also bereits vorher dafür gesorgt haben, dass Ihre Daten den Regeln entsprechen. Allerdings ist es durchaus möglich mit iSeries Navigator Check Pending-Situationen aufzulösen, die entstehen können, wenn Sie Integritätsbedingungen mit nativen i5/OS-Befehlen implementieren. Hierzu öffnen Sie das Kontextmenü Ihrer Datenbank:



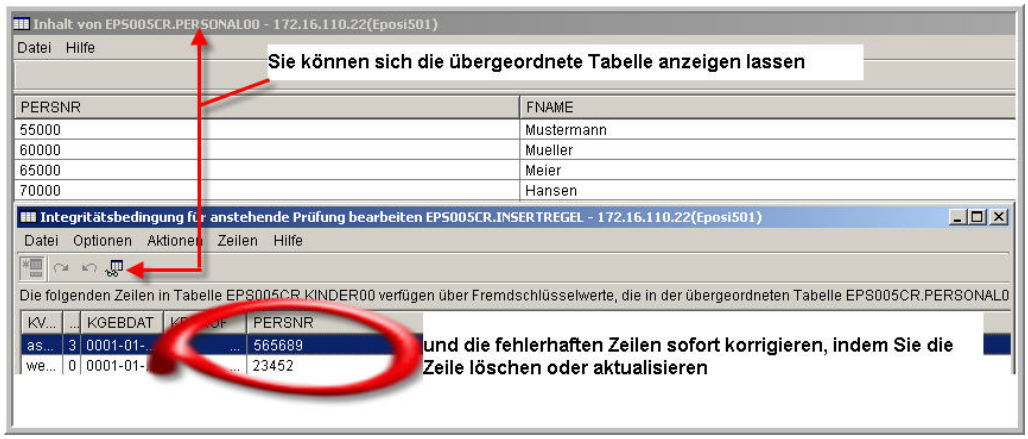
Kontextmenü der Datenbank

Wählen Sie anschließend den Eintrag „Integritätsbedingungen für anstehende Prüfung verwalten“. Daraufhin erscheint ein weiterer Dialog mit aktiven Integritätsbedingungen, die den Status „Disabled“ aufweisen.



Integritätsbedingungen mit anstehender Prüfung

Sie sehen, bei mir gibt es im Schemata „Eps005CR“ eine Regel mit anstehender Prüfung. Ich kann mir jetzt über iSeries Navigator die Definition anzeigen lassen, die Regel deaktivieren, sie löschen oder die fehlerhaften Daten bearbeiten. Um letzteres auszuführen wähle ich den dritten Eintrag: Integritätsbedingung für anstehende Prüfung bearbeiten. Daraufhin erscheint ein weiteres Fenster mit fehlerhaften Datensätzen:



Check Pending-Situation anzeigen

Das Gute an der Bearbeitung mit iSeries Navigator ist, dass ich mir zusätzlich die Parent-Tabelle anzeigen lassen kann, um die fehlerhaften Daten, die die Integritätsbedingung in den Status „Prüfung anstehend“ versetzt haben, zu prüfen. Jetzt müssen Sie die fehlerhaften Daten entweder korrigieren, indem Sie auf die fehlerhafte Spalte klicken oder die Zeilen löschen.

7.4.1
Seite 24

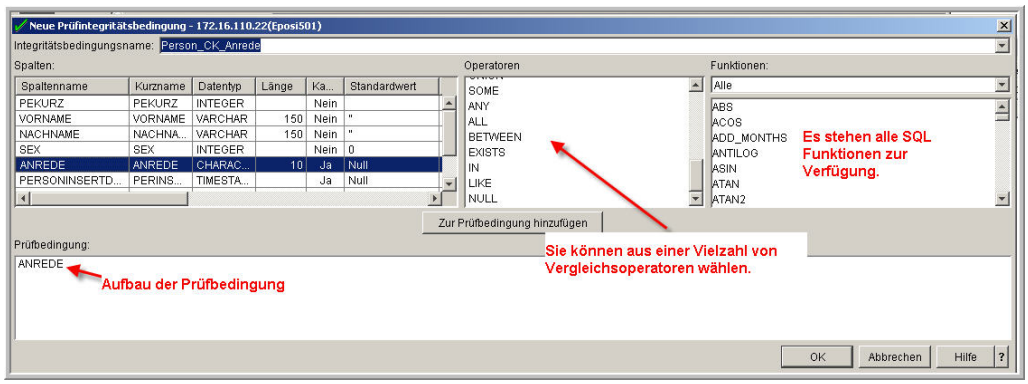
Sie können mit iSeries Navigator aber nicht nur Primärschlüssel und referentielle Bedingungen definieren, sondern auch Prüfbedingungen lassen sich einfach hinterlegen. Eine Prüfintegritätsbedingung wird für eine Spalte erstellt, um sicherzustellen, dass der Wert dieser Spalte einem bestimmten, durch Ihre Geschäftsregeln festgelegten Wert oder Wertebereich entspricht. Wenn Sie eine Spalte mit einer Prüfintegritätsbedingung einfügen oder aktualisieren, vergleicht die Integritätsbedingung die neuen Daten mit der Geschäftsregel und verhindert alle Versuche, diese zu verletzen.

Auch Prüfbedingungen hinterlegen Sie als Tabellendefinition. Sie öffnen also wieder das Kontextmenü einer markierten Tabelle und wählen den Eintrag „Definition“. Wenn Sie auf die Indexzunge „Prüfintegritätsbedingungen“ klicken, sehen Sie alle Prüfintegritätsbedingungen, die für diese Tabelle definiert sind.

Prüfintegritätsbedingungen verwalten

Sie können eine neue Integritätsbedingung hinzufügen, eine Integritätsbedingungsdefinition anzeigen oder eine Integritätsbedingung entfernen. Klicken Sie auf „Hinzufügen“, um eine neue Integritätsbedingung zu erstellen. Klicken Sie auf „Entfernen“, um eine Integritätsbedingung zu entfernen. Klicken Sie auf „Definition“, um die Definition einer Integritätsbedingung anzuzeigen. Nach der Erstellung einer Integritätsbedingung können Sie die Definition der Integritätsbedingung nicht ändern. Wenn Sie eine Prüfintegritätsbedingung ändern wollen, müssen Sie die Integritätsbedingung zuerst löschen und dann neu erstellen.

Im Dialog „Neue Prüfintegritätsbedingungen“ können Sie weitere Prüfintegritätsbedingungen für die Tabelle definieren.



Neue Prüfintegritätsbedingung definieren

In dem Dialog erstellen Sie zunächst einen Ausdruck für eine oder mehrere Spalten. Um eine Spalte, einen Operator oder eine Funktion zu einem Ausdruck hinzuzufügen, heben Sie das entsprechende Element hervor und klicken auf „Zur Prüfbedingung hinzufügen“. Sie können sehen, dass sich die Bedingung im Fenster „Prüfbedingung“ aufbaut. Abschließend bestätigen Sie wieder Ihre Eingaben mit „OK“.

7.4.1
Seite 26

7.4.1.4 Mit Statistikdaten arbeiten

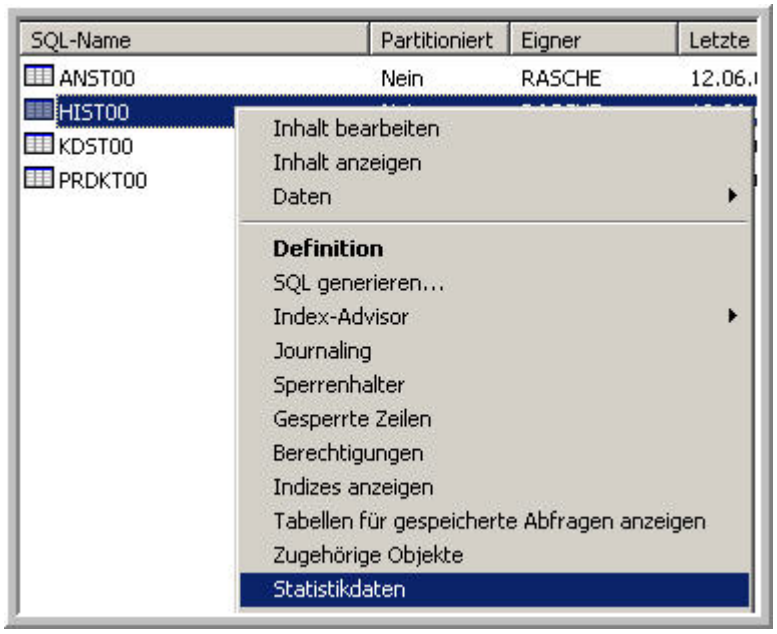
Vor Release V5R2 war allein der Optimizer für das Sammeln von Informationen zuständig, d. h. der Optimizer prüfte während des Optimierungsprozesses die Tabelleneigenschaften und deren Indizes, um anschließend anhand der ermittelten Daten den Zugriffsplan zu erstellen. Die CQE arbeitet noch heute nach diesem Prinzip.

Für die SQE gibt es allerdings seit dem Release V5R2 eine eigenständige Komponente – den Statistik Manager – der diese Aufgabe erfüllt. Diese Komponente ist nicht für die Optimierung oder Ausführung des SQL Statements verantwortlich, sondern hat die Kontrolle auf den Zugriff der Metadaten einer Tabelle und verfügt über weitere Informationen, die für den Optimierungsprozess notwendig sind. Man kann sagen, dass der Statistik Manager zwei grundlegende Aufgaben zu erfüllen hat:

- Einerseits erstellt und pflegt er Statistikinformationen für einzelne Spalten einer Tabelle,
- andererseits beantwortet der Statistik Manager Fragen der SQE, die für die Ermittlung des optimalen Zugriffplanes relevant sind.

System i Navigator enthält daher seit Release V5R2 einige Funktionen, um mit Statistikdaten zu arbeiten.

Sehen wir uns in einem ersten Schritt die Statistikdaten der Tabelle „HIST00“ an. Hierfür erweitere ich zunächst den Eintrag für die entsprechende Datenbank, öffne das Schema, das die Tabelle enthält, und klicke mit der rechten Maustaste auf die Tabelle. Daraufhin erscheint das folgende Kontextmenü:



Statistikdaten der Tabelle HIST00 anzeigen

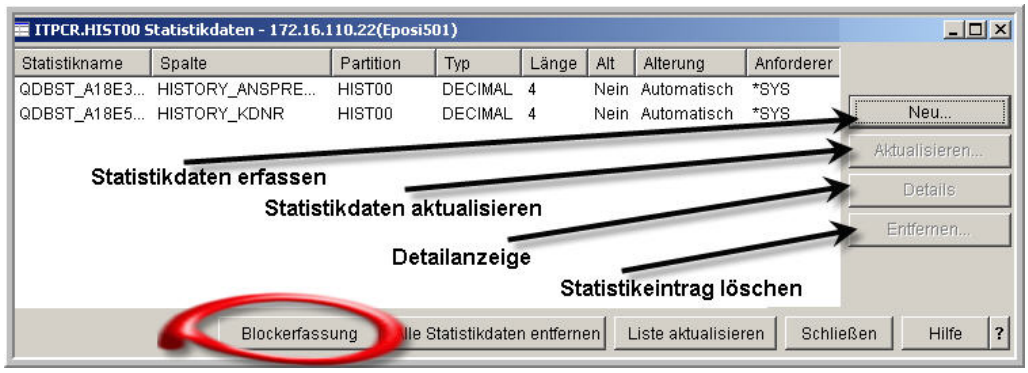
7.4.1
Seite 28

Falls bei Ihnen andere Menüpunkte dargestellt werden, könnte es an der Version des Navigators liegen. Ich arbeite mit der Version 6. iSeries Navigator, heißt übrigens seit Release V6R1 System i Navigator.

Tipp:

Sie können System i Navigator Version 6 selbst dann installieren, wenn Sie noch mit der i5/OS Version V5R4 arbeiten.

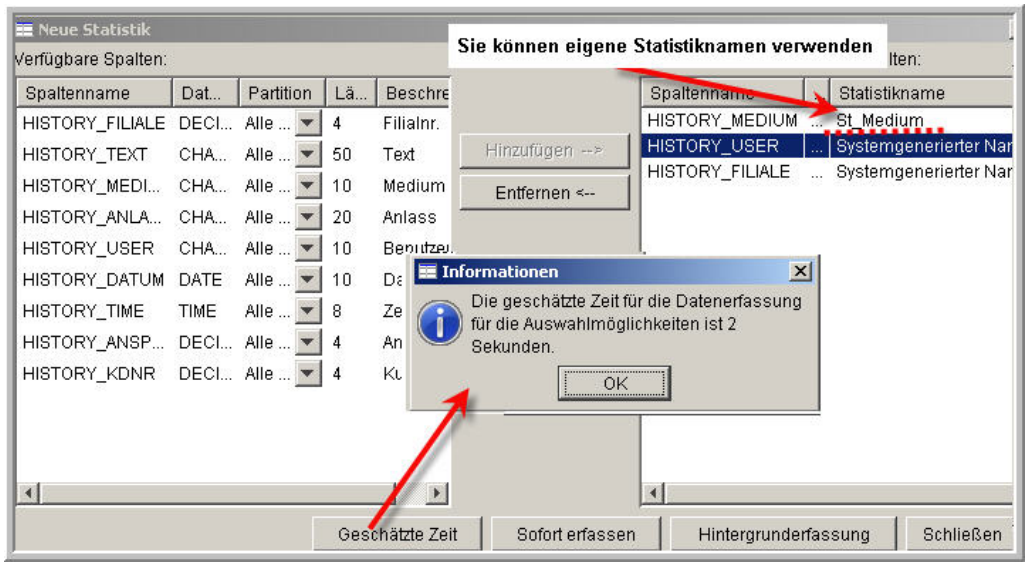
Als nächstes wähle ich den Eintrag „Statistikdaten“. Daraufhin öffnet sich ein neues Fenster mit den vorhandenen Spaltenstatistiken der Tabelle:



Verfügbare Statistikinformationen der Tabelle „HIST00“

Falls Sie jetzt verwundert feststellen, dass auf Ihrem System keine oder nur sehr wenig Statistiken existieren, könnte es daran liegen, dass die SQL Statements vielleicht nicht auf der SQE ausgeführt werden! Vielleicht weil logische DDS Dateien im Statement verwendet werden oder abgeleitete Indizes die Ausführung verhindern. Lesen Sie das Kapitel 7.6.1, um weitere Informationen zu erhalten. Ein anderer Grund könnten zahlreiche Indizes sein. Wenn sehr viele logische DDS Dateien oder SQL Indizes auf dem System existieren, können die benötigten Informationen anhand der Indizes geliefert werden. Entsprechende Statistikdaten werden in diesem Fall vom Statistik Manager nicht automatisch erstellt.

Für meine Tabelle „HIST00“ existieren bereits vom System erstellte Statistikinformationen. Ich könnte aber auch manuell Statistikdaten anfordern. Hierzu wähle ich „Neu“ und selektiere im folgenden Bildschirm die entsprechenden Spalten.



Statistiken manuell erstellen

Sie können Ihre Anforderung individuell benennen, die Ausführungszeit anzeigen und den Job interaktiv ausführen oder als Batchjob übergeben. Außerdem können Sie einzelne Statistiken „aktualisieren“ oder „entfernen“ oder global alle vorhandenen Statistiken aktualisieren (Liste aktualisieren) bzw. löschen (Statistikdaten entfernen). Der Button „Blockerfassung“ wurde schlecht übersetzt, wie ich finde. Wenn Sie den Button „Blockerfassung“ nutzen, unterbinden Sie die Statistikerfassung für die Tabelle „HIST00“. Die Anforderungen werden allerdings im Plan Cache gespeichert und ausgeführt, sobald Sie die Sperre wieder aufheben.

Für die Hintergrunderfassung der angeforderten Statistikdaten ist der Job „QDBFSTCCOL“ zuständig. Lassen Sie sich den Job anzeigen. Geben Sie hierfür folgenden Befehl auf einer 5250-Befehlszeile ein:

```
WRKJOB QDBFSTCCOL
```

7.4.1
Seite 30

Anschließend finden Sie im JOBLOG des Jobs Informationen über die ausgeführten Statistikanforderungen:

```
 Job . . :   QDBFSTCCOL    Benutzer :   QSYS

_      aktualisiert.
     1 Statistikerfassung(en) wurde(n) angefordert.
     1 Statistikerfassung(en) wurde(n) angefordert.
     1 Statistikerfassung(en) wurde(n) angefordert.
     1 Statistikerfassung(en) wurde(n) angefordert.
     1 Statistikerfassung(en) wurde(n) angefordert.
     Statistikerfassung QDBST_172C1801C48818B4BB180004AC0C02D6
       aktualisiert.
```

Das Joblog des Jobs QDBFSTCCOL

Nähere Informationen über die Tabelle und den Ausführungsgrund stehen in den Detailnachrichten. Aufgrund der niedrigen Priorität dieses Jobs und der Bündelung von Statistikermittlungen einer Tabelle wird Ihr Server durch die Statistikanforderungen kaum belastet. In der folgenden Grafik sehen Sie die einzelnen Threads dieses Jobs:

Thread	Status	Gesamt-CPU	Aux I/O	Ausführungs-priorität
00000001	EVTW	0,025	519	50
00000007	DEQW	0,000	0	99
00000006	DEQW	0,000	0	99
00000005	DEQW	0,017	283	96
00000004	DEQW	0,019	313	96
00000003	DEQW	0,050	102	93
00000002	DEQW	0,000	2	90

Threads des Jobs QDBFSTCCOL

Der erste Eintrag mit der Priorität 50 ist für die Statistikerfassung irrelevant. Threads der Priorität 99 und 96 werden aktiv, wenn das System Statistiken anfordert. Threads der Priorität 93 und 90 führen Benutzeranforderungen aus. Jeder Job mit einer höheren Priorität wird vorrangig verarbeitet, auch wenn das Zeitscheibenende des Jobs QDBFSTCCOL bislang nicht erreicht wurde. Allerdings nutzt der Job, wenn er aktiv wird, alle verbleibenden Systemressourcen, um fertig zu werden, was vorübergehend zu einer erhöhten CPU Auslastung Ihres Systems führen könnte.

TIPP:

Falls Sie selbst eine Art „Dauerjob“ mit niedriger Priorität nutzen, könnte dieser Job die Erhebung der Statistikdaten verhindern. Ändern Sie für diesen Fall die Priorität Ihres Jobs!

Auf vielen Plattformen ist die Erfassung von Statistikdaten ein manueller Prozess, für den der Datenbankadministrator verantwortlich ist. Bei System i Servern erfolgt die Erfassung der Statistikdaten allerdings weitestgehend automatisch, so dass der soeben dargestellte interaktive Dialog eher die Ausnahme sein wird.

Tatsächlich bestimmt aber der Systemwert „QDBFSTCCOL“ die Art der Ermittlung. Da dieser Wert als Standard den Wert *ALL nutzt, kann sowohl das System als auch der Benutzer Statistikdaten erheben.

```
Systemwert . . . . . :   QDBFSTCCOL
Beschreibung . . . . . :   Statistikerfassung für Datenbankdatei


Statistikerfassung für
  Datenbankdatei . . . . . :   *ALL           *ALL
                                              *SYSTEM
                                              *USER
                                              *NONE
```

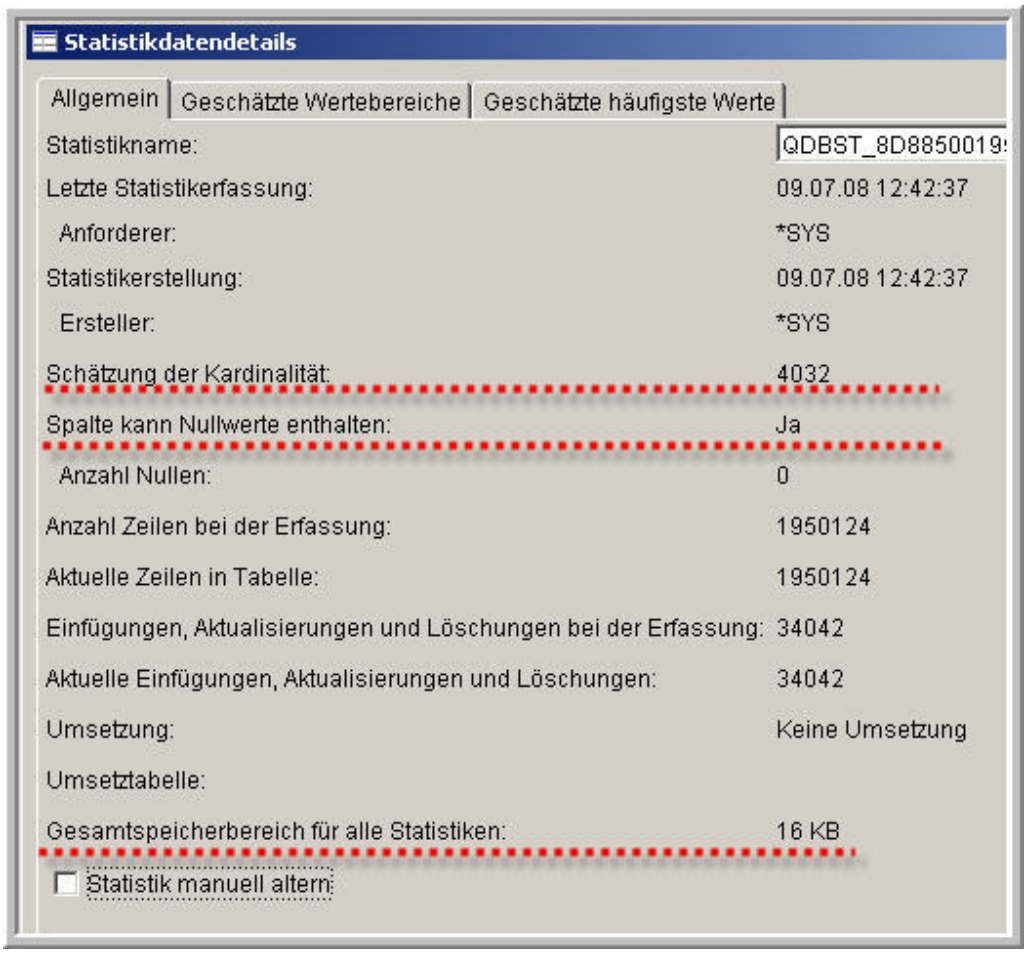
Der Systemwert QDBFSTCCOL

Die weiteren Einstellungen dieses Systemwertes sprechen für sich und müssen nicht näher erläutert werden. Eine Anmerkung vielleicht noch zur Einstellung *NONE. Der Wert *NONE unterbindet die Statistikerfassung für das gesamte System, solange Sie diese Einstellung beibehalten. Allerdings werden auch jetzt die Anforderungen im Plan Cache hinterlegt und ausgeführt, sobald der Systemwert wieder auf den Wert *ALL oder *SYSTEM geändert wird.

Schauen wir uns in einem nächsten Schritt an, welche Statistikwerte für die einzelnen Spalten erfasst werden. Ich markiere hierfür den Eintrag „History_Kdnr“ und klicke auf den Button „Details“.

7.4.1
Seite 32

Als erstes erscheinen Metadaten der Tabelle:



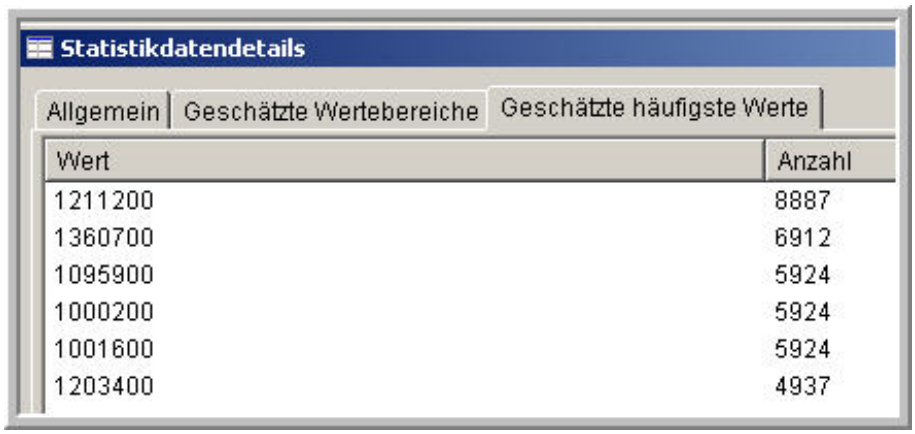
Allgemeine Statistikdetails

Die Tabelle „HIST00“ enthielt zum Zeitpunkt der Statistikerfassung beispielsweise 1.950.124 Zeilen und hatte 4.032 eindeutige Spaltenwerte. NULL-Werte gab es in dieser Spalte nicht. Die Statistikinformationen werden im Tabellenobjekt gespeichert, benötigen aber relativ wenig Speicher. Für die gezeigten Statistikinformationen der Tabelle „HIST00“ benötigt das System gerade einmal 16 KB.

Das Register „Geschätzte Wertebereiche“ (Histogramm) gruppiert die Spaltenwerte. Die hier gezeigten Werte basieren allerdings auf einer Stichprobe der Zeilen in der Tabelle und können von der realen Datenlage abweichen.

Das Register „Geschätzte Wertebereiche“

Das letzte Register „Geschätzte häufigste Werte“ enthält die am häufigsten auftretenden Werte für die entsprechende Spalte.



Geschätzte häufigste Werte

Gemäß den Statistikdaten existiert der Spaltenwert „1360700“ in 6.912 Zeilen der Tabelle „HIST00“. Tatsächlich gibt es aber nur 1.337 Zeilen, die diesen Wert enthalten, wie das nachfolgende SQL Statement schnell beweist:

```
Select count(History_kdnr) as "Häufigkeit"
from itpcr.hist00
where history_kdnr = '1360700';
```

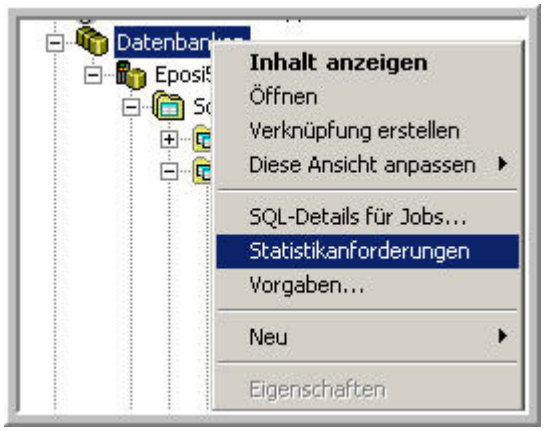
00001
1337

Häufigkeitsverteilung des Wertes ’1360700’

7.4.1
Seite 34

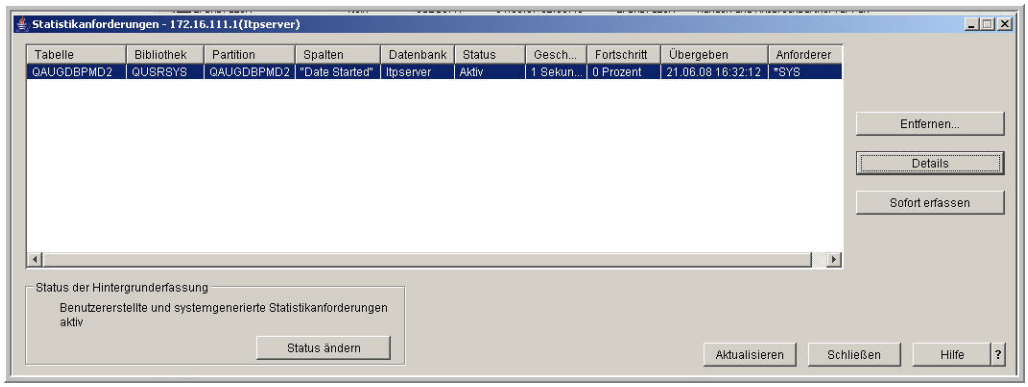
Ihre Statistikdaten sind aufgrund der Abweichungen nicht fehlerhaft und müssen auch nicht neu erstellt werden. Vielmehr basieren auch diese Daten nur auf einer Stichprobe der Zeilen in der Tabelle und geben repräsentative Werte an, nicht die tatsächlichen Zahlen. Es könnte sogar sein, dass die Liste der „Häufigsten Werte“ keine Daten enthält, dann enthielt die Tabelle zum Erfassungszeitpunkt möglicherweise keine oder nur eindeutige Werte. Beides lässt sich schnell anhand der Metadaten prüfen. Überprüfen Sie einfach die Felder „Anzahl Zeilen bei der Erfassung“ und „Schätzung der Kardinalität“.

Wollen Sie prüfen, ob derzeit Statistikanforderungen aktiv sind? Öffnen Sie hierfür den Dialog „Statistikanforderungen“ im System i Navigator. Öffnen Sie diesen Dialog, indem Sie mit der rechten Maustaste das Kontextmenü der „Datenbank“ öffnen.



Der Dialog „Statistikanforderungen“

In dem nun folgenden Dialog sehen Sie sowohl benutzererstellte als auch systemgenerierte Statistikerfassungen.



Statistikanforderungen bearbeiten

Hier können Sie die Statistikerfassungen stoppen, Fehler korrigieren, die sofortige Ausführung veranlassen und vieles mehr.

Wofür brauchen wir eigentlich die Statistikdaten? Meine Abfragen liefen auch ohne Statistiken! Und mal ehrlich, wer schaut sich jeden neuen Menüpunkt im System i Navigator nach einem Releasewechsel an? Manch einer hat die Einführung der Statistikdaten gar nicht bemerkt! Die SQE nutzt die Statistiken während des Optimierungsprozesses, um Informationen über die Tabelle und deren Indizes zu erhalten. „Wie viele Zeilen enthält die Tabelle?“ „Welche Indizes sind für die SQL Abfrage relevant?“ Aber auch Informationen über die Datenlage und die erforderliche Anzahl I/O Operationen werden abgefragt: „Wie viele Zeilen entsprechen den Selektionskriterien?“ „Wie viele I/O Operationen sind für den Datenabruf nötig?“ „Können parallele Prozesse aufgebaut werden?“ Der Statistik Manager muss die Anfragen der SQE beantworten. Er kann hierfür die selbst erstellten Spaltenstatistiken nutzen oder eine Schlüsselanfrage über einen geeigneten Index (Key Estimate Range) ausführen, um die bestmöglichen Antworten zu liefern. Stehen beide Möglichkeiten nicht zur Verfügung, wird die Anfrage mit Defaultwerten beantwortet. Diese Defaultwerte können mit der realen Datenverteilung übereinstimmen, sie können aber auch erheblich davon abweichen. Da die SQE den Informationen des Statistikmanagers immer vertraut und auf dessen Basis den Zugriffsplan erstellt, ist darauf zu achten, dass die Spaltenstatistiken korrekt und vollständig sind. Anders als auf vielen anderen Datenbanksystemen ist dieser Prozess für die DB2 aber weitestgehend automatisiert, denn nachdem der Statistik Manager eine Anfrage mit Standardwerten beantworten musste, werden die Statistiken umgehend automatisch erhoben und stehen beim nächsten Aufruf bereit. Statistikdaten werden also immer dann erstellt, wenn die Anfragen der SQE mit Defaultwerten beantwortet wurden. Falls Systemeinstellungen die sofortige Erhebung verhindern, wird die Anforderung im Plan Cache hinterlegt und ausgeführt, sobald die Systemwerteinstellung dies zulässt. Selbst wenn die SQL-Abfrage vorzeitig abgebrochen wird, wird der Statistikmanager fortfahren, die Daten zu erheben, um bei einer erneuten Anfrage die Informationen liefern zu können.

Ergänzung 13./2010

Allerdings werden die Statistikdaten nicht permanent aktualisiert, wenn Sie neue Datensätze in die Tabellen einfügen oder bestehende Sätze verändern. Lediglich die Spalte „aktuelle Einfügungen, Aktualisierungen und Löschungen" verändert sich. Eine Überprüfung der statistischen Werte erfolgt erst, wenn

- ein ODP für ein SQL Statement erstellt wird, das auf Grundlage der Statistiken optimiert wurde
- ein neuer Ausführungsplan erstellt wird
- oder die Tabellenstruktur sich ändert.

Wird bei der Überprüfung anhand der Metadaten festgestellt, dass die Tabelle ca. 15 % neue und/oder veränderte Datensätze enthält, werden die Statistiken als veraltet gekennzeichnet und neu erhoben. Die aktuelle Abfrage wird zwar in diesem Fall noch mit veralteten Statistiken ausgeführt, aber für zukünftige Ausführungen stehen die neuen Statistikwerte bereit. Der Erstellungs- und Aktualisierungsprozess ist also vollständig automatisiert, so dass Benutzereingriffe eigentlich überflüssig werden. Allerdings gibt es einige wenige Ausnahmen, die eine manuelle Erfassung rechtfertigen könnten:

- Wenn Sie ein Backup System betreiben, müssen Sie beachten, dass das Backup-System nicht über die aktuellen Statistikdaten verfügt. Wenn Sie also keine Zeitverzögerungen in Kauf nehmen wollen, sobald Ihr Backup-System zum Einsatz kommt, müssen Sie auf der Backupmaschine die Statistikdaten manuell erstellen.
- Denken Sie daran, dass Statistikdaten bis zum Release V5R4 nicht journalisiert werden. Dies ändert sich erst mit Release V6R1. Ein Wiederherstellen der Statistikdaten anhand der Journale ist somit nicht möglich.
- Wenn Sie Ihren Kunden Programme liefern, müssen Sie berücksichtigen, dass auf der Zielmaschine die bei Ihnen vorhandenen Statistikinformationen fehlen und erst nach und nach aufgebaut werden. Auch hier kann die Anlaufphase verhindert werden, wenn Sie z. B. ein Skript mitliefern, das die benötigten Statistikinformationen aufbaut.
- In vielen Unternehmen werden Massendaten über Nacht eingespielt. Da der Statistik Manager aber erst im Zusammenhang mit einer Query-Anfrage die Aktualität der Statistiken überprüft, werden die ersten Anfragen immer mit veralteten Statistikdaten arbeiten, was sich negativ auf die Performance auswirken könnte. Daher sollten Sie auch in diesem Fall in Betracht ziehen, die Statistikdaten manuell zu aktualisieren, nachdem Sie die Daten eingespielt haben. Sie können die

Aktualisierung am einfachsten erreichen, wenn Sie vor dem Einspielen den Systemwert QDBFSTCCOL auf *NONE setzen und anschließend wieder auf *ALL oder *System zurücksetzen. Ein entsprechender Hintergrundjob wird daraufhin alle Informationen des Plan Cache abarbeiten und auch die Aktualisierung der entsprechenden Statistiken veranlassen. Achten Sie aber darauf, dass Sie den Systemwert zurücksetzen und die Statistiken aktualisieren, bevor Sie ein IPL veranlassen, denn durch das IPL würden alle Informationen im Plan Cache verloren gehen. Außerdem dürfen Sie die Tabellen während der Datenübertragung nicht löschen. Auch dadurch gehen die Informationen verloren. Nutzen Sie also lieber den SQL Befehl DELETE oder den i5/OS-Befehl „Clear Physical File Member“ (CLRPFM) anstelle von DROP oder „Delete File“ (DLTF).

- Dieses Verfahren gilt natürlich auch, falls Sie eine beträchtliche Anzahl Zeilen aktualisieren oder löschen. Finden diese Änderungen dann auch noch während der Arbeitszeit statt, ist zu beachten, dass die Aktualisierung der Statistikinformationen ggf. zu früh erfolgt. Auch in diesen Fällen ist es sinnvoll, die automatische Aktualisierung zunächst zu stoppen und erst nach dem gesamten Update wieder zu initiieren.
- Falls Sie die Struktur Ihrer Datenbank verändern, z. B. neue Spalten hinzufügen, könnte es ebenfalls sinnvoll sein, die entsprechenden Statistikdaten manuell zu erstellen, damit die Statistiken verfügbar sind bevor die ersten Abfragen gestartet werden.
- Wenn der Statistik Manager sich die benötigten Informationen anhand eines Index beschaffen kann, werden ebenfalls keine Statistikdaten erhoben. Da aber die Spalteninformationen der Statistikdaten sehr viel effektiver für die Beantwortung der Optimizeranfragen sind, sollten Sie in diesen Fällen die Statistikdaten manuell erheben.
- Und ein letzter Hinweis: Der Befehl CPYF kopiert keine Statistikinformationen. Daher sollten Sie Ihre Tabellen – wenn möglich – mit dem i/5 OS-Befehl „Create Duplicate Object“ (CRTDUPOBJ) DATA(*YES) duplizieren. Andernfalls müssten Sie auch hier manuell eingreifen und die Statistiken selbst erzeugen. Sollten Sie die Statistikdaten nicht unmittelbar benötigen, werden sie natürlich wieder automatisch erzeugt, sobald eine SQL Abfrage ausgeführt wird.

Ergänzung 13/2010

7.4.1
Seite 38

IBM hat im Übrigen darauf verzichtet entsprechende 5250-Befehle für die Bearbeitung der Statistiken zu implementieren. Allerdings stehen entsprechende APIs zur Verfügung:

API	Funktion
QDBSTDS, QdbstDeleteStatistics	Statistikdaten löschen
QDBSTLS, QdbstListStatistics	Statistiken anzeigen
QDBSTUS, QdbstUpdateStatistics	Statistikdaten aktualisieren
QDBSTLS, QdbstRequestStatistics	Statistikdaten erstellen
QDBSTCRS, QdbstCancelRequestedStatistics	aktive Statistiken abbrechen
QDBSTLDS, QdbstListDetailStatistics	Statistikdetails anzeigen
QDBSTLRS, QdbstListRequestedStatistics	Aktive Statistikanforderungen anzeigen

Im Internet finden Sie zudem einige nützliche IBM Tools:

ftp://testcase.boulder.ibm.com/as400/fromibm/dbsttools.savf

Das Savefile „dbsttools“ enthält generierte 5250-Commands und ausführbare CPP-Programme, die die Arbeit mit den Statistiken deutlich erleichtern. Ich weiß, dass die Verwendung von APIs manchmal problematisch ist. In diesem Fall geht es aber aufgrund der verfügbaren Tools recht einfach.

Schauen wir uns ein kleines Beispiel an: Gehen wir einmal davon aus, dass sich meine Entwicklungsumgebung in der Bibliothek „ITPCR“ befindet. Bevor ich meine Programme freigebe, möchte ich die in der Entwicklungsumgebung erstellten Statistikinformationen auch in der Produktionsumgebung erstellen, um Anlaufschwierigkeiten zu vermeiden.

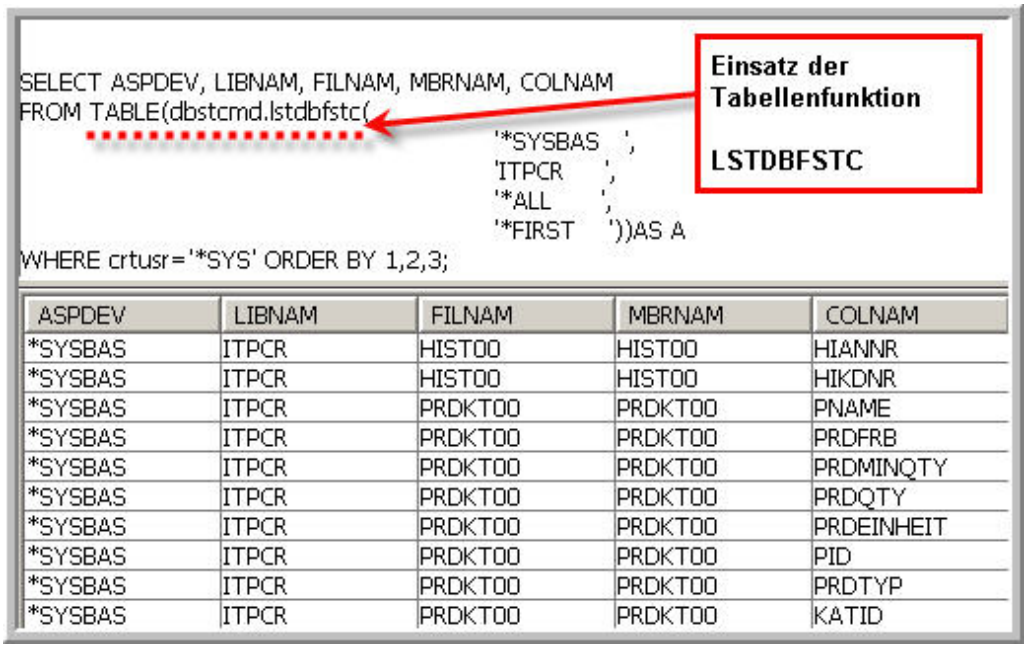
Zunächst einmal muss ich herausfinden, welche Statistiken in der Bibliothek „ITPCR“ vom System während der Testphase erstellt wurden. Dazu nutze ich das folgende SQL Statement:

```
SELECT ASPDEV, LIBNAM, FILNAM, MBRNAM, COLNAM
FROM TABLE(dbstcmd.lstdbfstc(
                     '*SYSBAS ',
                     'ITPCR ',
                     '*ALL ',
                     '*FIRST ')) AS A
WHERE CRTUSR = '*SYS'
ORDER BY 1, 2, 3;
```

Voraussetzung für dieses SQL Statement sind zwei Dinge:

- Sie müssen die User Defined Table Function (UDTF) „LSTDBFSTC“ erstellen
- und Sie benötigen das Serviceprogramm „LSTDBFSTC“. Das Serviceprogramm liefert IBM als generiertes Runtimeobjekt. Es ist im Savefile „dbsttools“ enthalten.

Wenn Sie die Tabellenfunktion nutzen, müssen Sie die Argumente als 10 Byte lange Zeichenwerte übergeben und dafür ggf. mit Blanks auffüllen. Auch müssen Sie darauf achten, dass Sie Großbuchstaben bei der Eingabe verwenden, andernfalls erhalten Sie entsprechende Fehlermeldungen. Wenn Sie die Funktion korrekt aufrufen, erhalten Sie eine Liste der vorhandenen Spaltenstatistiken in der Bibliothek „ITPCR“.



ASPDEV	LIBNAM	FILNAM	MBRNAM	COLNAM
*SYSBAS	ITPCR	HIST00	HIST00	HIANNR
*SYSBAS	ITPCR	HIST00	HIST00	HIKDNR
*SYSBAS	ITPCR	PRDKT00	PRDKT00	PNAME
*SYSBAS	ITPCR	PRDKT00	PRDKT00	PRDFRB
*SYSBAS	ITPCR	PRDKT00	PRDKT00	PRDMINQTY
*SYSBAS	ITPCR	PRDKT00	PRDKT00	PRDQTY
*SYSBAS	ITPCR	PRDKT00	PRDKT00	PRDEINHEIT
*SYSBAS	ITPCR	PRDKT00	PRDKT00	PID
*SYSBAS	ITPCR	PRDKT00	PRDKT00	PRDTYP
*SYSBAS	ITPCR	PRDKT00	PRDKT00	KATID

Vorhandene Statistiken anzeigen

7.4.1
Seite 40

Die UDTF „dbstcmd“ kann ich anschließend in weiteren SQL Prozeduren nutzen, um die Informationen z. B. in eine Datei zu schreiben.

```
CALL ITPCR.STCRTTAB('ITPCR');
```

Inhalt von ITPCR.CRTTAB - 172.16.110.

	TABNAM	MBRNAM	COLNAM
1	HIST00	HIST00	HIANNR
2	HIST00	HIST00	HIKDNR
3	PRDKT00	PRDKT00	KATID
4	PRDKT00	PRDKT00	PID
5	PRDKT00	PRDKT00	PNAME

Die Ausgabedatei CRTTAB

Die Dateieinträge verwende ich z. B. anschließend in einem CL-Programm, um die benötigten Statistiken automatisch in der Produktionsumgebung zu erstellen:

```
        DCLF    FILE(ITPCR/CRTTAB)
LOOP:   RCVF
        MONMSG  MSGID(CPF0864)  EXEC(RETURN)        Statistikdaten löschen

        ITPCR/DLTDBFSTC FILE(EPSW08CR/TABNAM)  MBR(&MBRNAM)

                                        neue Statistiken erstellen

        ITPCR/CRTDBFSTC FILE(EPSW08CR/TABNAM)  MBR(&MBRNAM)  +
                        COLLMODE(*BACKGROUND)  STCCOLL((&COLNAM *GEN))
        GOTO LOOP
```

Das CL Programm CRT_STAT

Im CL-Programm setze ich weitere IBM Commands ein, die ebenfalls im erwähnten Savefile als generierte RunTime Objekte vorhanden sind:

- DLTDBFSTC (Statistikdaten löschen) und
- CRTDBFSTC (Statistikdaten erstellen).

Sicherlich lässt sich diese Aufgabenstellung weiter optimieren. Die Beispiele sollen nur einen kleinen Überblick geben und erste Ideen liefern. Weiterführende Informationen zu den benutzten APIs finden Sie im Übrigen im Internet unter:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp

Spätestens zu diesem Zeitpunkt stellt man sich die Frage, ob Statistiken nicht vollständig zu Gunsten geeigneter Indizes vernachlässigt werden können. Wo liegt der Unterschied? Statistikdaten werden schließlich wie Indizes verwendet, um Anfragen des Optimizers zu beantworten:

- Ein entscheidender Unterschied ist darin zu sehen, dass ein Index ein permanentes Objekt ist, das aktualisiert werden muss, sobald Daten in der entsprechenden Tabelle verändert werden. Dies ist bei den Statistikdaten nicht der Fall. Natürlich veranlasst eine Veränderung der Tabellen auch eine Aktualisierung der Statistikdaten, aber dieser Hintergrundjob verlangt weit weniger Performance als die Aktualisierung des Index. Ich habe mittlerweile Anwendungen gesehen, die über zwei Minuten benötigten, um einen Datensatz einzufügen!
- Der Speicherbedarf für die Statistikdaten ist erheblich geringer; es werden nur ca. 12 k für eine Spaltenstatistik benötigt unabhängig von der Tabellengröße.
- Der Zugriff auf die Statistikdaten ist schneller und effektiver als die Nutzung der Indizes.

Da drängt sich schnell die nächste Frage auf: „Vielleicht können Indizes durch Spaltenstatistiken abgelöst werden?“ Einige Indizes sicherlich, aber

- nur die SQE kann die Spaltenstatistikdaten verwenden. Die CQE verwendet ausschließlich die Informationen aus den Indizes.
- Letztendlich können Statistikdaten nur während der Optimierungsphase verwendet werden. Für die Ausführung der Datenanfrage sind sie unbrauchbar. Indizes werden also nicht überflüssig, es sei denn, die Indizes wurden bisher nur erstellt, um Statistikwerte zu liefern.
- Wenn der Statistikmanager eine Anfrage beantwortet, die Selektionskriterien enthält, wird der Optimizer folgende Vorgehensweise bevorzugen:
  Enthält die Selektionsbedingung die Schlüsselwörter AND und/oder OR versucht der Statistik Manager einen Index zu finden, der die zusammengesetzten Kriterien enthält, um entsprechende Informationen zu liefern. Kann kein geeigneter Index genutzt werden, wird der Statistik Manager einzelne Indizes verwenden, um die Anfrage zu beantworten.
  Handelt es sich dagegen um die Abfrage einer einzelnen Spalte, verwendet der Statistikmanager bevorzugt Spaltenstatistikdaten.

Es geht also nicht um die Frage „Statistikdaten oder Indizes“, sondern vielmehr darum Statistikinformationen und Indizes für die Informationsgewinnung zu nutzen.

Ergänzung 13./2010

7.4.1
Seite 42

7.4.2 Visual Explain

Visual Explain ist ein Interface, das Ausführungspläne der SQL-Abfragen visualisiert. Beide SQL Engines – CQE und SQE – werden unterstützt. Hierbei werden alle notwendigen Verarbeitungsschritte sowie die verwendeten temporären und permanenten Objekte grafisch aufbereitet dargestellt. Verschiedene Icons repräsentieren einzelne Verarbeitungsschritte sowie verwendete Objekte. Zu jedem verwendeten Objekt stehen weiterführende Informationen zur Verfügung. So können Sie u. a. die zur Verfügung stehenden Indizes bestimmen. Sie können prüfen, welche Statistiken für Tabellen verfügbar sind. Sie haben die Möglichkeit, globale Informationen über Satzanzahl und Tabellengröße zu ermitteln, ohne den aktuellen Bildschirm verlassen zu müssen. Zusätzlich stellt das Tool Ihnen vielfältige Informationen über die Laufzeitumgebung Ihrer Abfrage zur Verfügung. Seit Release V5R4 wird in diesen Informationen auch die verwendete Query Engine benannt. So wird es einfacher, die Verarbeitung einer SQL-Abfrage zu verstehen und mögliche Kostenfaktoren zu analysieren.

Visual Explain ist eine Komponente vom System i Navigator; sie kann auf vielfältige Weise aufgerufen werden – beispielsweise für:

- SQL-Anweisungen des Plan Cache,
- SQL-Anweisungen eines Plan Cache Snapshots,
- SQL-Anweisungen eines detaillierten SQL Performance Monitors,
- ausgeführte SQL-Anweisungen eines Jobs,
- interaktiv ausgeführte SQL Statements in der Prozedurumgebung des System i Navigators
- usw.

Ergänzung 7/2008

7.4.2
Seite 2

Wenn Sie Visual Explain aus der Prozedurumgebung vom System i Navigator aus aufrufen, platzieren Sie den Cursor auf ein SQL-Statement, dessen Ausführungsplan angezeigt werden soll. Klicken Sie anschließend auf eines der verfügbaren Icons oder wählen Sie den Menüpunkt „Visual Explain“:

Visual Explain ausführen im System i Navigator V6R1

„Erklären“ (STRG + E)
Diese Funktion erlaubt die Erstellung eines grafischen Ausführungsplans ohne Ausführung der SQL-Abfrage. Intern wird dem Attribut „QUERY_TIME_LIMIT“ der Wert „0“ zugewiesen, um die Ausführung zu verhindern. Ausführungspläne der SQL-Abfragen, die viele Stunden laufen, können in wenigen Sekunden angezeigt werden. Die angezeigten Informationen über die Laufzeitumgebung könnten in diesem Fall aber von tatsächlichen Umgebungseinstellungen abweichen. Zudem könnte es passieren, dass die Ausführungspläne komplexer Abfragen nicht dargestellt werden. Probieren Sie es einfach aus!

„Ausführen und Erklären“ (STRG + U)
Hiermit wird die Abfrage ebenfalls an den Optimizer übertragen. Anschließend wird das Statement ausgeführt; erst nach der Ausführung erfolgt die Darstellung des Ausführungsplans. Visual Explain könnte ein Ergebnis liefern, dass nicht die aktuelle Laufzeitumgebung widerspiegelt. Das passiert z. B., wenn das SQL Statement in Umgebungen ausgeführt wird, die sich schnell und häufig ändern.

„Ausführen während der Ausführung“ (STRG + W)
Die Option „Ausführen während der Ausführung“ wurde genau aus diesem Grund dem Release V5R4 hinzugefügt. Das Ergebnis repräsentiert die aktuelle Ausführung der Abfrage. So sollen auch komplexe Abfragen exakter dargestellt werden. Hinzu kommt, dass Sie während der Analyse die Umgebungsattribute aktualisieren können.

Probieren wir es aus! Ich starte die Prozedurumgebung vom System i Navigator und lasse mir die Ausführung einer SQL-Abfrage erklären. Das ist zumindest die Absicht. Das Ergebnis sieht etwas anders aus:

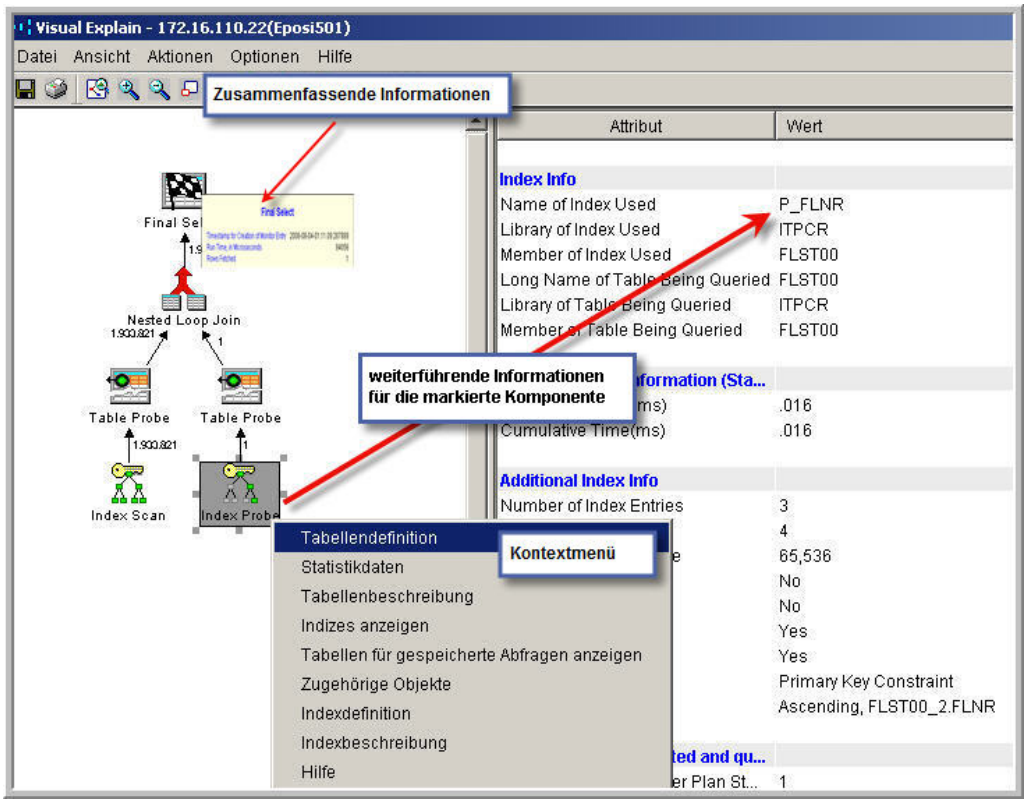
Fehlersituation

Seit V6R1 muss der Cursor für Abfragen nicht mehr vollständig geöffnet und geschlossen werden, wenn mit der Prozedurumgebung gearbeitet wird. Die Ausführung der SQL Statements ist dadurch schneller. Nur Visual Explain verursacht jetzt hin und wieder Ärger. Wenn die Fehlermeldung auch auf Ihrem Desktop erscheint, dann helfen veränderte JDBC-Einstellungen. Öffnen Sie hierfür in der Prozedurumgebung von System i Navigator das Menü „Verbindungen“ und wählen Sie „JDBC-Einstellungen“. Im Register „Leistung“ ändern Sie folgende Einstellung:

Veränderte JDBC-Einstellungen

7.4.2
Seite 4

Jetzt sollte es funktionieren! Das Visual Explain Ergebnisfenster selbst besteht aus mehreren Komponenten. Auf der linken Seite wird der Ausführungsplan grafisch aufbereitet dargestellt. Jede grafische Ausführungskomponente benennt die verwendete Implementierungsmethode. Anhand der Pfeilbeschriftungen erkennen Sie die Anzahl verarbeiteter Tabellenzeilen.



Visual-Explain-Ergebnisanzeige

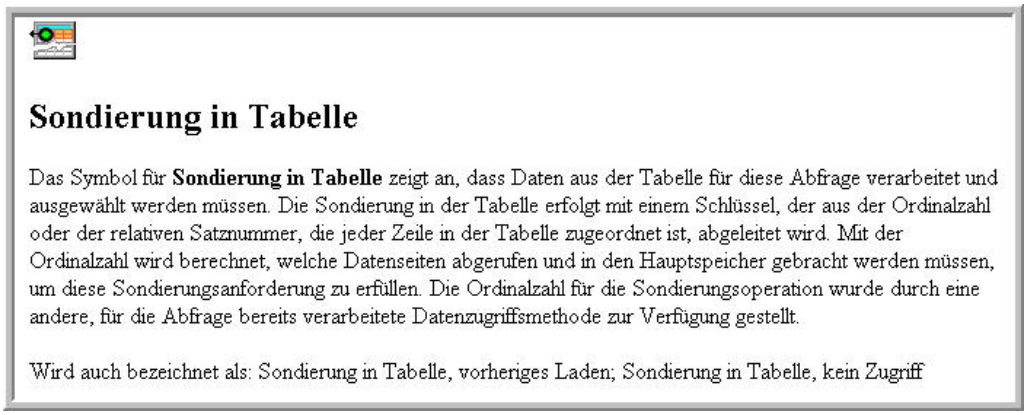
Wenn Sie den Mauszeiger auf eine der Komponenten bewegen, dann erhalten Sie zusammenfassende Informationen über den Ausführungsschritt. Die zusammenfassenden Informationen beinhalten Laufzeiten sowie die Anzahl selektierter Sätze, soweit sie verfügbar sind.

Wenn Sie eine Ausführungskomponente im linken Fensterbereich markieren, ändern sich die Informationen des rechten Fensterbereichs. Die weiterführenden Informationen korrespondieren somit stets mit der markierten Komponente des Plans.

Für jede Komponente existiert zudem ein Kontextmenü. Über dieses Kontextmenü stehen für permanente Objekte folgende Funktionen zur Verfügung:

- Tabellendefinition und Tabelleneigenschaften anzeigen
- Verfügbare Statistikdaten und Indizes anzeigen
- Tabellen für gespeicherte Abfragen (MQTs) anzeigen
- Abhängige Objekte anzeigen
- Indexdefinitionen und Indexeigenschaften anzeigen
- Hilfefunktion

Viele dieser Aktionen erkennen Sie beim Ausprobieren schnell wieder. Erwähnenswert in diesem Zusammenhang ist die „Hilfefunktion“. Für jedes Icon im Ausführungsplan erhalten Sie detaillierte Erklärungen:



Erläuterungen der grafischen Symbole

Der untere Teil des Ergebnisfensters besteht aus zwei Indexzungen, die einerseits das ausgeführte SQL Statement und andererseits automatisch die zugehörigen Debug-Informationen anzeigen. Die Nachrichten des Abfrageoptimierungsprogramms sind allerdings nur verfügbar, wenn die Erklärung der Abfrage aus dem Skript-Fenster „SQL-Prozeduren ausführen“ angefordert wurde. Bei diesen Nachrichten handelt es sich um eine Teilauflistung aller Debug-Nachrichten, die im Jobprotokoll enthalten sind. In Version V6R1 entfällt die Anzeige der Debug-Nachrichten aus den bereits erläuterten Gründen.

7.4.2.1 Toolbar und Menüoptionen

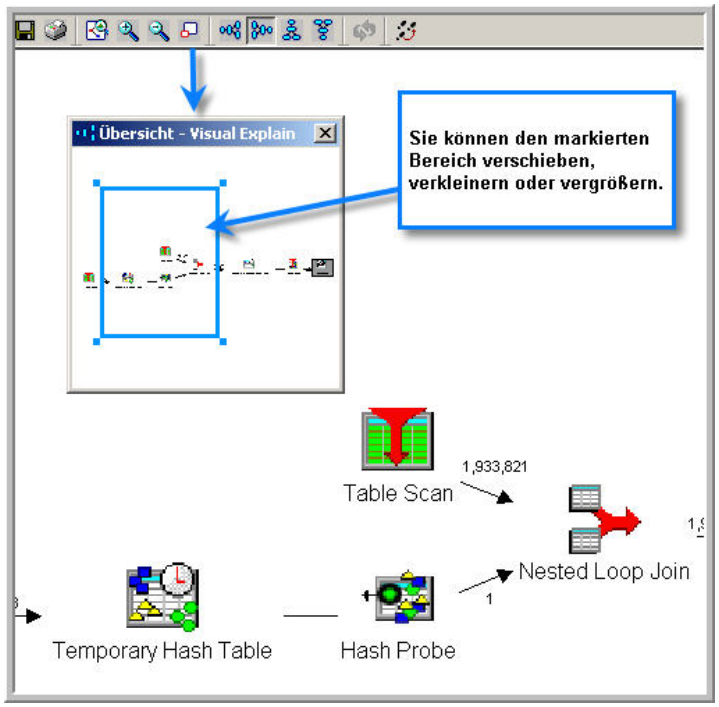
Die Toolbar im oberen Fensterbereich hilft Ihnen, in Visual Explain zu navigieren:

Toolbar

Schauen wir uns die Wirkungsweise der verschiedenen Schaltflächen an:

Das erste Icon ermöglicht die Sicherung des Ausführungsplanes als detaillierten SQL-Performance-Monitor in einem beliebigen Schema. Nach dem Speichern sind diese Daten als „SQL-Performance-Monitor" weiterhin verfügbar. Dadurch ist es jederzeit möglich, zwei Ausführungspläne miteinander zu vergleichen, die zu unterschiedlichen Zeitpunkten oder mit unterschiedlichen Umgebungseinstellungen erstellt wurden. Das nächste Icon erlaubt den Ausdruck des Ausführungsplanes und muss nicht weiter kommentiert werden. Mit den folgenden Schaltflächen können Sie die Zoomstufe des Abfragediagramms ändern, um die Größe des Diagramms zu steuern, das innerhalb des linken Teilfensters sichtbar sein soll.

Das sechste Icon von links erstellt zunächst ein separates Übersichtsfenster:



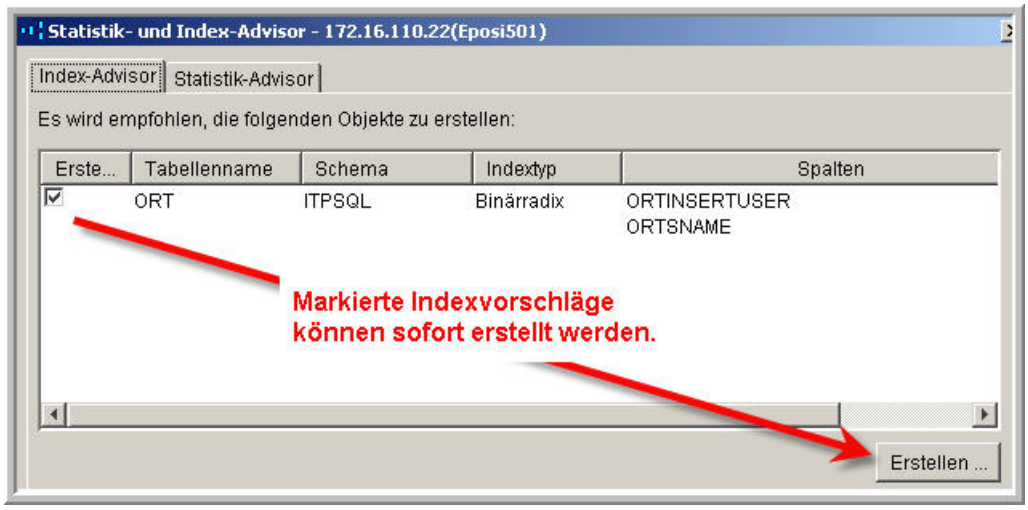
Ausschnittvergrößerung

Anschließend können Sie im Übersichtsfenster einen beliebigen Bereich des Ausführungsplans markieren und verschieben. Der markierte Bereich des Übersichtsfensters wird entsprechend vergrößert im Hauptfenster dargestellt. Dies ermöglicht auf einfache Art eine Teilansicht sehr komplexer Ausführungspläne.

Die vier blauen Icons der Toolbar bestimmen die Ausrichtung des Ausführungsplans. Das Endergebnis kann links, rechts, oben oder unten angezeigt werden. Probieren Sie aus, welche Darstellungsweise Sie bevorzugen.

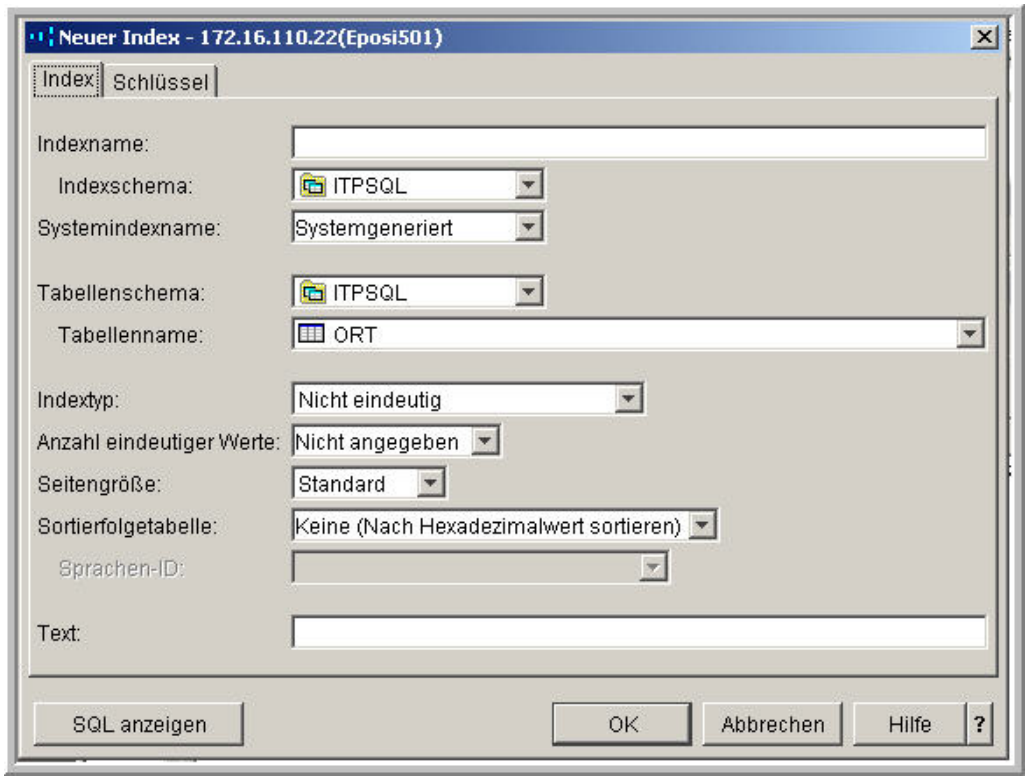
Das vorletzte Icon aktualisiert die Laufzeitumgebungsinformationen. Es ist allerdings nur verfügbar, wenn Sie die Erklärung der Abfrage während der Ausführung (STRG + W) angefordert haben.

Das letzte Icon öffnet den Visual Explain Statistik- und Index-Advisor. Hier finden Sie Indexvorschläge, die sich positiv auf die Ausführung dieser SQL-Abfrage auswirken könnten.



Statistik- und Index-Advisor

Sie können einzelne Indexvorschläge markieren und sofort erstellen. Daraufhin öffnet sich für jeden Index, der erstellt werden soll, ein weiterer Dialog:

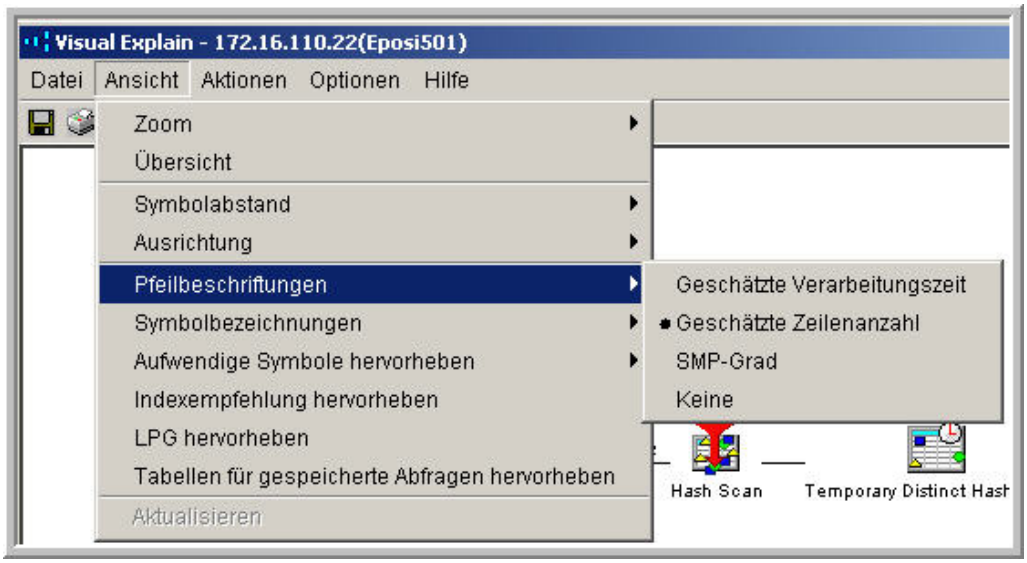


Indizes mit Visual Explain erstellen

Ergänzen müssen Sie lediglich den Indexnamen. Seien Sie aber vorsichtig mit der Erstellung! Der Vorschlag berücksichtigt keine bestehenden Indizes; er bezieht sich nur auf die soeben ausgeführte Abfrage und das aktuelle Datenvolumen. Ob der Index anschließend benutzt wird, ist nicht garantiert. Sie können auf diesem Weg sehr schnell viele unnütze Indizes erstellen und Ihr System belasten, ohne Ihre SQL-Abfragen zu beschleunigen.

Oberhalb der Toolbar befinden sich in Visual Explain diverse Menüoptionen: „Datei“, „Ansicht“, „Aktionen“, „Optionen“ und „Hilfe“.

Das Menü „Ansicht“
enthält viele Möglichkeiten, die grafische Darstellung zu verändern und entspricht in weiten Teilen den Funktionen der Toolbar.



Menü „Ansicht“

Zusätzlich können Sie Pfeilbeschriftungen verändern und Ausführungssymbolen Objektnamen hinzufügen. Aufwendige Verarbeitungsschritte – hohe Satzanzahl und/oder lange Verarbeitungszeiten – können hervorgehoben werden. Das gleiche gilt für Indexempfehlungen.

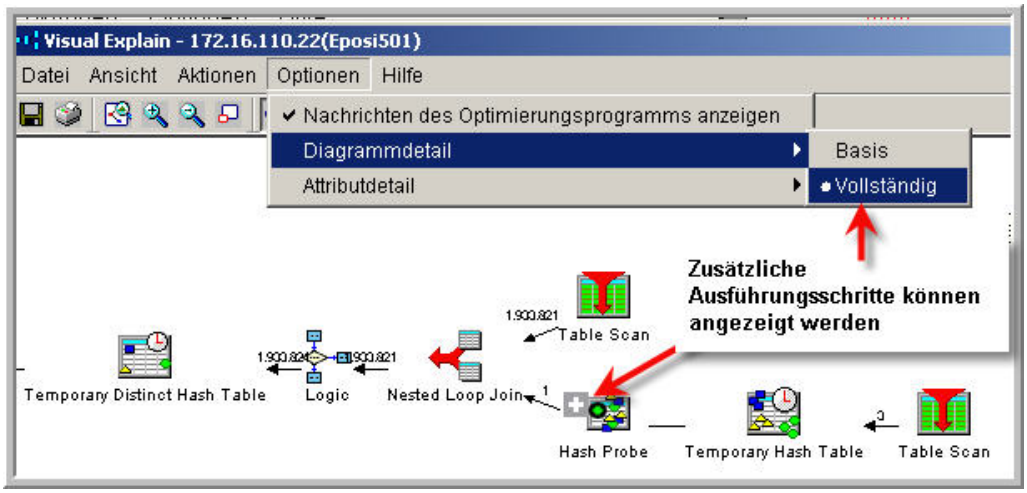
Eine kurze Erklärung ist für die Option „LPG hervorheben“ nötig. LPG ist die Abkürzung für „Look Ahead Predicate Generation“ und eine besondere Art der Verarbeitung für „Join“-Abfragen. Nutzt der Optimizer die Methode „LPG“, dann versucht er, den I/O-Aufwand einer Verknüpfung zu minimieren, indem er die lokalen Selektionskriterien großer Tabellen vorab ausführt und in temporären Objekten zwischenspeichert. LPG wird vorrangig für „Star-Join“-Verknüpfungen ausgeführt. Sie können angeben, dass alle Symbole hervorgehoben werden, die die Methode verwenden.

„Tabellen für gespeicherte Abfragen“ sind „Materialized Query Tables“(MQTs). MQTs sind spezifische Tabellen, die neben den Daten auch die Abfrage dieser Daten speichern. Seit V5R4 können derartige Tabellen im Optimierungsprozess verwendet werden. Visual Explain erlaubt die Hervorhebung des Ausführungsschrittes, der MQTs verwendet.

Das Menü „Aktionen“
entspricht dem Kontextmenü einer Ausführungskomponente und muss nicht weiter erläutert werden.

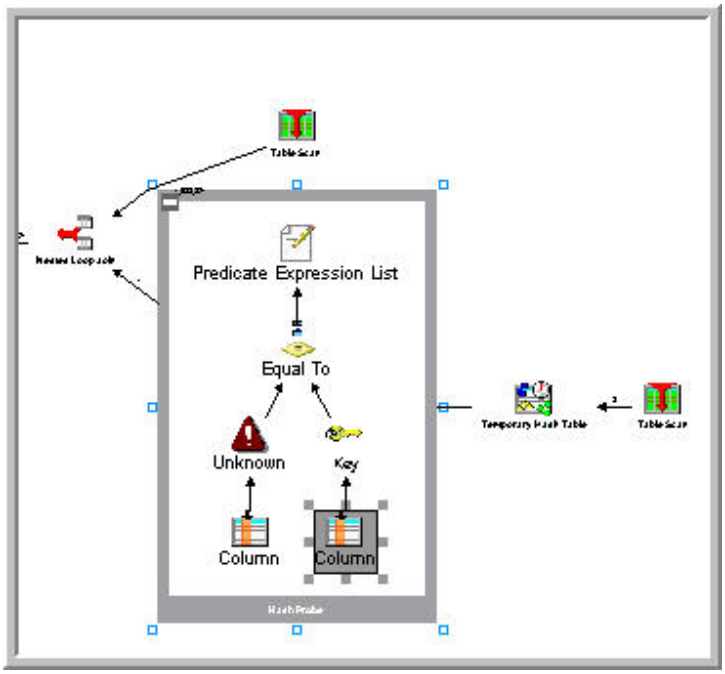
7.4.2
Seite 10

Das Menü „Optionen“
enthält Einstellungen, die es ermöglichen den Ausführungsplan detaillierter darzustellen. Die Option „Vollständig“ wird gewöhnlich von IBM Entwicklern in Rochester verwendet, um detailliert zu prüfen, welche Verarbeitungsschritte ausgeführt wurden. Wenn Sie diese Option aktivieren, dann werden einige Ausführungssymbole mit einem Pluszeichen markiert – wie die folgende Grafik zeigt.



Menü „Vollständige Optionen“

Über das Kontextmenü der Komponente wählen Sie anschließend die Aktion „Erweitern“. Angezeigt wird jetzt die Einzelverarbeitung dieses Ausführungsschrittes:



Wirkung der Option „Vollständig“

7.4.2.2 Visual Explain Abfrageattribute und Werte

Die Abfrageattribute im rechten Bildschirmbereich ergänzen die grafische Darstellung des Ausführungsplans. Wenn Sie ein Icon markieren, erhalten Sie nützliche Informationen über den jeweiligen Ausführungsschritt. Anhand einiger Beispiele, die in keinem logischen Zusammenhang stehen und sich nicht immer auf den eben dargestellten Ausführungsplan beziehen, zeige ich Ihnen einige der verfügbaren Informationen:

Geschätzte Ausführungszeit und Tabelleninformationen

Estimated Time Information (Start Time, Total Time)	
Processing Time(ms)	14,600
Cumulative Time(ms)	14,600
Additional Table Info	
Total Rows in Table	1,933,821
Table Size(bytes)	226,224,396
Active Table Rows	1,933,821
Deleted Table Rows	16,330

Geschätzte Ausführungszeit und Tabelleninformationen

Für jeden Ausführungsschritt kalkuliert der Optimizer die Ausführungszeit und kumuliert die Werte. Zusätzliche Tabelleninformationen geben Auskunft über die Datenlage. Die Informationen werden in meinem Beispiel für die Methode „Table Scan“ der Tabelle „HIST00“ angezeigt. Die Tabelle hat fast zwei Millionen Sätze. Das Verlagern der benötigten Sätze in den Hauptspeicher dauerte 14,6 Millisekunden. Da dies der erste Ausführungsschritt im Ausführungsplan ist, existieren keine kumulierten Werte.

7.4.2
Seite 12

Geschätzte Satzanzahl und „Join"-Informationen

Estimated rows selected and qu...	
Total Selected Row Count	1,933,821
Total Rows Processed	1,933,821
Optimize for N Rows	All
Plan Step Iterations	1
Percent Selectivity	100
Cumulative Percent Selectivity	100
Fetch N Rows	All

Geschätzte Satzanzahl und „Join"-Informationen

Die Anzahl der selektierten Zeilen zeigt die vom Optimizer erwartete Satzanzahl für diesen Verarbeitungsschritt an. Wenn die Abfrage nur erklärt wird, dann handelt es sich hierbei um einen Schätzwert; andernfalls wird die tatsächlich verarbeitete Satzanzahl dargestellt. In meinem Beispiel wurde also die gesamte Tabelle in den Arbeitsspeicher verlagert. Erkennbar ist auch das Optimierungsziel des einzelnen Ausführungsschritts. Es kann vom globalen Optimierungsziel, dass Sie für die Abfrage definiert haben, durchaus abweichen.

Geschätzte Ausführungskosten

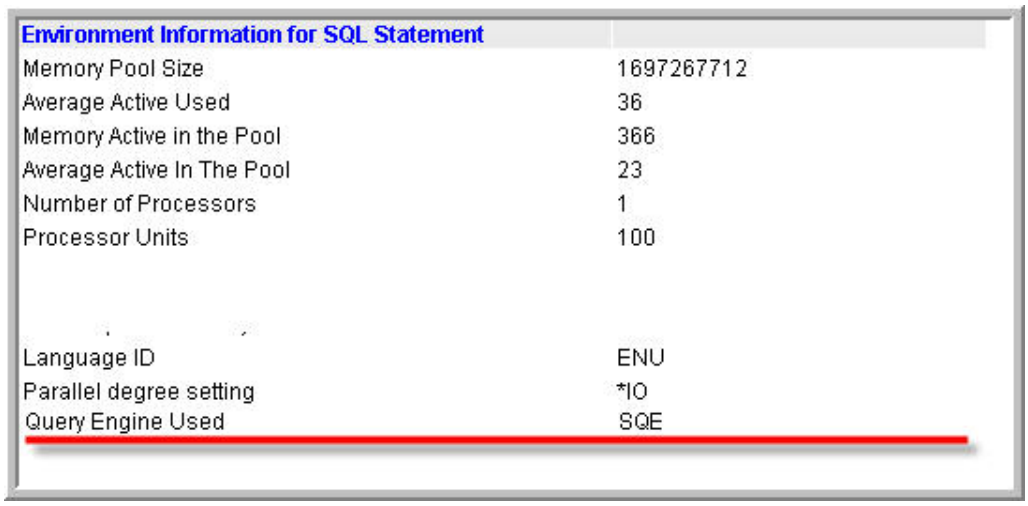
Estimated Cost Information About the Plan Performed	
Processing Time(ms)	14,600
I/O Or CPU Bound	I/O Bound
CPU Cost(ms)	307.699
I/O Cost(ms)	14,600
I/O Count	1,752
PreLoad Relation	No
Memory Used(bytes)	3,617,284
Share of Memory Available(bytes)	46,599,600
Memory Constrained	No
Cumulative Memory Constrained	No

Geschätzte Verarbeitungskosten

Der Abschnitt zeigt, die geschätzte Ausführungszeit des Verarbeitungsschritts in Millisekunden. Abfragen können CPU- oder I/O-intensiv sein. Abhängig davon, welche Ressourcen dieser Schritt vorwiegend beansprucht, nennt man dieses Verhalten „I/O Bound" oder „CPU Bound". Wenn mit dem Verarbeitungsschritt Daten von der Festplatte in den Arbeitsspeicher übertragen werden, dann können Sie erkennen, wie viele I/O-Operationen und wie viel Arbeitsspeicher benötigt wurde. Hier lässt sich schnell feststellen, ob Pools zu klein definiert wurden oder der verfügbare Arbeitsspeicher nicht ausreichend ist.

Globale Informationen

Die Informationen der einzelnen Ausführungsschritte werden sinnvoll durch globale Informationen ergänzt. Hierfür markieren Sie das Icon „Final Select“. Dargestellt werden die verwendeten Attribute der Optionsdatei „QAQQINI“ und die Attribute der Laufzeitumgebung.



Environment Information for SQL Statement	
Memory Pool Size	1697267712
Average Active Used	36
Memory Active in the Pool	366
Average Active In The Pool	23
Number of Processors	1
Processor Units	100
Language ID	ENU
Parallel degree setting	*IO
Query Engine Used	SQE

Allgemeine Informationen über die Laufzeitumgebung

In den globalen Informationen finden Sie auch Hinweise zur verwendeten SQL Engine. Falls die CQE für die Ausführung verantwortlich ist, werden die Gründe hierfür genannt.

7.4.2
Seite 14

Globale Laufzeitinformationen

Verwirrend in der Darstellung sind immer wieder die globalen Laufzeitinformationen. Wenn Sie sich ein erstes Mal die Ausführung erklären lassen, sind die Werte eindeutig zu interpretieren.

Time Information	
Timestamp for Creation of Monit...	2008-08-04-13.34.34.124902
Statement Start Timestamp	2008-08-04-13.34.34.115509
Statement End Timestamp	2008-08-04-13.34.34.408953
Optimization Time, in Milliseconds	1450
Run Time, in Microseconds	10072
Statement Open Time, in Micros...	9384
Statement Fetch Time, in Micros...	168
Statement Close Time, in Micros...	520
Total Estimated Runtime(ms)	16,200

Globale Laufzeitinformationen

Der Optimizer schätzte für die Ausführung meines Statements „16200“ Mikrosekunden oder „16,2“ Millisekunden. Tatsächlich benötigt diese Abfrage aber „10,072“ Millisekunden. Das Öffnen des Datenpfades nahm die meiste Zeit nämlich „9,384“ Millisekunden in Anspruch. Verändere ich jetzt die Einstellungen meiner Prozedurumgebung, ohne die Prozedurumgebung zuvor zu schließen, und führe das SQL Statement ein zweites Mal aus, passiert Folgendes:

Time Information	
Timestamp for Creation of Monit...	2008-08-04-13.36.28.469480
Statement Start Timestamp	2008-08-04-13.36.28.428713
Statement End Timestamp	2008-08-04-13.36.29.063340
Optimization Time, in Milliseconds	1450
Run Time, in Microseconds	41080
Statement Open Time, in Micros...	40752
Statement Fetch Time, in Micros...	152
Statement Close Time, in Micros...	176
Total Estimated Runtime(ms)	16,200

Auswirkung veränderter Umgebungseinstellungen

Die Werte „Run Time“ und „Statement Open Time“ werden kumuliert und liefern unsinnige Werte. Hier hilft im Moment nur eines: Schließen Sie die Prozedurumgebung und erstellen Sie den Ausführungsplan neu.

7.4.3 SQL Performance Monitor

Wenn Sie ein SQL Performance-Problem haben, aber die Ursache dafür nicht lokalisieren können, kann eine Möglichkeit der Analyse darin bestehen, Daten über die Ausführung der SQL-Abfragen zu sammeln. Diese Daten können Sie mit Hilfe eines SQL Performance Monitors erstellen. Grundsätzlich lassen sich zwei Monitortypen unterscheiden: Zusammenfassende und detaillierte Monitore. In englischer Sprache nennt man die Zusammenfassungen „Summary SQL Performance Monitor“ oder „Memory Resident Database Monitor“. „Detailed SQL Performance Monitor“ ist die englische Bezeichnung für „Detaillierte Datenbankmonitore“ und „Detaillierte SQL Performance Monitore“. Zwischen den verschiedenen Begriffen gibt es keine Unterschiede. Häufig werden nur aufgrund des verwendeten Interfaces sprachliche Unterschiede gemacht. Im System i Navigator spricht man von einem „Detaillierten SQL Performance Monitor“. Im 5250-Umfeld verwendet man eher die Bezeichnung „Detaillierter Datenbankmonitor“. Ich spreche im Folgenden stets von einem „Detailmonitor“ (Detailed SQL Performance Monitor) und einer „Monitorzusammenfassung“ (Summary SQL Performance Monitor).

Es gibt viele verschiedene Möglichkeiten, Daten für einen Monitor zu erstellen:

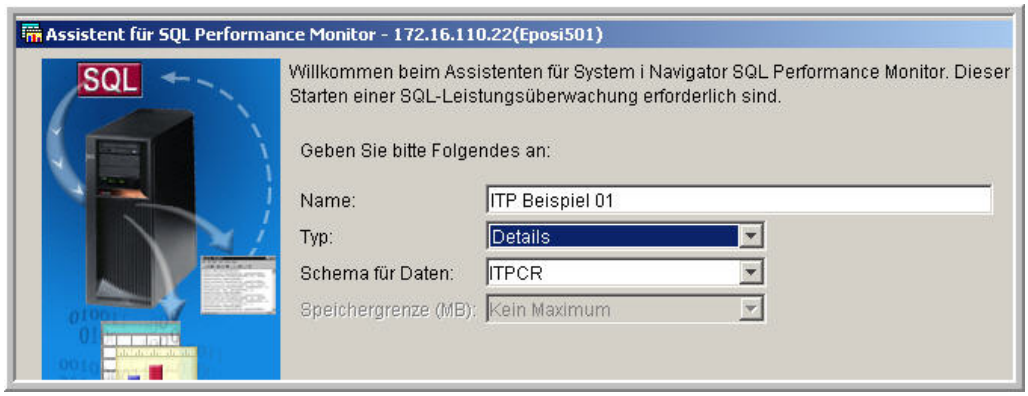
- über den i5/OS-Befehl „Start Database Monitor“ (STRDBMON)
- über den SQL Performance-Monitor Wizard von System i Navigator
- über Open Database Connectivity (ODBC) Clients
- über Object Linking and Embedding (OLE) DB Clients
- über Java Database Connectivity (JDBC) Clients
- über .NET-Clients
- über Exit-Programme

Hier werde ich mich auf System i Navigator und den in Release V5R4 neu integrierten Wizard beschränken.

7.4.3
Seite 2

7.4.3.1 Detailed SQL Performance Monitor starten

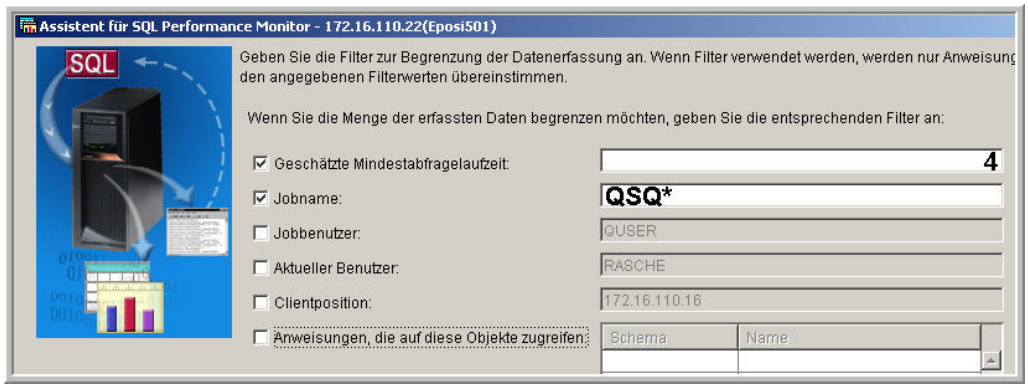
Release V5R4 enthält für die Monitorerfassung einen neuen Wizard mit vielen praktischen Filteroptionen. Um den Wizard zu starten, öffnen Sie mit der rechten Maustaste das Kontextmenü des Eintrags „SQL Performance Monitor“ im System i Navigator; dort wählen Sie den Eintrag „Neu“. Daraufhin erscheint folgendes Begrüßungsfenster:



Begrüßungsfenster

Im Begrüßungsfenster müssen Sie dem Monitor zunächst einen Namen zuweisen und das Schema benennen, in dem die gesammelten Informationen gespeichert werden sollen. Ein Detailmonitor speichert die erfassten Leistungsdaten in Echtzeit und muss nicht angehalten oder beendet werden, um die Ergebnisse zu analysieren. Sie können sogar Visual Explain ausführen, während der Detailmonitor weitere Daten erfasst. Allerdings kann keine physikalische Speichergrenze benannt werden. Sie dürfen also nicht vergessen, den Detailmonitor zu beenden.

Um die Datenmenge zu begrenzen, hinterlegen Sie im nächsten Bildschirm entsprechende Filterkriterien:



Filterkriterien

Das Kriterium „Geschätzte Mindestabfragelaufzeit“
spezifiziert einen Wert in Sekunden. Wenn die geschätzte Ausführungszeit einer SQL-Abfrage geringer ist als der hier benannte Wert, dann werden für die Abfrage keine Leistungsdaten erfasst. Mit diesem Filter können Sie die Datenmenge eingrenzen und erhalten für die Analyse nur die Leistungsdaten schlecht laufender Abfragen.

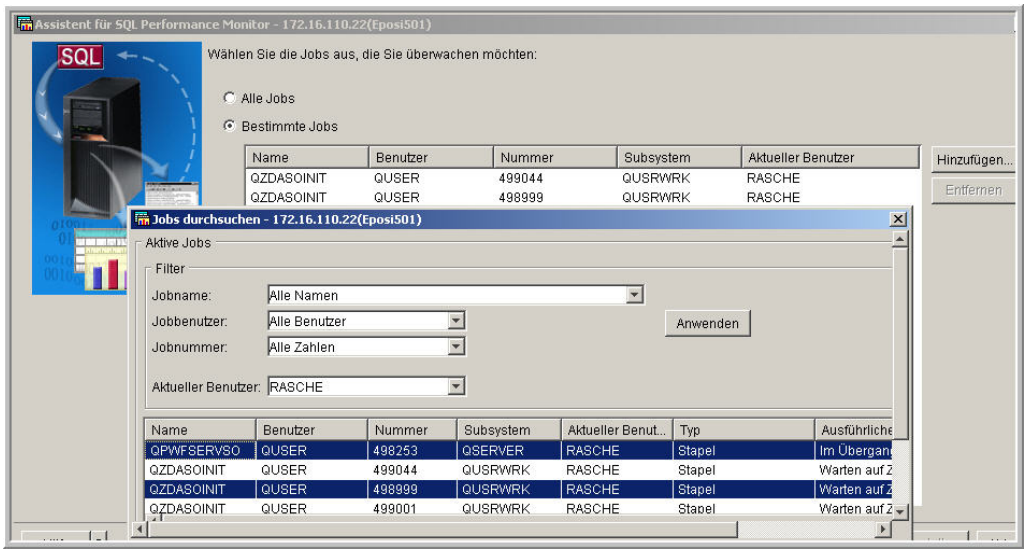
Im Kriterium „Jobname“
hinterlegen Sie spezifische Jobnamen, damit nur für einige Jobs Leistungsdaten erfasst werden. Sie können Wildcards für die Angabe des Jobnamens verwenden.

Die Kriterien „Jobbenutzer“ und „Aktueller Benutzer“
Seit V5R4 unterscheidet die i5 zwischen dem Jobbenutzernamen und dem aktuellen Benutzer. Im Client/Server-Umfeld wird meistens „Quser“ als Jobbenutzerprofil verwendet. Der Anwender selbst wird durch den aktuellen Benutzernamen spezifiziert. Auch hier können Sie Wildcards verwenden.

Clientposition
Manchmal können Sie die Benutzer nicht anhand des Benutzernamens identifizieren, weil mehrere Personen an einem Client arbeiten. Das Filterkriterium „Clientposition“ erlaubt es, Daten für einen oder mehrere Clients zu erfassen. Hierzu geben Sie die TCP/IP-Adresse an.

Anweisungen, die auf diese Objekte zugreifen
Zusätzlich kann es sinnvoll sein, die Datenerfassung auf Abfragen zu beschränken, die sich auf bestimmte Tabellen beziehen. Wenn Sie alle Tabellen eines Schemas überwachen wollen, geben Sie nur den Schemanamen an und lassen das Feld „Name“ leer.

Ein weiterer Selektionsbildschirm ermöglicht Ihnen, die detaillierte Auswahl einzelner Jobs.



Alternative Selektionskriterien zur Jobauswahl

7.4.3
Seite 4

Diesen Bildschirm werden Sie allerdings nur erhalten, wenn zuvor keine Filterkriterien für die zu überwachenden Jobs hinterlegt wurden. Anschließend sehen Sie noch eine Zusammenfassung Ihrer Definition und dann können Sie die Leistungsüberwachung starten.

Filter sind wichtig! Sie sollten nicht auf Filter verzichten, denn es ist unsinnig, für alle SQL-Abfragen Leistungsdaten zu erfassen. Die Informationen sind aufgrund des Volumens nicht interpretierbar und helfen keinesfalls weiter!

Die Filterkriterien der Detailmonitore spiegeln sehr schön die Situationen wieder, in denen es sinnvoll sein kann, Daten für die Leistungsanalyse zu erstellen. Leistungsanalysen werden benötigt, wenn Sie bemerken, dass

- nur eine Anwendergruppe oder einzelne Anwender,
- nur bestimmte Clients,
- nur die Verarbeitung bestimmter Tabellen

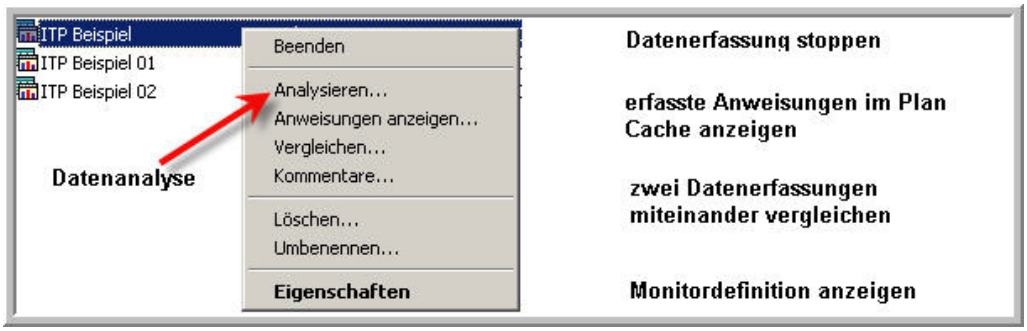
SQL-Probleme verursachen. Wenn Sie hingegen grundsätzlich Probleme mit dem Laufzeitverhalten Ihrer SQL-Abfragen feststellen, dann werden die Leistungsdaten kaum helfen. Die Masse der erfassten Daten ist wenig aussagekräftig. In diesen Fällen ist es sinnvoll, die Indexstrategie zu prüfen, um dafür zu sorgen, dass die Grundlagenwerkzeuge verfügbar sind. Vielleicht haben Sie dann bereits keinerlei SQL-Performance-Probleme mehr und benötigen auch keine Leistungsanalysen.

7.4.3.2 Leistungsanalyse Detailmonitor

Die erfassten Daten werden in Dateien mit dem Präfix „QZG“ gespeichert. Diese Tabellen benötigen häufig viel Platz auf der Festplatte. Sie sollten also von Zeit zu Zeit aufräumen, wenn Sie die Analyse abgeschlossen haben. Alle erfassten Leistungsdaten finden Sie im Menüpunkt „SQL Performance Monitor“ vom System i Navigator:

Aktive Leistungserfassungen

Wenn Sie einen Performance Monitor mit der rechten Maustaste markieren, erscheint ein Kontextmenü:



Kontextmenü der Datenerfassungen

Die Funktionen des Kontextmenüs ermöglichen

- die Erfassung zu beenden,
- sich die überwachten SQL-Abfragen im Plan Cache anzusehen, wenn zwischenzeitlich kein IPL durchgeführt wurde,
- zwei Detailmonitore miteinander zu vergleichen,
- die Monitore zu kommentieren (V6R1),
- die Leistungsdaten zu löschen
- oder sich die Definitionen (Eigenschaften) anzeigen zu lassen.

Und damit sind wir bei der eigentlichen Aufgabe angekommen. Die Leistungsdaten müssen analysiert werden!

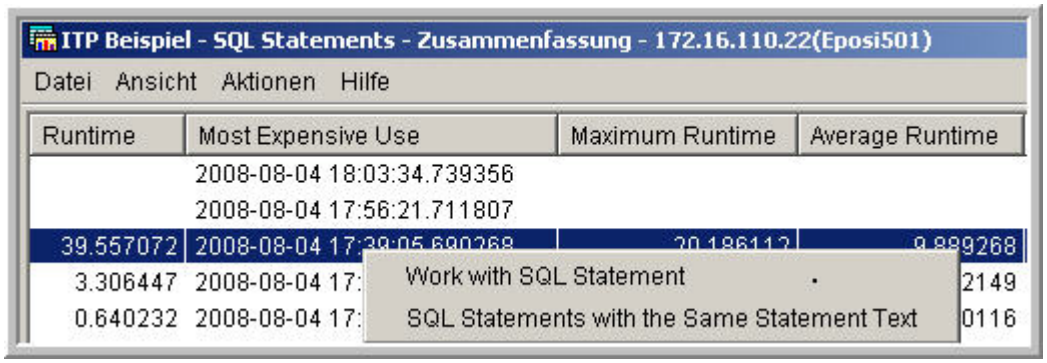
7.4.3
Seite 6

Schauen wir uns die Informationen an, die die Analyse bietet. Zu diesem Zweck klicke ich auf den Eintrag „Analysieren…“ meines Detailmonitors. Daraufhin erscheint ein Übersichtsfenster mit dem erweiterten Eintrag „Overview“:

Übersicht

Unter einem Überblick könnte man auch etwas anderes verstehen. Ohne ein wenig Erfahrung ist die Vielzahl der Informationen anfangs verwirrend. Ich möchte daher auch nicht jeden Eintrag detailliert erläutern, sondern werde mich auf die wesentlichen Dinge beschränken:

Mit dem Detailmonitor wurden 224 SQL-Abfragen erfasst. Für jedes SQL Statement existieren detaillierte Informationen: Statement-Text, längste, kürzeste und durchschnittliche Laufzeit der SQL-Abfrage, Anzahl der Ausführungen, Variablenwerte und vieles mehr. Verfügbar werden die beschriebenen Informationen, wenn Sie die Zeile „SQL Statement“ markieren und anschließend auf den Button „Zusammenfassung“ klicken. Es erscheint ein Fenster mit den detaillierten weiterführenden Informationen.



Zusammenfassung SQL Statement

Die Informationen sind komprimiert, da die Werte mehrfach ausgeführter Abfragen kumuliert werden. Die Informationen eignen sich daher gut, um beispielsweise die Abfragen mit den längsten Laufzeiten und häufigsten Ausführungen zu ermitteln. Das Kontextmenü der Einträge bietet zusätzliche Möglichkeiten:

- „Work with SQL Statement“ öffnet die Prozedurumgebung mit dem markierten SQL Statement zur Bearbeitung oder erneuten Ausführung.
- „SQL Statements with the Same Statement Text“ ist eine Art Filter für das Fenster. Wenn Sie die Option wählen, werden nur Ausführungen des markierten SQL Statements angezeigt.

Der Button „Anweisungen“ stellt Ihnen identische Informationen zur Verfügung, ohne allerdings die Informationen zu verdichten. So können Sie alle 224 Abfragen individuell analysieren. Selbst dann, wenn in der Zwischenzeit ein IPL durchgeführt wurde.

„User“, „Jobs“, „Threads“
Diese Informationen stellen vergleichbare Informationen allerdings für Benutzer, Jobs und Threads zur Verfügung. Sie können die Anzahl ausgeführter SQL Statements je Benutzer bestimmen; Sie erfahren welches SQL Statement schnell und welches langsam ausgeführt wurde. Der Button „Zusammenfassung“ hilft dabei, den Benutzer mit den höchsten durchschnittlichen Ausführungszeiten zu ermitteln oder denjenigen, der durchschnittlich am längsten auf die Datenbank zugegriffen hat. Schauen Sie sich die Informationen einfach mal an!

SQE und CQE
Anhand eines Detailmonitors erfahren Sie schnell, wie viele Ausführungspläne von der CQE erstellt wurden. Wenn Sie feststellen, dass die CQE relativ viele oder sogar mehr SQL Statements ausführt als die SQE, dann sollten Sie die Gründe ermitteln. Markieren Sie den Eintrag „CQE“ und klicken Sie auf den Button „Anweisungen“.

Optimierung durch die CQE

In den Detailinformationen finden Sie die Gründe. Meine SQL-Abfragen wurden von der CQE optimiert, weil einerseits logische DDS-Dateien im SQL Statement verwendet wurden, andererseits die JDBC-Schnittstelle falsch konfiguriert und eine CCSID-Konvertierung erzwungen wurde.

7.4.3
Seite 8

Open Informationen

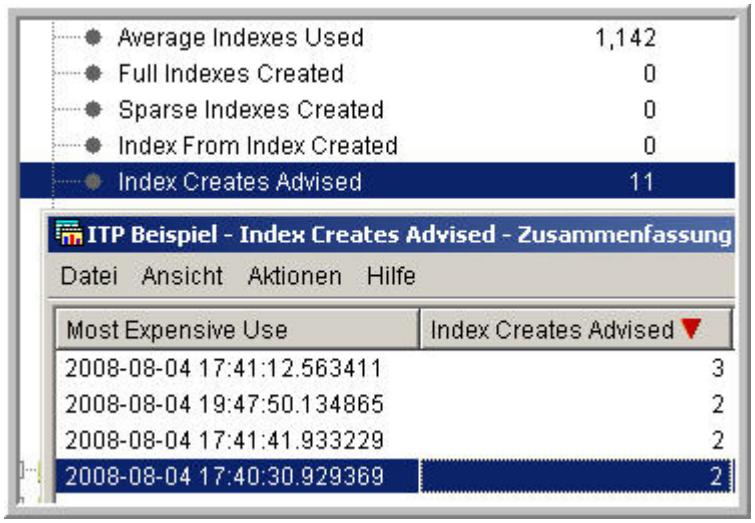
Die drei Einträge „Unique open Statements“, „Full Opens“ und „Pseudo Opens“ liefern Hinweise zur Wiederverwendung geöffneter Datenpfade. „Unique open Statements“ zeigt, wie viele Zugriffspfade für die erfassten Daten erstellt wurden. Doppelte Erstellungen bleiben hierbei unberücksichtigt. Die SQL Engine muss für die Ausführung einer Abfrage einen Datenpfad (ODP) erstellen. Für die erste Ausführung muss der ODP stets vollständig geöffnet („Full Open“) werden. Häufig benötigt dieser Ausführungsschritt viel Zeit. Aus diesem Grund verfügt die SQL Engine über die Fähigkeit, den ODP nach Ausführung der Abfrage nicht vollständig zu schließen. Wird nun die Abfrage innerhalb des Jobs oder der Verbindung mit anderen Abfragewerten erneut ausgeführt, dann kann der ODP wieder verwendet werden. In diesen Fällen spricht man von einem „Pseudo Open“. Versuchen wir jetzt, mein Ergebnis zu interpretieren:

Analyse der Datenpfaderstellung

„11“ Datenpfade wurden eindeutig erstellt. Diese ODPs müssen vollständig geöffnet werden. Insgesamt wurden zusätzlich „14“ Datenpfade vollständig geöffnet. Nur einer konnte wieder verwendet werden. Ist der Zähler für „Full Open“ relativ hoch im Verhältnis zu den beiden anderen Zählern, dann könnte das ein Hinweis darauf sein, dass eines der Probleme in der permanenten Neuerstellung der Zugriffspfade liegt. Bei mir ist der relativ hohe „Full-Open“-Wert allerdings dadurch zu erklären, dass ich in den JDBC-Einstellungen der Prozedurumgebung die Wiederverwendung der ODPs explizit ausschließe, um Visual Explain ohne Probleme nutzen zu können. Bei der Interpretation dieser Zahlen ist auch zu berücksichtigen, dass die Zähler die Werte vieler verschiedener Jobs kumulieren. ODPs aber können nur innerhalb eines Jobs oder einer Client-Verbindung wieder verwendet werden.

FULL Index Created und Index Creates Advised

Die nächsten Informationen zeigen, wie viele Indizes die SQL Engines erstellen mussten, um die Abfragen auszuführen.

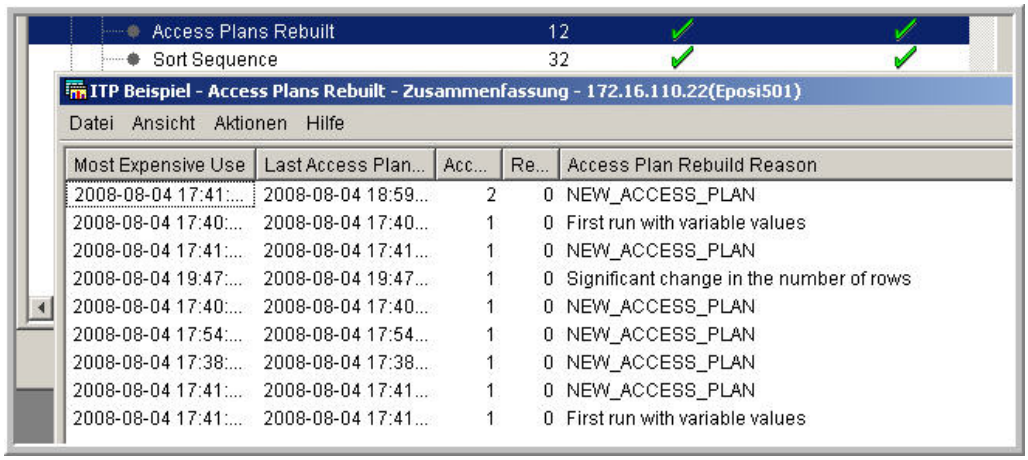


Index Advised

Wenn häufig temporäre Indizes erstellt werden und die durchschnittliche Nutzung der permanenten Indizes gering ist, dann könnte dies ein erster Anhaltspunkt dafür sein, dass die Indexstrategie nicht schlüssig ist. Die Vermutung wird in meinem Beispiel durch die vielen Indexvorschläge bestätigt. Doch Sie müssen berücksichtigen, dass der Index-Advisor jeden erdenklichen Index vorschlägt, der für die Ausführung der SQL-Abfrage relevant sein könnte – unabhängig davon, wie schnell die Abfrage ausgeführt werden konnte. Sie sollten daher sehr genau die zusammenfassenden Informationen analysieren, bevor Sie handeln.

7.4.3
Seite 10

Access Plan Rebuilt

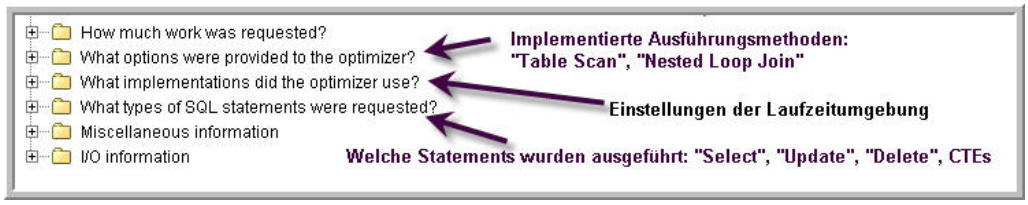


Most Expensive Use	Last Access Plan...	Acc...	Re...	Access Plan Rebuild Reason
2008-08-04 17:41:...	2008-08-04 18:59...	2	0	NEW_ACCESS_PLAN
2008-08-04 17:40:...	2008-08-04 17:40...	1	0	First run with variable values
2008-08-04 17:41:...	2008-08-04 17:41...	1	0	NEW_ACCESS_PLAN
2008-08-04 19:47:...	2008-08-04 19:47...	1	0	Significant change in the number of rows
2008-08-04 17:40:...	2008-08-04 17:40...	1	0	NEW_ACCESS_PLAN
2008-08-04 17:54:...	2008-08-04 17:54...	1	0	NEW_ACCESS_PLAN
2008-08-04 17:38:...	2008-08-04 17:38...	1	0	NEW_ACCESS_PLAN
2008-08-04 17:41:...	2008-08-04 17:41...	1	0	NEW_ACCESS_PLAN
2008-08-04 17:41:...	2008-08-04 17:41...	1	0	First run with variable values

Access Plan Rebuilt

Der Detailmonitor enthält die Anzahl neu erstellter Ausführungspläne. Ist die Anzahl der erstellten Pläne relativ hoch, sollten die Gründe hierfür geprüft werden. In meiner Beispielanalyse werden nur „12“ Pläne neu erstellt, aber „224“ Abfragen ausgeführt. Das Verhältnis scheint in Ordnung. Dennoch nutze ich den Button „Zusammenfassung“, um die Begründung für die Planerstellung zu erfahren. Wie erwartet, kann ich die Neuerstellungen nicht verhindern, weil die Abfragen entweder zum ersten Mal ausgeführt wurden oder die veränderte Datenlage eine Neuerstellung des Ausführungsplanes erforderte.

Wenn Sie sich die ersten Informationen der Analyse im Überblick angeschaut haben, sind Sie keineswegs fertig. Es gibt weitere Rubriken, die zusätzliche Informationen enthalten:

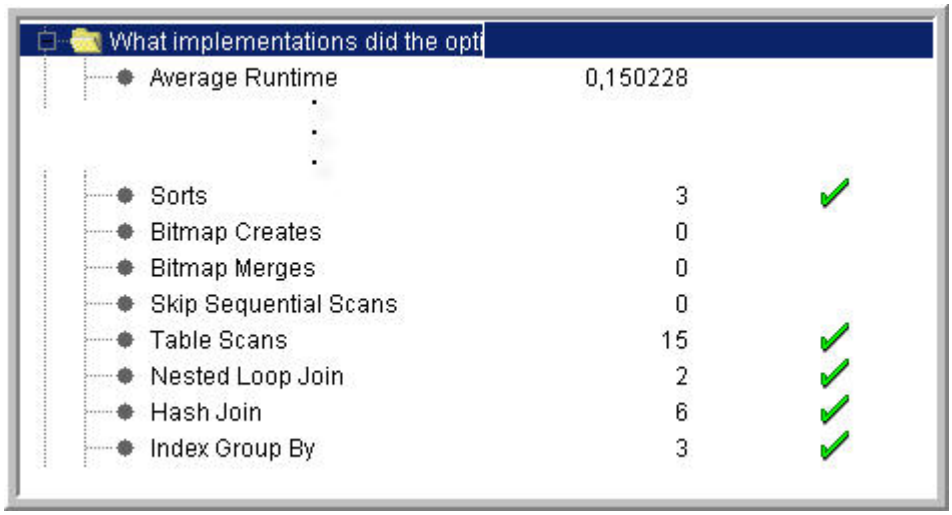


Weitere Informationen

„How much work was requested?“
Die Rubrik enthält bekannte Informationen der Rubrik „Overview“. Informationen über die Anzahl ausgeführter SQL Statements, Benutzer, Jobs und Threads werden hier lediglich wiederholt, damit Sie sich schnell einen Überblick über die erfassten Abfragen verschaffen können.

„What options were provided to the optimizer?“
Wenn Sie die Laufzeitumgebung prüfen wollen oder die verwendeten QAQQINI-Attribute analysieren möchten, müssen Sie die Rubrik „What options were provided to the optimizer?“ öffnen.

„What implementations did the optimizer use?“
Dem Optimizer stehen für die Ausführung Ihrer SQL-Abfrage viele verschiedene Implementierungen zur Verfügung. Welche Ausführungsstrategie gewählt wurde, zeigt die Rubrik „What implementations did the optimizer use?“



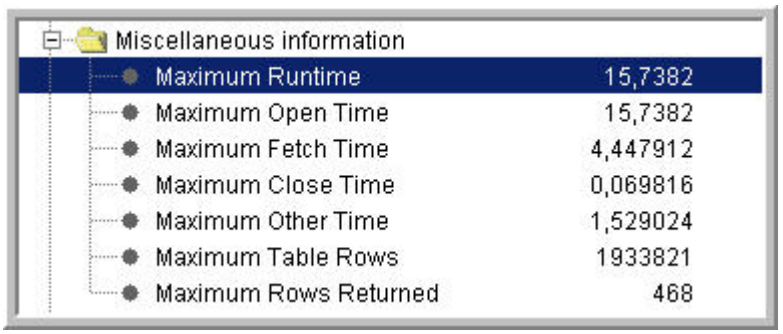
Verwendete Implementierungen

Die Methode „Table Scan“ und der Aufbau temporärer Objekte ist von besonderem Interesse. In meiner Beispielanalyse verwendet der Optimizer häufig die Methode „Table Scan“. Der Grund hierfür könnten fehlende Indizes sein. Außerdem werden häufig für „Join“-Abfragen temporäre Hash-Tabellen aufgebaut, um die Daten der Sekundärtabellen zu lesen. Entweder standen auch hier keine geeigneten Indizes zur Verfügung oder die Anzahl der I/O-Operationen rechtfertige den Aufbau der temporären Hash-Tabellen.

„What types of SQL Statements were requested?“
Wollen Sie analysieren, welche Arten von SQL Statements ausgeführt wurden, müssen Sie diese Rubrik öffnen. Sie erfahren wie viele „Inner“-Joins, „Left Outer“-Joins, „Union“-Operationen oder Gruppierungen ausgeführt wurden.

7.4.3
Seite 12

„Miscellaneous information“
Die Rubrik „„Miscellaneous information“ enthält verschiedene statistische Werte.



Verschiedene statistische Werte

Hier finden Sie schnell die SQL-Abfrage mit der längsten Laufzeit oder dem größten Result Set. Markieren Sie den entsprechenden Eintrag und lassen Sie sich die Anweisung anzeigen, indem Sie auf den Button „Anweisungen“ klicken.

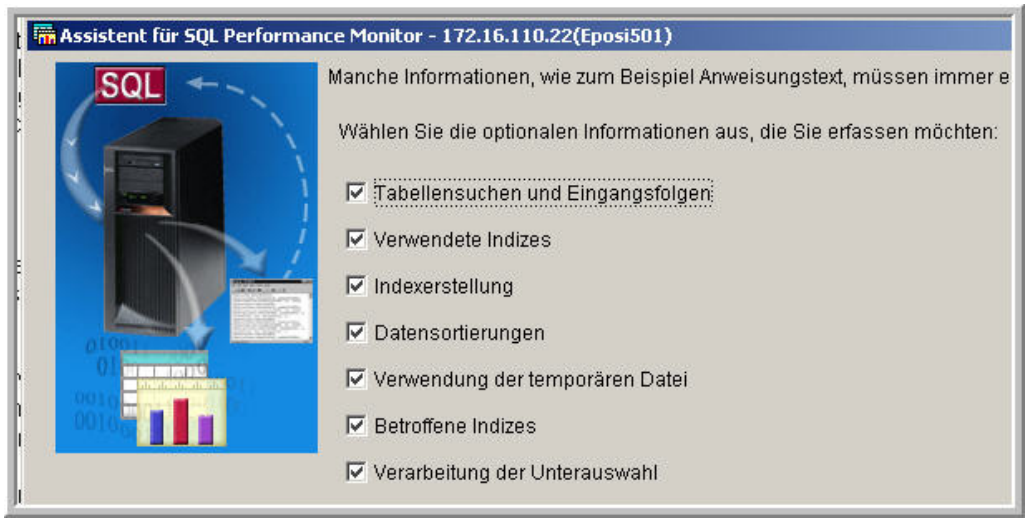
„I/O information“
Die Anzahl der Plattenzugriffe ist ein entscheidender Faktor für die Ausführungszeiten Ihrer SQL-Abfragen. Wie viele asynchrone und synchrone Lesezugriffe erforderlich waren, wie viele Fehlseiten gelesen wurden, erfahren Sie in der Rubrik „I/O information“.

Ich denke, die Beispiele vermitteln einen ersten Eindruck über die vielfältigen Informationen der SQL Performance-Monitore. Sie können später helfen, Laufzeitprobleme Ihrer SQL-Applikationen aufzudecken.

7.4.3.3 Summary Performance-Monitor starten

Der Wizard für eine Monitorzusammenfassung wurde im Release V5R4 nicht verändert. Der Aufruf wurde lediglich in den Wizard des Detailmonitors integriert. Die Datenaufnahme der Monitorzusammenfassung findet im Arbeitsspeicher statt. Erst wenn der Monitor angehalten oder beendet wird, werden diese Daten auf die Festplatte geschrieben und können analysiert werden.

Um die Monitorzusammenfassung zu starten, gehen Sie zunächst genauso vor wie eben erklärt. Allerdings wählen Sie im Begrüßungsfenster des Wizard den Typ „Zusammenfassung“. Danach folgt ein etwas anderes Selektionsfenster:



Selektionen für eine Monitorzusammenfassung

Die Selektionskriterien „Tabellensuche und Eingangsfolgen“, „Indexerstellung“, „Datensortierungen“, „Verwendung der temporären Dateien“, „Verarbeitung der Unterauswahl“ beziehen sich auf die Ausführungen der SQL-Abfragen. Wenn im Ausführungsplan temporäre Objekte, die Methode „Table Scan“ oder aufwendige Datensortierungen durchgeführt werden, erhalten Sie für diese Ausführungen Leistungsdaten. In den Leistungsdaten können Begründungen für die verwendeten Indizes eingeschlossen werden, wenn Sie die Filterkriterien „Verwendete Indizes“ und „Betroffene Indizes“ aktivieren. Verwendet der Optimizer häufig diese Methoden für die Ausführung Ihrer Abfragen, dann könnte dies die Vermutung bestätigen, dass geeignete Indizes für die Implementierung fehlen.

In einem weiteren Bildschirm stehen auch hier Möglichkeiten zur Verfügung, die Erfassung der Daten auf einzelne Jobs zu beschränken. Es folgt eine Zusammenfassung Ihrer Angaben; anschließend kann auch die Monitorzusammenfassung gestartet werden.

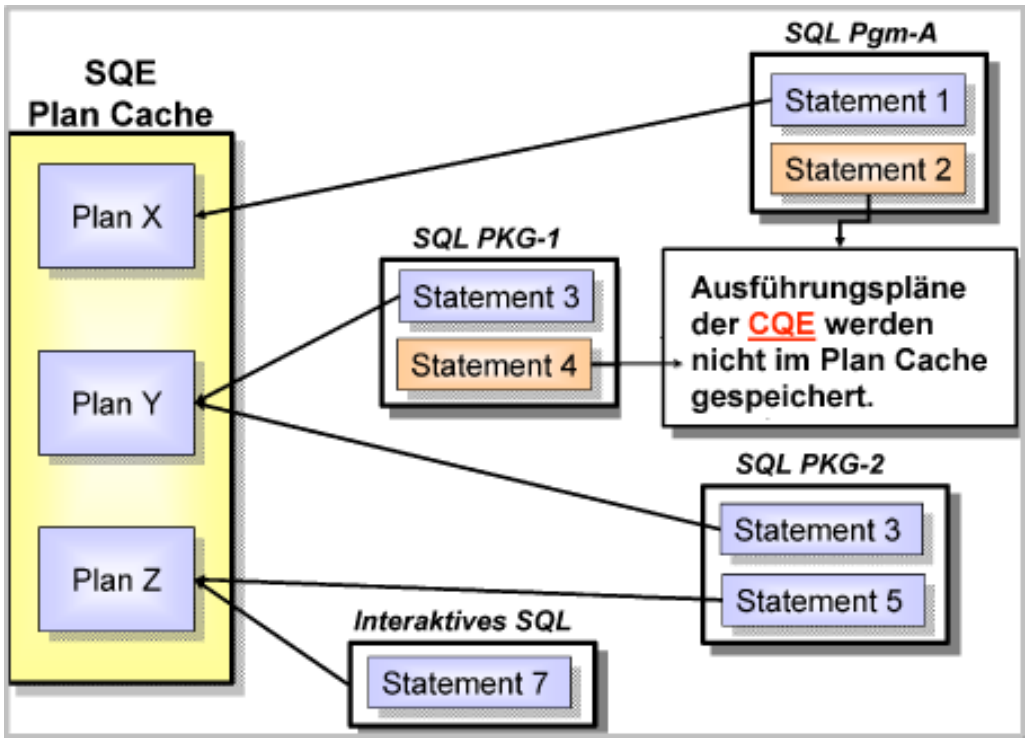
7.4.3
Seite 14

Zusammenfassende Leistungserfassungen benötigt man in erster Linie, um einen Überblick über die Ausführungsmethoden der SQL-Abfragen zu erhalten. Wie werden die Abfragen ausgeführt? Werden Indizes verwendet? Wie häufig müssen temporäre Objekte für die Ausführungen erstellt werden? Werden temporäre Indizes erstellt? Antworten auf diese Fragen erlauben erste Rückschlüsse darauf, ob und warum SQL Performance-Probleme vorhanden sind.

7.4.4 Der Plan Cache

Der Plan Cache ist keine unmittelbare Komponente der SQE; er ist das „Lager" für bestehende Ausführungspläne. Die Wirkungsweise des Plan Cache ist einfach, aber effektiv. Im Plan Cache werden alle optimierten Zugriffspläne der SQE gespeichert. Dadurch ist die Wiederverwendung garantiert. Wird dieselbe oder eine ähnliche Abfrage erneut ausgeführt, kann der gespeicherte Plan für die Ausführung der Abfrage verwendet werden. Die Optimierungszeit wird so eingespart. Die gespeicherten Pläne sind für alle SQL-Abfragen – unabhängig vom genutzten Interface oder Benutzer – zugänglich.

Die folgende Grafik verdeutlicht dieses Konzept:



Wirkungsweise des SQE Plan Cache

Schauen Sie sich die Abbildung an und stellen Sie sich vor, dass das SQL Statement „3" zum ersten Mal durch die SQE ausgeführt wird. Der Optimizer prüft den Plan Cache, findet keinen geeigneten Ausführungsplan und ermittelt daraufhin einen geeigneten Zugriffsweg. Die Abfrage wird ausgeführt und anschließend im Plan Cache gespeichert. Zu einem späteren Zeitpunkt wird das gleiche SQL Statement von einem anderen Benutzer ausgeführt. Der Optimizer untersucht daraufhin erneut den Plan Cache, findet jetzt den zuvor gespeicherten Plan und kann die Abfrage ohne erneute Optimierung ausführen. Neben der Originalabfrage und dem Zugriffsplan enthält der Ausführungsplan auch generierte ausführbare Query-Runtime-Objekte sowie Infor-

Ergänzung 9/2009

7.4.4
Seite 2

mationen über die Laufzeitumgebung. Da das SQL Statement selbst als Primary Hash Key verwendet wird, ist es auch möglich, für identische SQL Statements derzeit maximal drei unterschiedliche Ausführungspläne zu hinterlegen. So können unterschiedliche Laufzeitumgebungen abgebildet werden:

- unterschiedliche SMP-Einstellungen
- unterschiedliche Bibliothekslisten
- unterschiedliche ALWCPYDTA-Einstellungen
- unterschiedliche FOR-OPTIMIZE-Klauseln

Die CQE kann den Plan Cache nicht nutzen. Wenn wir davon ausgehen, dass die Statements „2“ und „4“ in der Abbildung von der CQE ausgeführt werden, dann werden diese Ausführungspläne nicht im Plan Cache hinterlegt. Sie müssen vor jeder Ausführung neu optimiert werden, wenn sie nicht in SQL Packages gespeichert wurden. In gewisser Weise können Sie den Plan Cache in Funktion und Wirkungsweise mit den SQL Packages vergleichen. Der Unterschied besteht darin, dass SQL Packages vorwiegend bei verteilten Anwendungen genutzt oder für dynamisches SQL manuell erstellt werden müssen. Der Plan Cache hingegen wird automatisch gefüllt. Eine schlecht optimierte Abfrage bleibt auf Grund dieses Konzeptes somit über einen längeren Zeitraum bestehen und wird von vielen Jobs genutzt, ohne eine erneute Optimierungsphase zu durchlaufen. Allerdings besitzt der Optimizer durchaus die Fähigkeit, gespeicherte Ausführungspläne für ungültig zu erklären, wenn er der Meinung ist, bessere Ausführungsstrategien auf Grund neuer Objekte oder neuer Informationen zu finden. Neue Indizes und Statistiken werden vom Statistik-Manager im Plan Cache hinterlegt. Erkennt der Optimizer die neuen Informationen, wird der Ausführungsplan automatisch neu optimiert.

Die SQE speichert im Übrigen auch die Ausführungspläne dynamischer SQL-Anweisungen im Plan Cache. Für dynamische SQL Statements wird der erste Ausführungsplan erstellt, wenn das Prepare-Statement ausgeführt wird – also zur Laufzeit. Da der Optimizer aber für seine Entscheidungen die Inhalte der Host-Variablen berücksichtigt, kann es passieren, dass der Plan neu aufgebaut wird, wenn die Werte der Host-Variablen verfügbar sind. Enthält ein HLL-Programm hingegen statische SQL-Anweisungen, kann der erste Ausführungsplan bereits zur Umwandlungszeit des Programms erstellt werden. Aber auch in diesem Fall könnte der erste Ausführungsplan aufgrund der Werte der Host-Variablen zur Laufzeit verworfen werden.

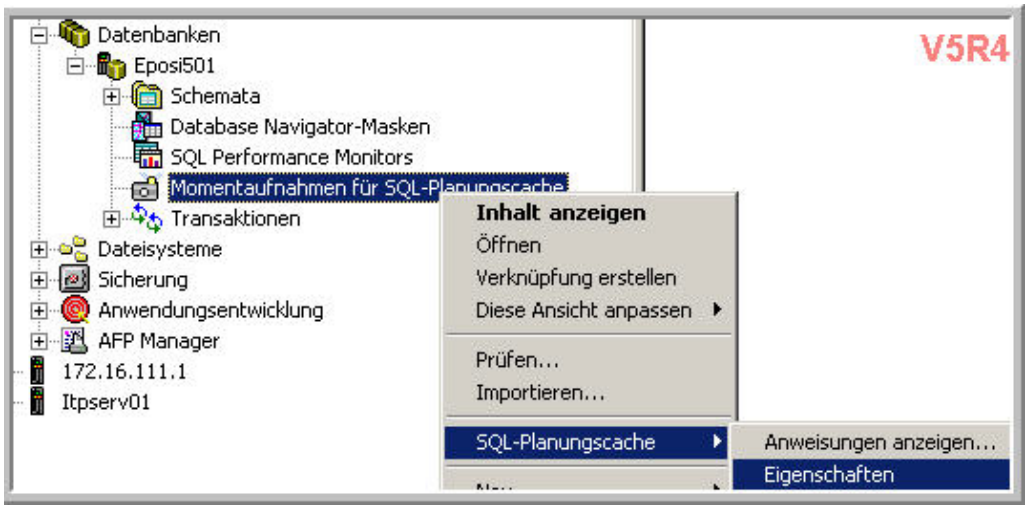
Unabhängig vom Maschinentyp und der Hardware-Ausstattung Ihres Servers beansprucht der Plan Cache maximal 512 MB Hauptspeicher. Wie viele Pläne in diesem Bereich gespeichert sind, kann nicht allgemein beantwortet werden, da die Komplexität der jeweiligen SQL Statements hierbei eine entscheidende Rolle spielt. IBM spricht von ca. 5.000 Plänen. Wenn der Cache-Bereich die Maximalgröße erreicht, dann bereinigt automatisch ein Hintergrundjob die Einträge. Hierbei spielt das Datum der Erstellung und die Anzahl der Zugriffe auf den jeweiligen Plan eine entscheidende Rolle. Bis Release V5R4 haben Sie nur indirekt Einfluss auf die Neuerstellung der Pläne. Wenn Sie

- für das Attribut REOPTIMIZE_ACCESS_PLAN der Attributdatei "QAQQINI" den Wert „*YES" oder „*FORCE" verwenden,
- eine Tabelle oder einen Index löschen, der im gespeicherten Zugriffsplan verwendet wird,
- einen neuen Index-Kandidaten erstellen,

dann wird der gespeicherte Ausführungsplan verworfen.

Um alle gespeicherten Pläne zu löschen, führen Sie ein IPL durch. Bedenken Sie dabei aber, dass anschließend jede SQL-Abfrage neu optimiert werden muss und sich dadurch gegebenenfalls die Ausführungszeit verlängert.

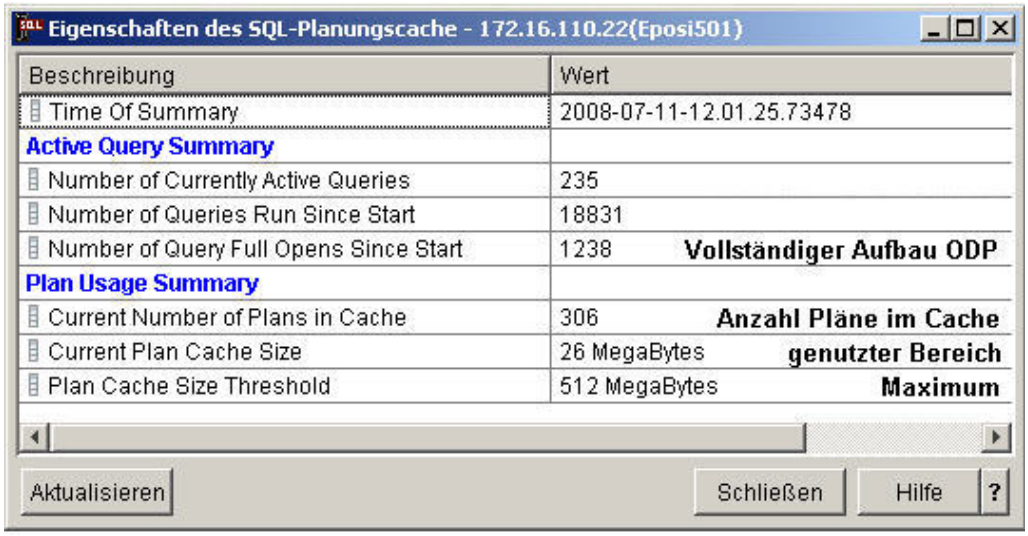
Seit Release V5R4 können Sie sich die Eigenschaften des Plan Cache anzeigen lassen. System i Navigator stellt dafür ein geeignetes Interface zur Verfügung. Öffnen Sie hierfür zunächst den Eintrag „Datenbanken" und anschließend das Kontextmenü des Eintrages „Momentaufnahmen für SQL Planungscache":



„SQL Planungscache" – V5R4

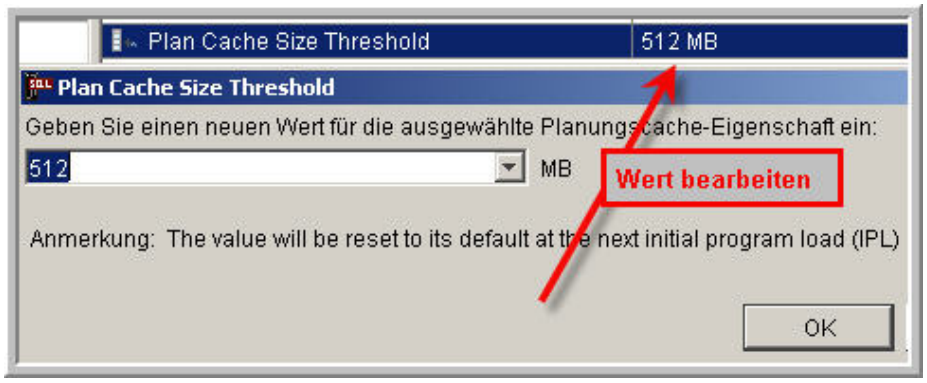
7.4.4
Seite 4

Jetzt wählen Sie den Eintrag „Eigenschaften“. Daraufhin erscheint das „Eigenschaftsfenster“:



Eigenschaften des Plan Cache

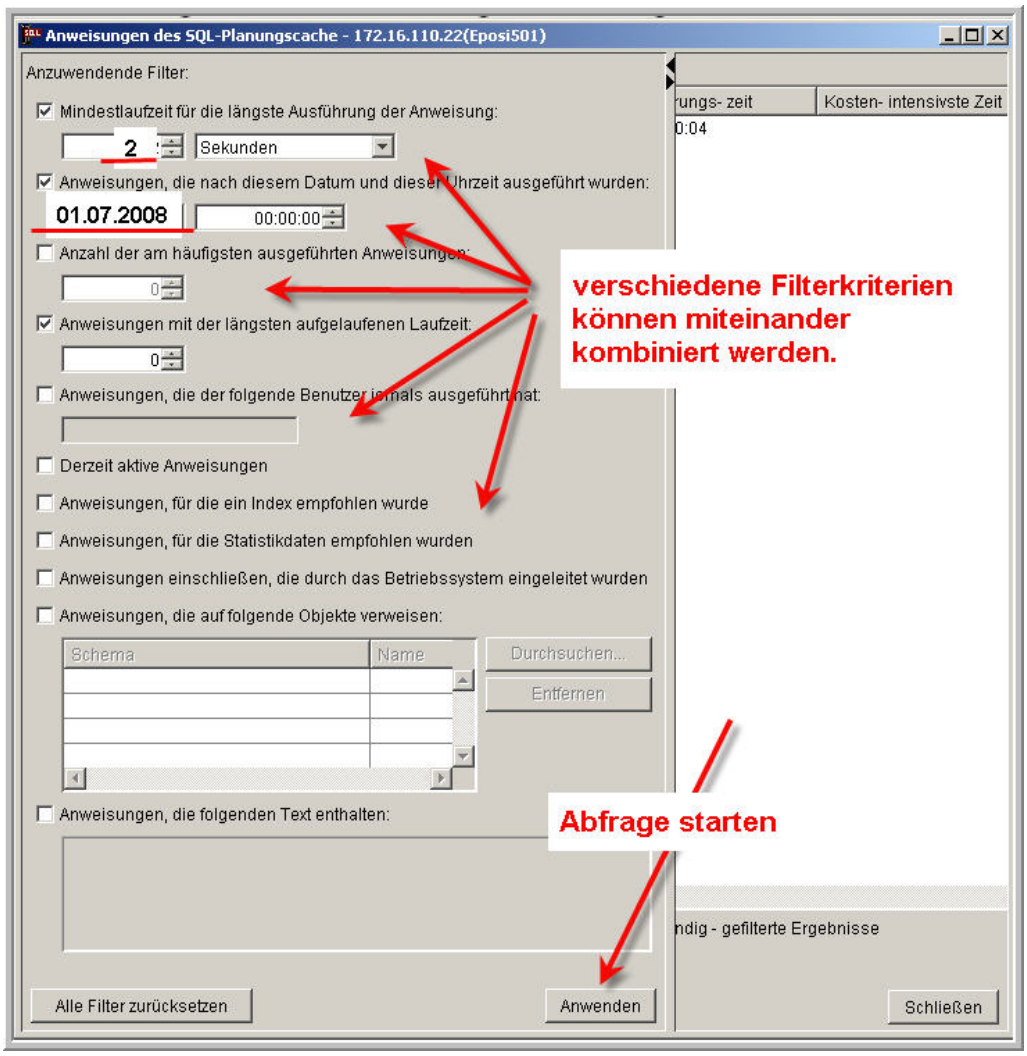
In meiner Testumgebung wurden seit dem letzten IPL 18.831 SQL Abfragen ausgeführt. Für die Abfragen wurden 306 Ausführungspläne neu erstellt und im Plan Cache gespeichert. Genutzt werden hierfür 26 MB Hauptspeicher. Seit Release V6R1 können Sie zudem die Maximalgröße des Cache-Bereiches manuell beeinflussen. Hierfür klicken Sie mit der rechten Maustaste auf den Eintrag „Plan Cache Size Threshold“ und wählen „Wert bearbeiten“. Es öffnet sich ein neues Dialogfenster:



Eigenschaften des Plan Cache ändern – V6R1

Sie können jetzt die Maximalgröße des Plan Cache individuell festlegen. Wenn Sie ein IPL durchführen, wird dieser Wert allerdings immer auf den Default-Wert 512 MB zurückgesetzt. Alternativ steht die SQL-Prozedur „change_plan_cache_size“ im Schema „QSYS2“ zur Verfügung, um die Größe des Plan Cache anzupassen. Seien Sie aber vorsichtig mit Ihren Veränderungen und beobachten Sie genau, welche Auswirkung Ihre Veränderungen auf die Performance Ihrer SQL-Abfragen haben.

Lange Zeit blieb der Inhalt des Plan Cache dem Anwender verborgen. Aber seit Release V5R4 ist es möglich, sich die gespeicherten Ausführungspläne anzeigen zu lassen. Öffnen Sie hierfür die Datenbank ➜ Rechtsklick auf „Momentaufnahmen für SQL Planungscache“ ➜ „SQL Planungscache“ ➜ „Anweisungen anzeigen…“. Daraufhin öffnet sich das Fenster „Anweisungen des SQL-Planungscache“:

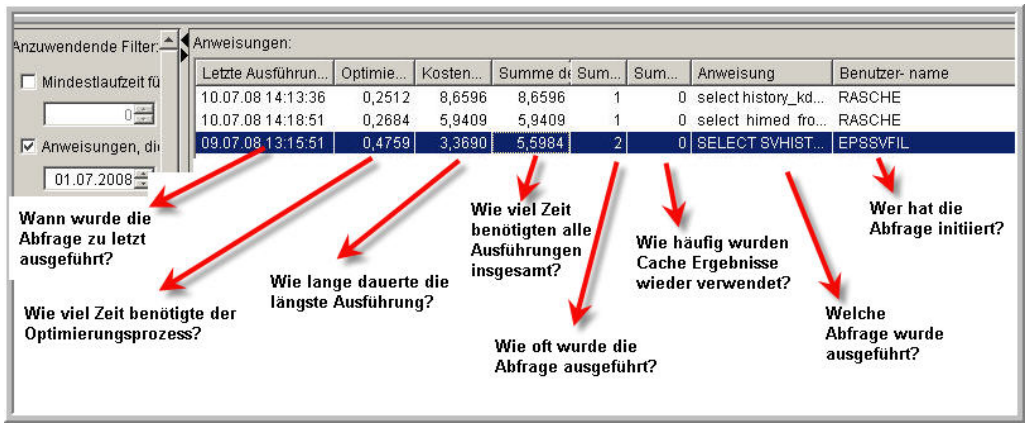


Ausführungspläne selektieren

Das Fenster besteht aus zwei Bereichen. Der rechte Bereich enthält die Ausführungspläne und ist zunächst leer. Der linke Fensterbereich dient der Selektion der gespeicherten Ausführungspläne. Hier können Sie zwischen verschiedenen Filterkriterien wählen, auch eine Kombination der Selektionskriterien ist möglich.

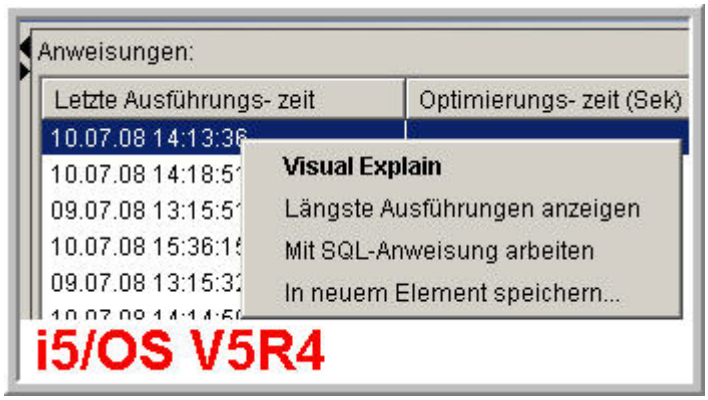
7.4.4
Seite 6

Ich möchte mir in einem nächsten Schritt alle Anweisungen anzeigen lassen, die nach dem 01.07.2008 ausgeführt wurden und deren längste Ausführung mindestens zwei Sekunden dauerte. Hierfür trage ich zunächst im Selektionskriterium „Anweisungen, die nach diesem Datum und dieser Uhrzeit ausgeführt wurden“ das Datum 01.07.2008 ein und setze im Kriterium „Mindestlaufzeit für die längste Ausführung der Anweisung“ den Wert „2“ ein. Anschließend klicke ich auf den Button „Anwenden“. Nach einer gewissen Wartezeit erscheint das Ergebnis im rechten Fenster:



Die selektierten Ausführungspläne

Sie erhalten eine Vielzahl nützlicher Informationen, wobei leider die Überschriften teilweise sehr schlecht übersetzt wurden. Über das Kontextmenü der Zugriffspläne stehen weitere Funktionen zur Verfügung.



Kontextmenü i5/OS V5R4

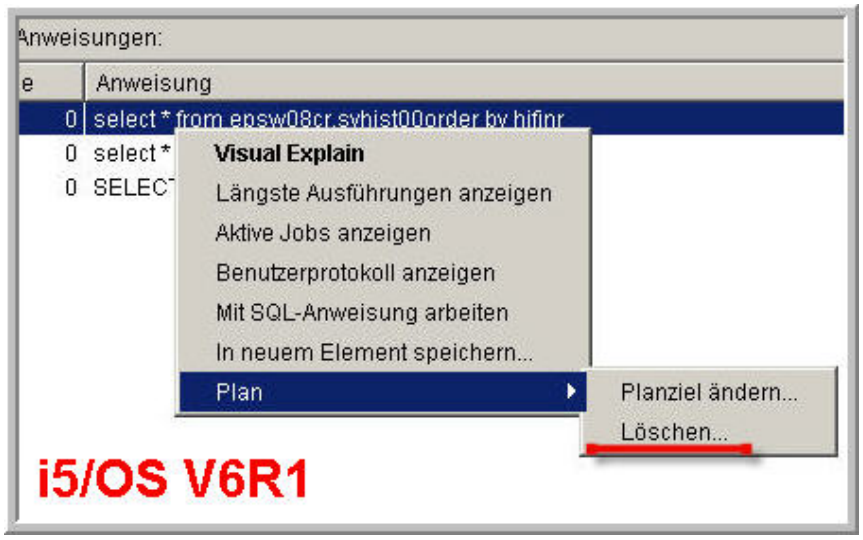
Sie können sich beispielsweise den grafisch aufbereiteten Ausführungsplan ansehen, indem Sie Visual Explain aufrufen oder die SQL Abfrage in der Prozedurumgebung von System i Navigator bearbeiten und erneut ausführen. Interessant in diesem Zusammenhang ist die Option „Längste Ausführungen von SQL-Anweisungen anzeigen“. In einem neuen Fenster werden die längsten Ausführungen dieses Statements dargestellt.

Längste Ausführungen:

Ausführungs- zeit	Verarbeitungs- zeit...	Ausgewählte Sätze	CPU-Zeit ...	Synchrone...	Asynchrone...	An...	An...	Fehl...
07.07.08 12:12:55	0,5704	7298	0,0177	6	306	78	0	84
08.07.08 10:58:04	0,5067	7300	0,0147	3	39	97	0	100
07.07.08 14:23:15	0,4575	7298	0,0163	6	198	63	0	69
07.07.08 12:05:05	0,4251	7298	0,0215	8	600	7	0	15
09.07.08 09:59:40	0,3905	7300	0,0187	4	324	3	0	7
07.07.08 11:55:11	0,3813	7298	0,0223	2	717	3	0	5
08.07.08 09:55:19	0,3326	7298	0,0149	4	56	113	0	117
08.07.08 11:07:08	0,2910	7300	0,0147	3	38	97	0	100
08.07.08 09:50:56	0,1564	7298	0,0149	5	63	1	0	6

Längste Ausführungen anzeigen

Hier können Sie die benötigte CPU-Zeit, die Anzahl synchroner und asynchroner Lesezugriffe und vieles mehr analysieren. Seit Release V6R1 hat sich das Kontextmenü der angezeigten Zugriffspläne verändert.



Kontextmenü V6R1

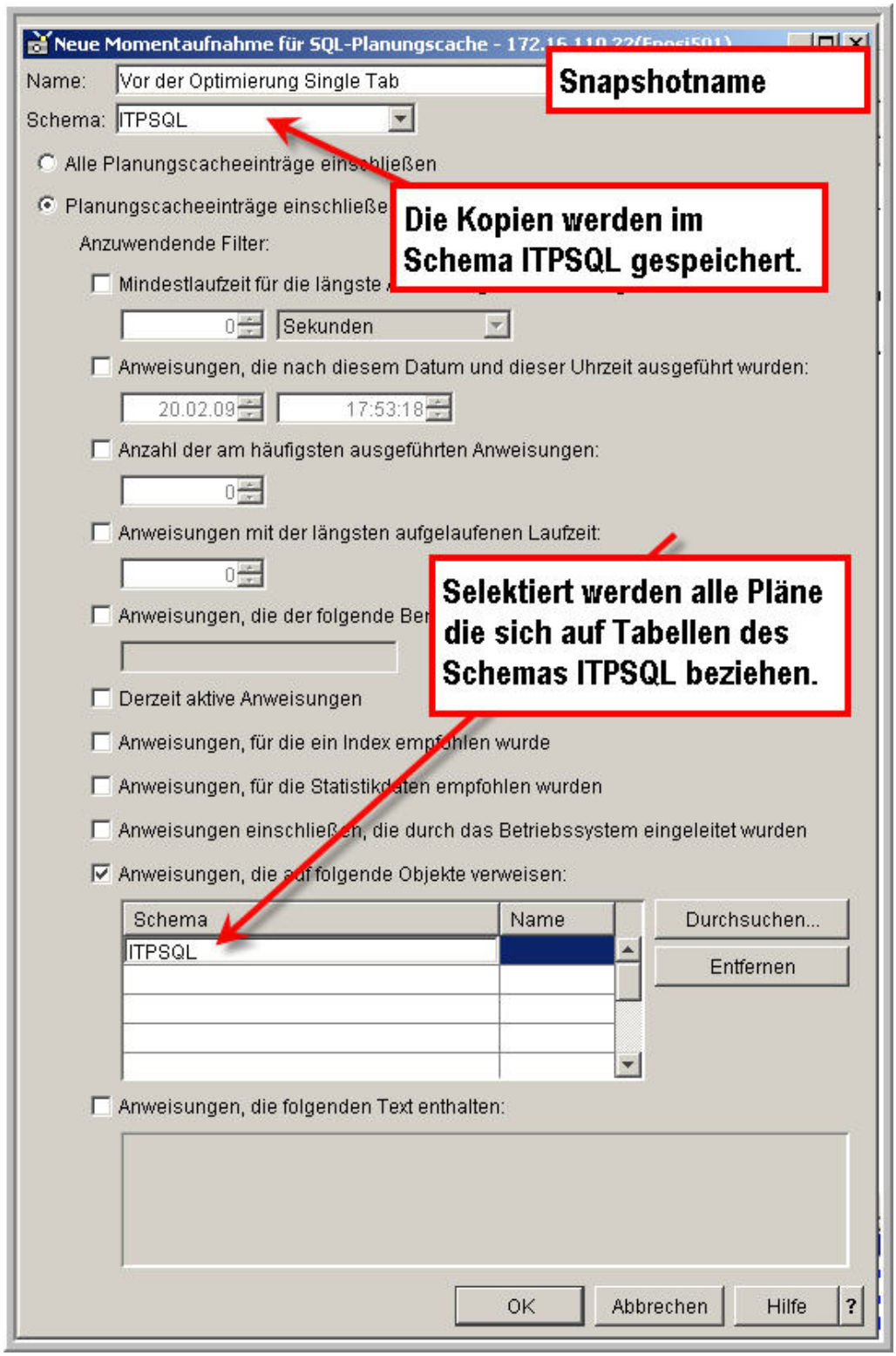
Erwähnenswert in diesem Zusammenhang ist insbesondere der Punkt „Planziel ändern“ und „Löschen“. Die Option „Löschen“ ermöglicht es, gezielt einzelne Ausführungspläne aus dem Plan Cache zu entfernen. Mit der Funktion „Planziel ändern“ haben Sie Einfluss auf die automatische Bereinigung des Plan Cache. Je höher der hier hinterlegte Wert ist, desto unempfindlicher sind die Pläne bei der automatischen Bereinigung.

7.4.4
Seite 8

7.4.4.1 Snapshot des SQL Plan Cache

Der Plan Cache bietet weitere Möglichkeiten als die zuvor erwähnten Funktionen und Analysemöglichkeiten. Die Informationen des Plan Cache gehen nämlich verloren, wenn Sie ein IPL ausführen. Das bedeutet, dass alle Pläne unwiderruflich gelöscht sind. Bereits aus diesem Grund kann es vorteilhaft sein, eine Kopie der aktuellen Ausführungspläne zu erstellen. Außerdem können nur drei Ausführungspläne für ein SQL Statement im Plan Cache hinterlegt werden. Wenn nun aber viele Anwender dasselbe SQL Statement mit unterschiedlichen Laufzeitumgebungen ausführen, dann werden manche Ausführungspläne bereits nach wenigen Minuten verworfen und können nicht mehr analysiert werden. Auch für derartige Situationen empfiehlt es sich, eine Kopie der entsprechenden Pläne zu erstellen, um Sie später jederzeit in Ruhe zu analysieren und gegebenenfalls mit neuen Plänen zu vergleichen. Und es gibt weitere interessante Einsatzmöglichkeiten: Sie optimieren eine SQL-Applikation, erstellen zu diesem Zweck neue Indizes und möchten die Auswirkungen prüfen („Vorher-Nachher-Situation“).

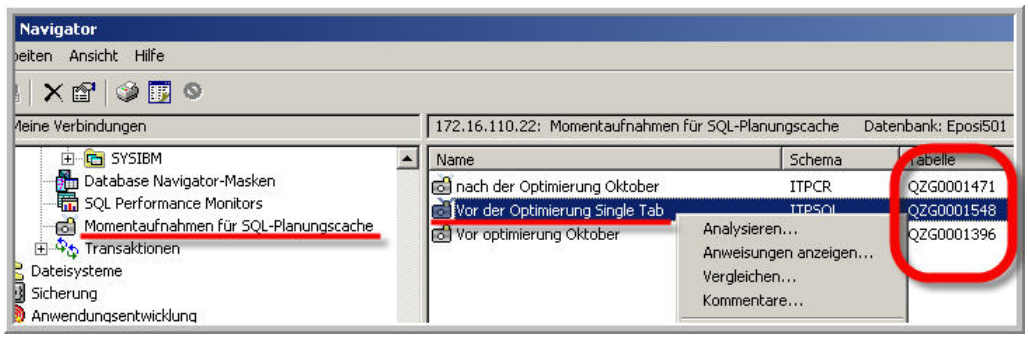
Für derartige Situationen bieten sich Snapshots an. Um mit dem System i Navigator Snapshots (=Momentaufnahmen) zu erstellen, klicken Sie mit der rechten Maustaste auf den Eintrag „Momentaufnahmen für SQL-Planungscache“ ➜ „Neu“ ➜ „Momentaufnahme“. Nach kurzer Zeit erscheint der bereits bekannte Bildschirm mit diversen Filterkriterien. Ich sichere in dem folgenden Beispiel alle Ausführungspläne, die derzeit für Objekte des Schemas „ITPSQL“ vorhanden sind, bevor ich beginne, meine Applikation zu optimieren. So kann ich zu einem späteren Zeitpunkt jederzeit das Ergebnis meiner Arbeit überprüfen.



Snapshot-Selektion

7.4.4
Seite 10

Zusätzlich muss ich noch einen „Snapshot"-Namen wählen und den Speicherort für den Snapshot bestimmen. Die erstellte Momentaufnahme finde ich anschließend im angegebenen Schema in der Datei „QZG…". Sie können diese Datei sichern, aber auch auf andere System i Server exportieren. Gleichzeitig finden Sie die Snapshots im Menüpunkt „Momentaufnahmen für SQL-Planungscache" des System i Navigators wieder:

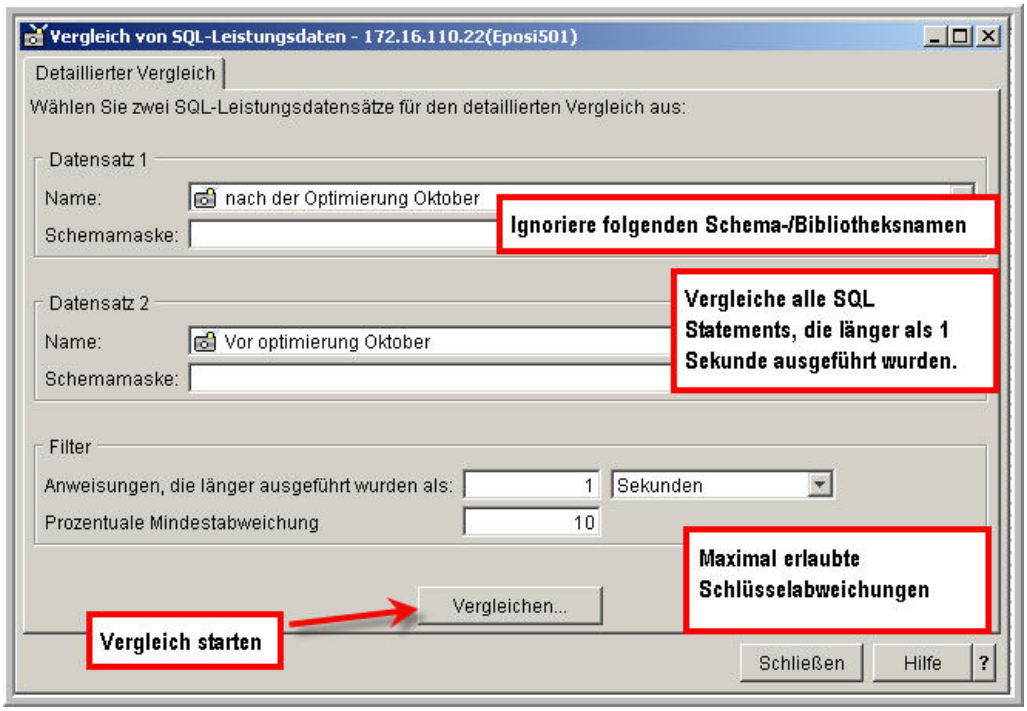


Der Snapshot

Das Kontextmenü der „Snapshots" bietet uns vielfältige Möglichkeiten. Wir können

- die Daten analysieren,
- zugehörige SQL-Anweisungen anzeigen lassen,
- zwei Momentaufnahmen miteinander vergleichen,
- die Aufnahme umbenennen oder löschen
- und seit Release V6R1 Kommentare hinzufügen, damit wir auch morgen noch wissen, warum der Snapshot erstellt wurde.

Der Platz ist nicht ausreichend, um alle Funktionen detailliert zu erläutern. Vieles erklärt sich von selbst. Sie sollten es einfach ausprobieren. Ich benutze beispielsweise gern die Funktion „Vergleichen". Die Funktion „Vergleichen" ermöglicht es, zwei Momentaufnahmen einander gegenüber zu stellen. Probieren wir es aus: Zu Testzwecken habe ich bereits zwei Snapshots im Schema ITPCR erstellt, die ich jetzt nutze, um einen Vorher-Nachher-Vergleich durchzuführen. Hierfür öffne ich zunächst das Kontextmenü des Snapshots und wähle die Funktion „Vergleichen". Es ist egal, welchen der beiden Snapshot man als erstes verwendet.



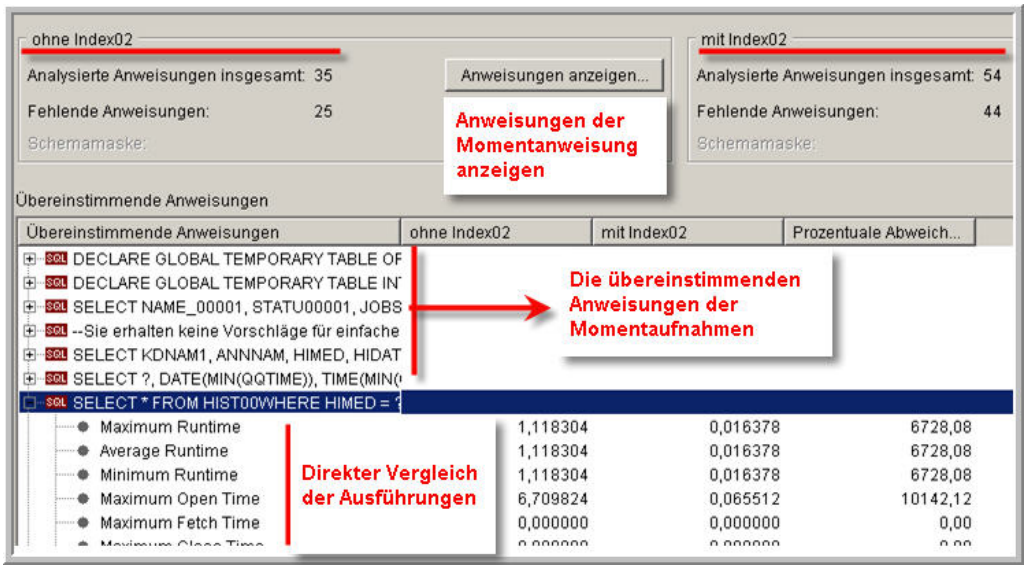
Vergleich der SQL-Leistungsdaten

Sie sehen, dass jetzt noch einige Kriterien benötigt werden. Die Wirkung des Feldes „Schemamaske“ ist am einfachsten anhand eines Beispiels zu erläutern: Stellen Sie sich vor, ich hätte ein Testsystem. Im Schema „Test“ entwickle ich die Applikation und verwende in den SQL Abfragen den Schemanamen zur Qualifizierung. Die Applikation läuft auf dem Produktionssystem im Schema „ITPCR“. Jetzt will ich zwei Snapshots vergleichen. Einer wurde für die Testumgebung erstellt – der zweite für das Produktionssystem. Ein Vergleich könnte aufgrund der abweichenden Schemanamen nicht durchgeführt werden. In diesem Szenario hilft das Feld „Schemamaske“: Für den ersten Snapshot müsste ich festlegen, dass das Schema „Test“ ignoriert werden soll. Im zweiten Eintrag unterbinde ich die Berücksichtigung des Schemanamens „ITPCR“ – und schon kann der Vergleich trotz abweichender Qualifizierung ausgeführt werden.

Zurück zu unserem Beispiel. Ich muss die Vergleichsbedingungen hinterlegen. In meinem Beispiel werden nur SQL-Abfragen miteinander verglichen, deren Ausführung mindestens eine Sekunde dauerte und deren Schlüsselbedingungen um höchstens zehn Prozent voneinander abweichen. Die letzte Bedingung erlaubt damit auch Vergleiche zweier SQL Statements, obwohl beispielsweise unterschiedliche Vergleichsoperatoren verwendet wurden.

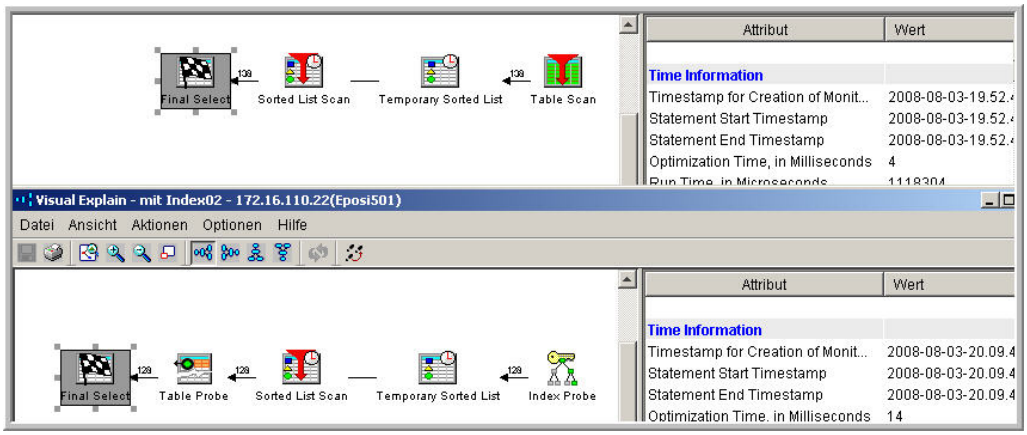
Ergänzung 9/2009

7.4.4
Seite 12



Vergleichsergebnisse

Die Ergebnisse des Snapshot-Vergleichs sind übersichtlich; Sie ermöglichen Ihnen, unterschiedliche Laufzeitergebnisse schnell zu erfassen. Durch Doppelklick auf einen der Ergebnissätze werden Ihnen sogar die unterschiedlichen Ausführungspläne für die Analyse bereitgestellt.



Vergleichende Darstellung der Ausführungspläne

Der Snapshot-Vergleich ermöglicht es, auf komfortable Weise einen „Vorher-Nachher-Vergleich“ durchzuführen, wenn Sie

- Ihre Applikation modifizieren,
- neue Indizes erstellen oder bestehende löschen,
- neue PTFs (Fixes) installieren,
- ein neues Release einspielen,
- Systemwert- oder Abfrageoptionsattribute verändern,
- Subsystemdefinitionen ändern,
- Hardware austauschen
- usw.

Ich nutze diese Vergleichsfunktion zu einem späteren Zeitpunkt sicherlich erneut, um die Auswirkungen erstellter MQTs zu analysieren.

IBM hat gut daran getan, den Plan Cache im System i Navigator zu integrieren. Vor Release V5R4 war dieser Bereich stets eine „Black Box“ und man musste einerseits das SQL Statement kennen, das Probleme bereitet, andererseits musste man das Statement ausführen, um es zu analysieren. Beides entfällt, seitdem über den Plan Cache auf die Ausführungspläne zugegriffen werden kann.

Abzuwarten bleibt die Entwicklung in diesem Bereich. Derzeit ist die SQE leider noch nicht in der Lage, die Informationen im Plan Cache mit der realen Ausführung zu vergleichen. Dies könnte eine Vision sein: ein selbst lernender SQE Query Optimizer. Dieser wäre in der Lage, festzustellen, dass die tatsächliche Ausführungszeit länger dauert als erwartet. Ein selbst lernender Optimizer könnte dann vielleicht selbstständig eine bessere Implementierungsstrategie finden.

Ergänzung 9/2009

7.4.4
Seite 14

7.4.5 Die SQL Workbench/J – Eine Alternative zum System i Navigator?

Immer wieder werde ich gefragt, ob es nicht eine Alternative zur SQL-Umgebung des System i Navigator gibt, um SQL-Statements zu entwickeln oder auszuführen? Und obwohl ich selbst gern und ohne Probleme mit dem Navigator arbeite, nutze auch ich hin und wieder ein Alternativprodukt: Die SQL Workbench/J von Thomas Kellerer.

Die Workbench ist ein kostenloses, datenbankunabhängiges DBMS-Tool, das in Java geschrieben wurde. Das Freeware-Tool bietet einige Features, die wir auch im System i Navigator finden, verfügt aber auch über Funktionen, die im System i Navigator fehlen. Der Schwerpunkt dieses Tools liegt in der Skriptausführung – interaktiv und im Batchmodus. Dabei kann die SQL Workbench/J beispielsweise problemlos mit BLOB-Daten arbeiten und enthält einige individuelle Java-Funktionen, die die Definition und Verwendung von Variablen auch in interaktiven SQL-Abfragen ermöglichen und hat zudem starke Import- und Export-Funktionen. Allerdings wird vorausgesetzt, dass Sie die SQL-Syntax sicher beherrschen, da grafische Hilfsmittel zum Aufbau der SQL-Abfragen fehlen und auch nicht geplant sind.

7.4.5
Seite 2

7.4.5.1 Download und Installation der SQL Workbench/J

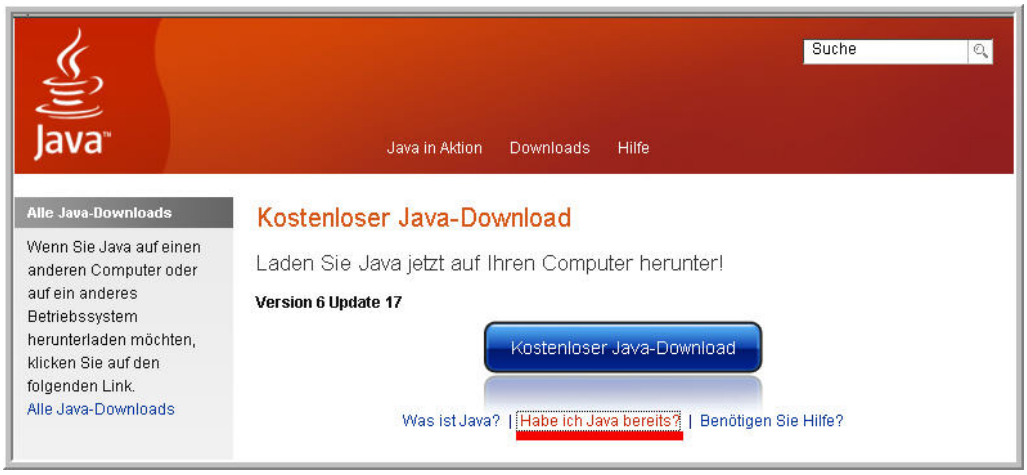
Doch beginnen wir der Reihe nach und klären zunächst einmal die Fragen:

„Woher bekommen wir die SQL Workbench\J?“ und „Wie wird sie installiert und eingerichtet?“

Wenn Sie – wie ich es getan habe – die aktuelle Version 107 vom 16.12.2009 herunterladen wollen, sollten Sie vorab die auf dem PC installierte Java-Version überprüfen. Die Workbench benötigt die Java-Version 6, um korrekt zu arbeiten. Sie haben zwei Möglichkeiten, Ihre installierte Java-Version zu prüfen:

Entweder öffnen Sie auf Ihrem PC die Systemsteuerung, klicken auf das Icon „Java“ und lassen sich die Anwendungsinformationen anzeigen

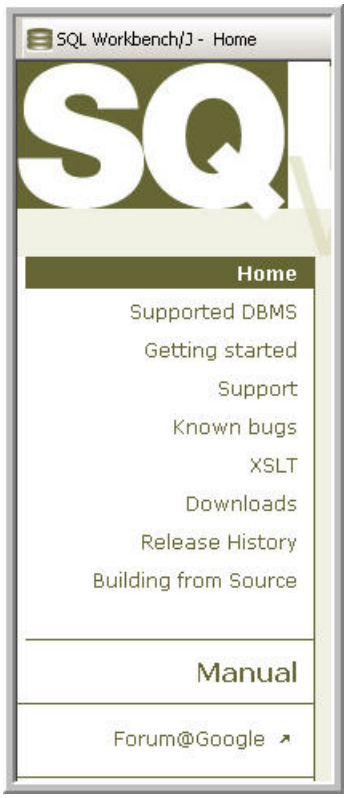
oder Sie besuchen die Java-Homepage unter http://www.java.com/de/download/index.jsp



Java-Homepage

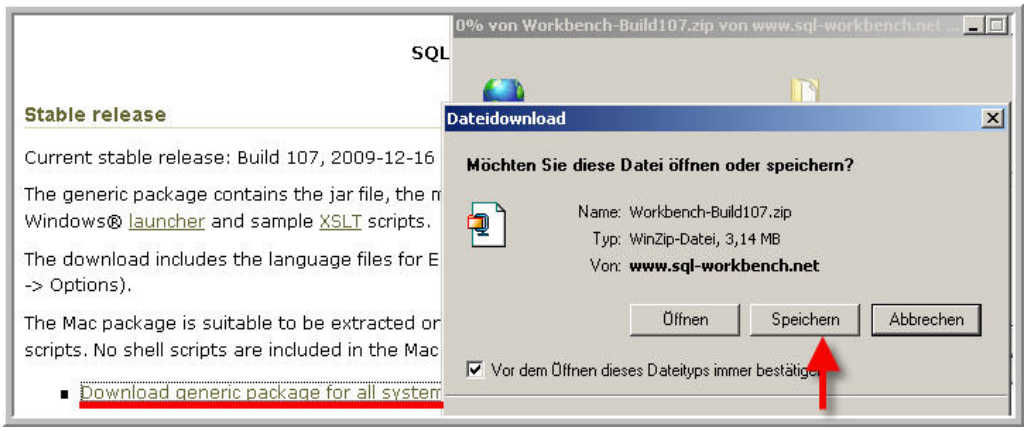
Hier können Sie Ihre installierte Java-Version überprüfen lassen und ggf. die aktuelle Version 6 gleich herunterladen. Erst wenn Java 6 auf Ihrem PC installiert ist, können Sie die SQL Workbench installieren.

Informationen zum Produkt und das Downloadangebot finden Sie auf der Homepage von Thomas Kellerer: http://www.sql-workbench.net



Die Homepage

Klicken Sie auf „Downloads“ und laden Sie die Version „Downloads generic package for all systems“ herunter.

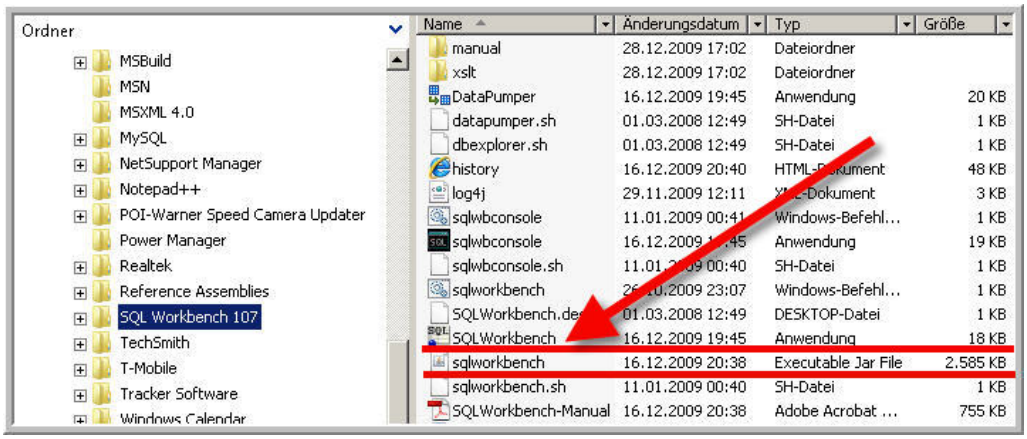


Der Download

Das ZIP-File können Sie auf einem beliebigen Laufwerk Ihres PCs speichern. Anschließend entpacken Sie das ZIP-File in ein beliebiges Verzeichnis. Die SQL Workbench/J ist jetzt bereit; weitere Installationsschritte sind nicht erforderlich. Eine angenehme Eigenschaft, wie ich finde.

Wenn Sie Windows benutzen, können Sie die Workbench jetzt starten, indem Sie im Installationsverzeichnis auf die ausführbare JAR-Datei sqlworkbench doppelt klicken oder das EXE-File SQL Workbench ausführen.

7.4.5
Seite 4

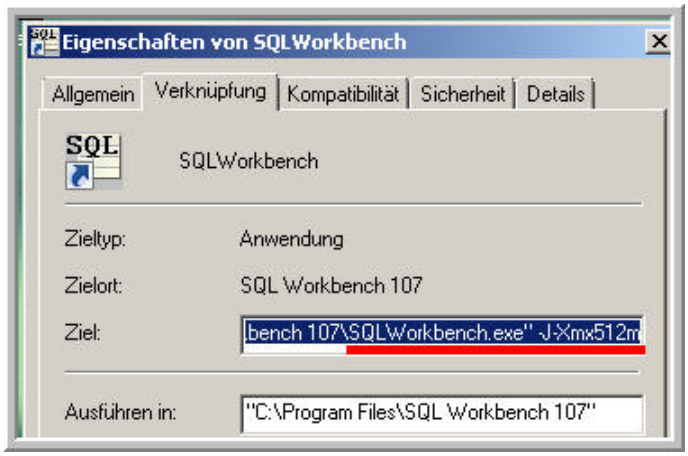


Inhalt des Installationsverzeichnisses

Am einfachsten ist es sicherlich, eine Desktopverknüpfung für die ausführbare EXE-Datei zu erstellen. Machen wir an dieser Stelle einen kurzen Ausflug: Obwohl wir zu diesem Zeitpunkt noch keinen SQL-Zugriff realisiert haben, sollten wir uns dennoch ein wenig Gedanken um den Arbeitsspeicher machen. Die SQL Workbench/J ist eine Java-Applikation und läuft daher in einer virtuellen Java-Umgebung (JVM). Der verfügbare Arbeitsspeicher für die Workbench ist somit einerseits vom verfügbaren Speicher unseres PCs abhängig, andererseits von der Java-Umgebung. Der zur Verfügung stehende Arbeitsspeicher ist aber wichtig, da die SQL Workbench/J die Daten später direkt in den Hauptspeicher unseres PCs einliest und vielleicht bekommen Sie dann bei einigen Abfragen – die große Result Sets abrufen – , die Meldung, dass zu wenig Arbeitsspeicher für die Ausführung der SQL-Abfrage zur Verfügung steht. Für diesen Fall können Sie entweder die SQL-Abfrage dahingehend ändern, dass weniger Daten abgerufen werden, oder Sie müssen der Java-Umgebung mehr Arbeitsspeicher zur Verfügung stellen, indem Sie die SQL Workbench/J mit folgendem Parameter aufrufen:

```
SQLWorkbench.exe -J-Xmx512m
```

In meinem Beispiel wird der maximale Arbeitsspeicher auf 512MB erhöht – normalerweise liegt dieser Maximalwert für die Java-Umgebung zwischen 64MB und 256MB Speicher abhängig vom Aufruf und der individuellen Konfiguration Ihres PCs. Wenn Sie sich entschließen sollten, den für die JVM maximal verfügbaren Arbeitsspeicher zu erhöhen, können Sie die Parameter auch direkt in den Eigenschaften Ihrer Desktop-Verknüpfung hinterlegen.



Eigenschaften der Desktop-Verknüpfung

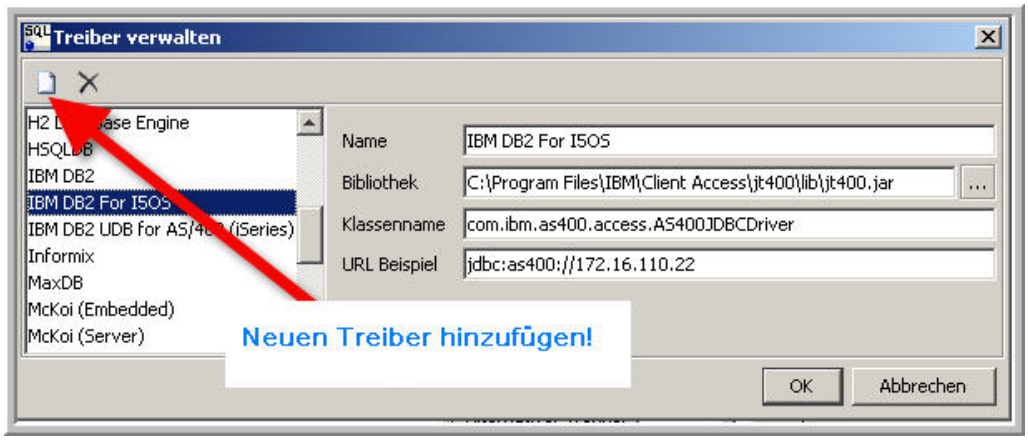
Eine Anmerkung zum Schluss: Der Parameter –Xmx bestimmt lediglich den Maximalwert. Das heißt nicht, dass die SQL Workbench/J grundsätzlich 512MB Arbeitsspeicher beansprucht.

7.4.5.2 Konfiguration des JDBC-Treibers

Nun wollen wir die SQL Workbench/J starten. Führen Sie hierfür die JAR-Datei oder die EXE-Datei aus. Im anschließenden initialisierenden Dialogfenster muss beim ersten Aufruf bekannt gegeben werden, wo der JDBC-Treiber für die Datenbankverbindung zu finden ist.

JDBC-Treiber konfigurieren

Dazu klicken Sie im Eröffnungsbildschirm auf den Button „Treiber verwalten". Daraufhin öffnet sich ein weiterer Dialog:



Treiberinformationen hinterlegen

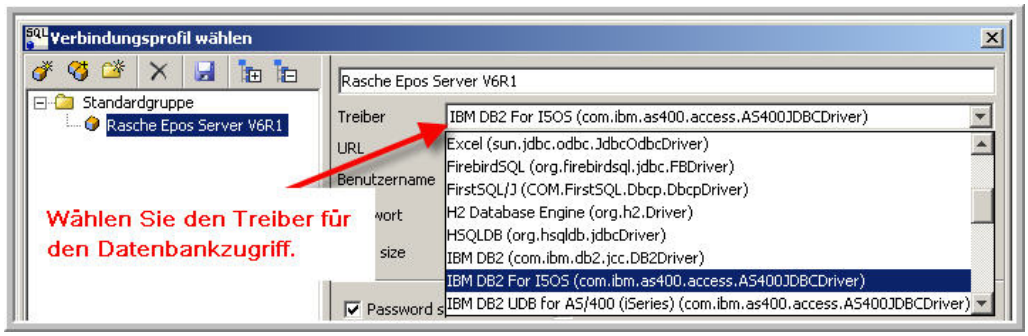
Im rechten Fenster finden Sie eine Reihe vordefinierter Treiber mit den entsprechenden JAR-Dateien. Ich weiß nicht mehr genau, ob die Treiberinformationen für die DB2 for i5/OS auch vorinstalliert waren, falls nicht, fügen Sie einfach einen neuen Treiber hinzu. Sie können für den Eintrag einen beliebigen Namen verwenden – nur sprechend sollte er sein! Anschließend selektieren Sie das JAR-File auf Ihrem PC. Sowohl Client Access für Windows als auch WDSC und RDI enthalten einen JDBC-Treiber, den Sie nutzen könnten. Sobald Sie den Speicherort hinterlegt haben, wird der Klassenname automatisch eingetragen, so dass Sie nur noch die URL Ihres System i Servers eintragen müssen. Nachdem diese Informationen erfasst sind, können Sie diesen Dialog mit „OK" bestätigen und anschließend schließen.

7.4.5.3 Konfiguration der Verbindungsprofile

Als nächstes müssen wir ein Verbindungsprofil erstellen.

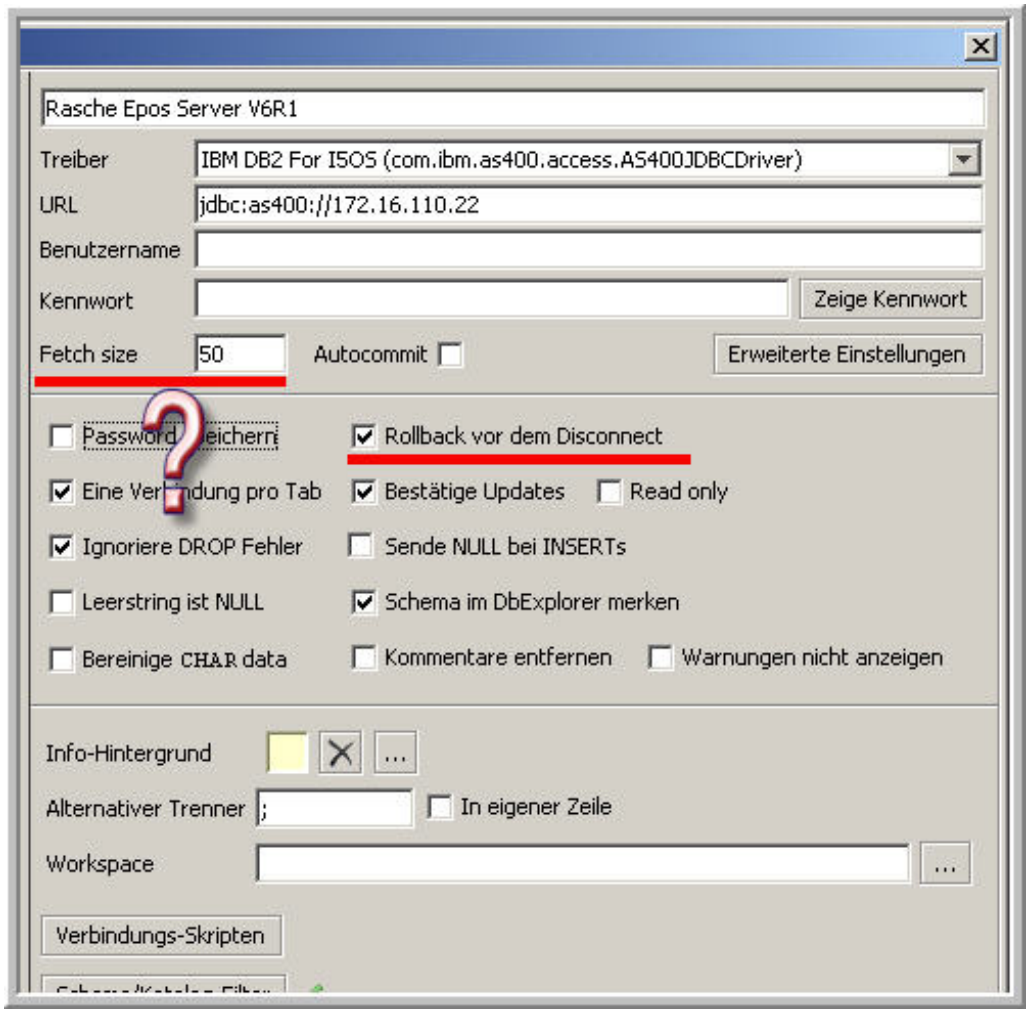
Verbindungsprofile erstellen

In dem Verbindungsprofil hinterlegen wir detaillierte Informationen für den Datenbankzugriff. Als erstes wählen wir den soeben lokalisierten JDBC-Treiber aus:



Verbindungsprofile erstellen – Treiber wählen

Sie können für die Verbindung einen Benutzernamen und ein Kennwort hinterlegen. Tun Sie es nicht, erfolgt ein separater Anmeldedialog. Ich verzichte meistens auf diese Einstellungen, um unbefugte Anmeldungen zu verhindern.



Einstellungen Verbindungsprofil

Schauen wir uns jetzt einige der weiteren Parameter an. Allerdings beschränke ich mich hierbei auf die wichtigen Optionen. Detaillierte Informationen enthält das Handbuch:

Autocommit
Wenn Sie die Option AUTOCOMMIT aktivieren, wird jedes DDL- und DML-Statement auf dem System i Server automatisch festgeschrieben (commit). Andernfalls müssen Sie selbst Ihre INSERT-, UPDATE-, DELETE- und CREATE-Statements explizit festschreiben, wenn Sie die Veränderungen in der Datenbank nach Beendigung der Sitzung übernehmen wollen.

Passwort speichern
Ist diese Einstellung aktiviert, wird das Passwort für das Benutzerprofil in einer XML-Datei auf Ihrem PC gespeichert. So lange Sie nicht in den globalen Optionen der SQL Workbench die Passwortverschlüsselung aktiviert haben, ist das Passwort in Klarschrift gespeichert. Schauen Sie, was gemeint ist:

```
<void property="name">
 <string>LGADOZO2</string>
</void>
<void property="password">
 <string>HAUSTIER</string>
</void>
<void property="storeExplorerSchema">
<void property="url">
<void property="useSeparateConnectionPerTab">
<void property="username">
 <string>LGADOZO2</string>
```

Wirkungsweise der Option „Passwort speichern“

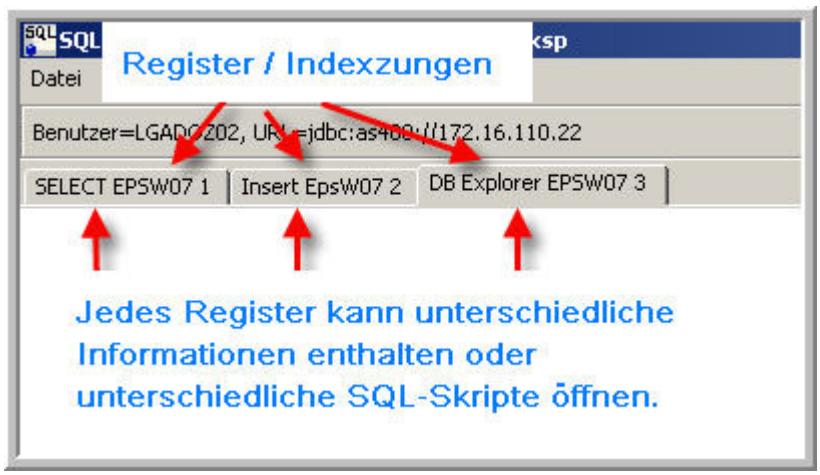
Sie sollten also das Passwort besser nicht speichern oder zuvor die Verschlüsselung aktivieren. Hierzu müssen Sie nach dem Start der SQL Workbench den Menüpunkt „Extras ➜ Optionen ➜ Allgemein“ aufrufen und die Passwortverschlüsselung aktivieren. Wenn Sie dann Ihr Passwort speichern, ist es verschlüsselt:

```
<void property="name">
 <string>LGADOZO2</string>
</void>
<void property="password">
 <string>@*@#-1#15#-33#-9#-21#-108#-31#83</string>
```

Option Passwort speichern – verschlüsselt

7.4.5
Seite 10

Eine Verbindung pro Tab
Innerhalb der SQL-Workbench können Sie verschiedene Datenbankskripte und mehrere DB-Explorer öffnen. Dargestellt werden diese Verbindungen auf unterschiedlichen Registern:



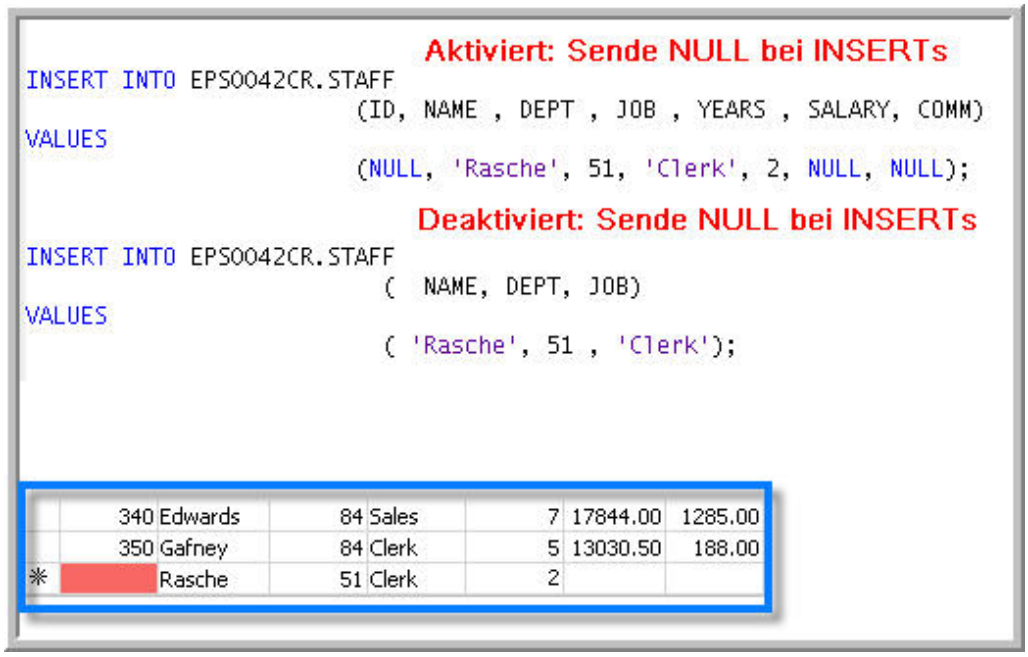
Wirkungsweise der Option „Eine Verbindung pro Tab“

Wenn Sie die Option „Eine Verbindung pro Tab“ aktivieren, wird für jedes Register eine separate Verbindung aufgebaut und auf dem System i Server ein eigener Serverjob genutzt. Andernfalls teilen sich die Register eine Verbindung, mit der Folge, dass es sich dann auch nur um einen Job handelt und Datenbanküberschreibungen, offene Transaktionen etc. für alle Register gleichermaßen gelten. Sie müssen sich nicht jetzt entscheiden, welches die bessere Wahl ist, denn die Optionen sind jederzeit änderbar und außerdem könnten Sie für unterschiedliche Aufgaben und damit verbundenen unterschiedlichen Anforderungen verschiedene Verbindungsprofile erstellen.

Leerstring ist NULL
Die SQL Workbench erlaubt die Bearbeitung von Result Sets. Die auf diesem Weg vorgenommenen Änderungen können anschließend auf den Server übertragen werden. Setzen Sie dabei eines der Felder im Result Set auf *BLANK, wird der Wert NULL übertragen, wenn Sie diese Option aktivieren. Da die meisten unserer Tabellen aber keine NULL-Werte zulassen, könnte es sinnvoll sein, diese Option zu deaktivieren.

Sende NULL bei INSERTs

Um die Wirkungsweise dieser Option zu verstehen, schauen wir uns das nachfolgende Beispiel an:



Wirkungsweise der Option Sende NULL bei Inserts

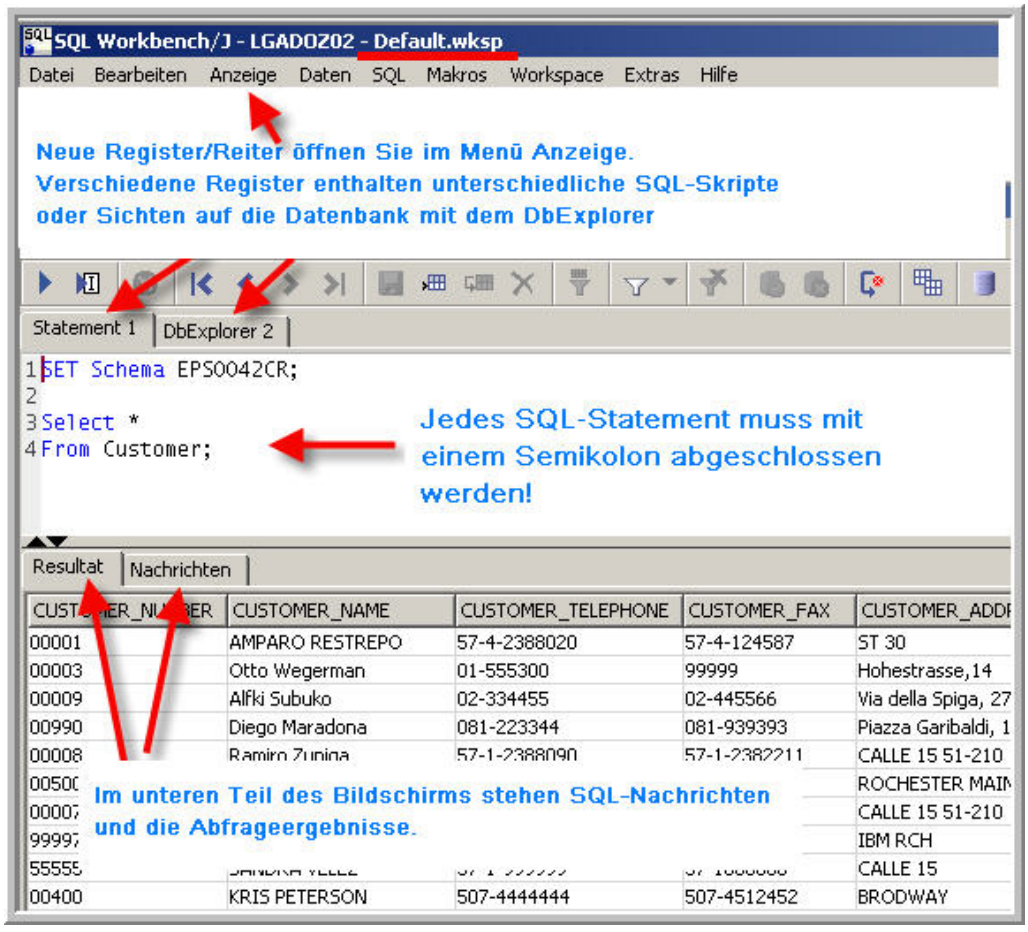
In dem SQL-Skript habe ich im Result Set eine neue Zeile eingefügt. Anschließend lasse ich mir ein SQL-Skript mit den entsprechenden Insert-Statements generieren. Wenn die Option „Sende NULL bei INSERTs“ aktiviert ist, werden Spalten, die keine Werte enthalten, in der Value-Liste des Insert-Statements berücksichtigt und erhalten NULL-Werte. Ist die Option aber deaktiviert, bleiben die leeren Spalten im Insert-Statement unberücksichtigt. Die Option kann später einmal sehr nützlich sein, wenn Identity-Columns in den Tabellen vorhanden sind, die nicht in der Value-Liste eines Insert-Statements angesprochen werden dürfen!

7.4.5
Seite 12

Workspace

Abschließend müssen wir noch den Parameter Workspace klären. Es kann und sollte für jedes Benutzerprofil ein eigener Workspace generiert werden. Ein Workspace ist eine Arbeitsumgebung. Diese Arbeitsumgebung enthält alle geöffneten Register, beinhaltet auch die ausgeführten SQL-Statements. Wenn Sie den Parameter „Workspace“ ignorieren und leer lassen, wird Ihnen ein Standardarbeitsbereich zugewiesen. Diesen Default-Workspace teilen Sie sich mit allen Benutzern, denen ebenfalls kein eigener Arbeitsbereich zugewiesen wurde. Dies gilt natürlich auch umgekehrt: Alle anderen Benutzer können dann auch Ihre SQL-Abfragen lesen und – bei entsprechender Berechtigung für die Datenbank – ggf. auch ausführen. Solange Sie mit der SQL Workbench lokal arbeiten und sich mit niemandem den PC teilen, sollte dies eigentlich kein Problem darstellen. Außer Sie selbst arbeiten mit unterschiedlichen Profilen, um verschiedene Aufgaben wahrzunehmen. Besser Sie erstellen einen individuellen Workspace – entweder wenn Sie das Profil erstellen oder später in Ihrer ersten Sitzung, wie ich es gleich tun werde.

Nun ist es an der Zeit sich anzumelden und eine erste Verbindung zur Datenbank herzustellen. Wenn das Profil ordnungsgemäß erstellt wurde, sollten Sie jetzt das Hauptfenster der SQL Workbench/J sehen:

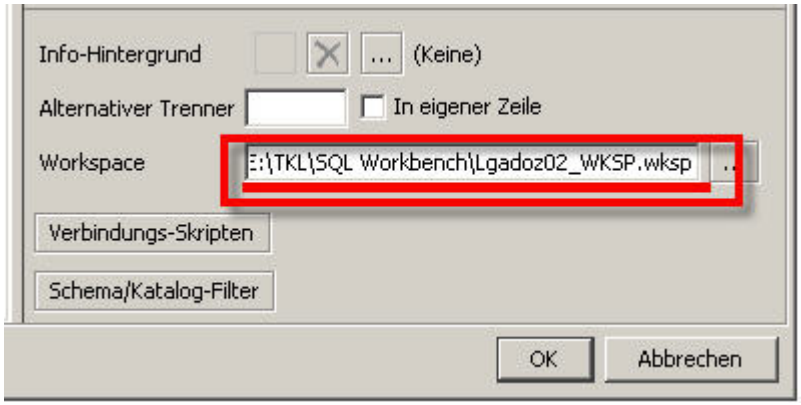


Hauptfenster der SQL Workbench

Das Hauptfenster besteht aus einer Menüleiste und einer Toolbar, die viele Menüpunkte als Shortcuts beinhaltet. Weiter unterteilt sich das Hauptfenster zunächst in zwei Register. Das Register „Statement 1“ ist für die Ausführung Ihrer SQL-Statements zuständig. Im Register „DbExplorer“ können Sie sich Tabellen und andere Datenbankobjekte einer beliebigen Bibliothek anzeigen lassen. In vielen Dingen ist die SQL Workbench mit dem System i Navigator vergleichbar. Vieles ist außerdem selbsterklärend und braucht keine weiteren Erläuterungen.

Ich möchte zum Abschluss dieses Beitrages nur noch meine Arbeitsumgebung speichern und meinem Profil zuweisen. Dazu öffne ich zunächst in der SQL Workbench das Menü „Workspace ➜ Workspace speichern als“. Alle Arbeitsumgebungen werden mit der Endung WKSP in einem beliebigen Verzeichnis gespeichert. Anschließend werden Sie automatisch gefragt, ob Sie diese Arbeitsumgebung dem aktuellen Profil zuweisen wollen. Diesen Dialog bestätige ich mit „JA“. Wenn Sie später Ihre Sitzung beenden, werden Sie noch gefragt, ob Sie die Veränderung an den Profilen speichern wollen. Auch jetzt bestätigen Sie wieder den Dialog mit „Ja“.

Bei einer erneuten Anmeldung unter dem Profil sollte der soeben zugewiesene Arbeitsbereich automatisch wieder geöffnet werden:



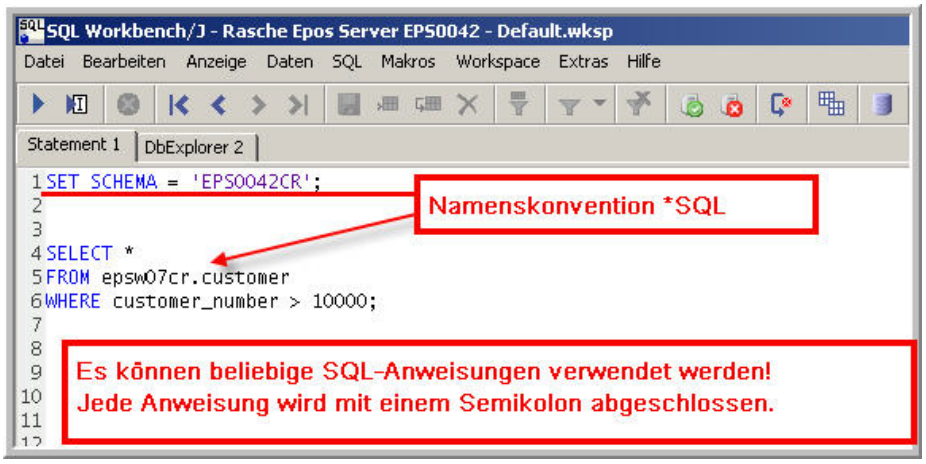
Individueller Workspace

Die SQL Workbench/J ist eingerichtet, wir haben ein Profil erstellt und dem Profil eine individuelle Arbeitsumgebung zugewiesen.

7.4.5.4 SQL Workbench/J – Ein Überblick der Skriptumgebung

Nachdem wir die SQL Workbench/J installiert und uns bereits ein wenig mit der Oberfläche vertraut gemacht haben, wollen wir uns jetzt einige grundlegende Funktionen der Skriptumgebung anschauen. Da wir bereits mit dem System i Navigator gearbeitet haben, sind wir mit der Bedienung grafischer Oberflächen vertraut. Aus diesem Grund werde ich mich bei der SQL Workbench/J auf wichtige Grundfunktionen beschränken, die insbesondere den Zugriff auf die DB2 for i5/OS betreffen und einige Spezifika vorstellen, die in dieser Form dem System i Navigator fremd sind. So können Sie beispielsweise Variablen für die interaktiven Abfragen deklarieren und auch problemlos mit CLOB- und BLOB-Feldern arbeiten.

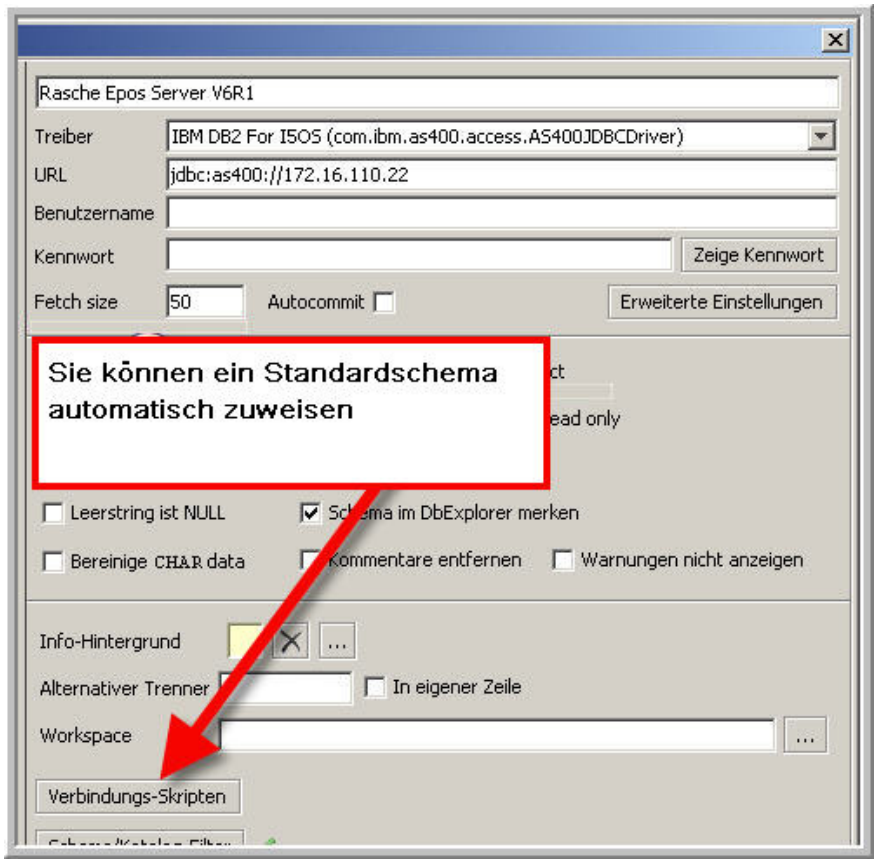
Doch beginnen wir erst einmal mit dem Hauptfenster der SQL-Workbench/J. Es besteht aus verschiedenen Registern: einerseits sollten Sie den DbExplorer sehen, andererseits sollte es eine Skriptumgebung geben. Schauen wir uns in einem ersten Schritt die Skriptumgebung etwas genauer an:



SQL-Statement ausführen

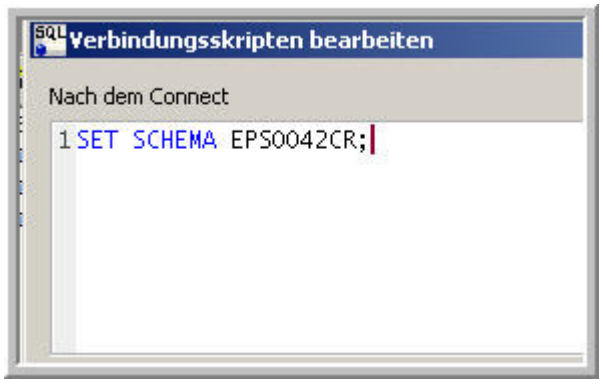
Grundsätzlich können Sie natürlich alle SQL-Statements ausführen. Sie müssen lediglich darauf achten, jedes Statement mit einem Semikolon abzuschließen. Vergessen Sie das Semikolon, sind die Fehlermeldungen häufig innerhalb der SQL Workbench/J irreführend – also prüfen Sie im Fehlerfall immer zuerst, ob die SQL-Statements korrekt begrenzt wurden. Außerdem handelt es sich bei der SQL Workbench/J um ein datenbankunabhängiges Tool, das dem SQL-Standard folgt. Daraus resultiert, dass Sie mit der Namenskonvention SQL arbeiten müssen. Das wiederum bedeutet, dass Sie nicht mit der Bibliotheksliste arbeiten können, sondern nur ein aktuelles Schema setzen können – ähnlich einer CURRENT-Library. Verwenden können Sie hierfür das SQL-Statement SET SCHEMA wie in der oberen Abbildung gezeigt. Außerdem dient der Punkt als Trennzeichen für die qualifizierte Objektadressierung und nicht der Slash – wie Sie es vielleicht gewohnt sind.

Sie können das aktuelle Schema auch automatisch über den Anmeldedialog zuweisen. Öffnen Sie dazu im Anmeldefenster den Dialog „Verbindungs-Skripten“, wie nachfolgend gezeigt.



Verbindungs-Skripten

Daraufhin öffnet sich ein SQL-Editor:

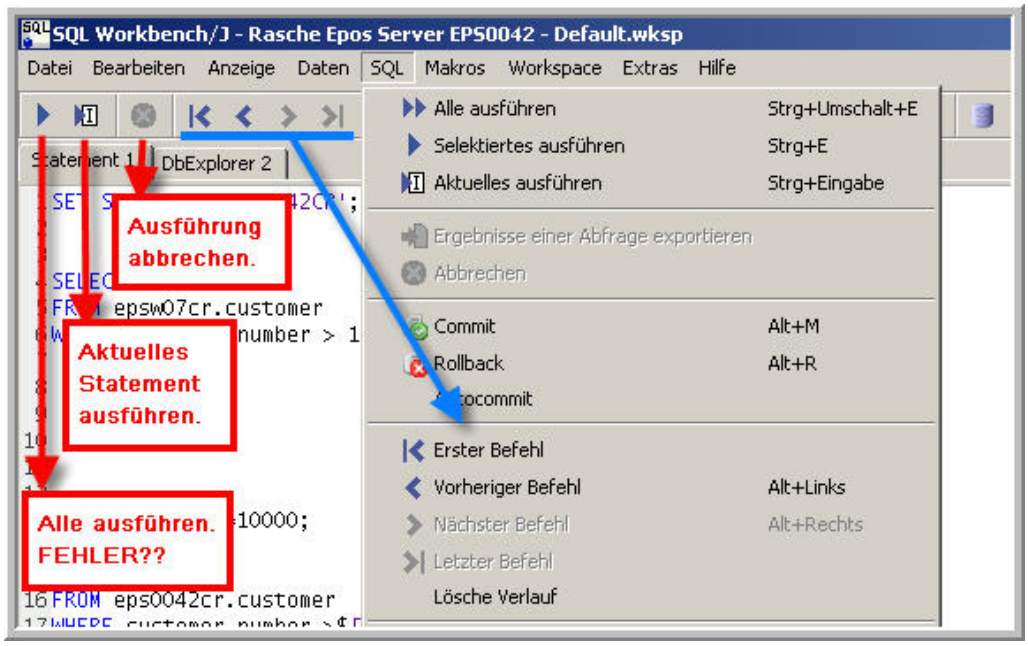


Verbindungsskripte erstellen

7.4.5
Seite 16

Mit Hilfe dieses Editors können Sie SQL-Skripte hinterlegen, die unmittelbar nach der Anmeldung ausgeführt werden. Das ist vergleichbar mit unseren CL-Startprogrammen, die wir in den Benutzerprofilen hinterlegen. Das SQL-Skript kann zwar mehrere SQL-Statements enthalten, aber SQL kennt weiterhin nur eine aktuelle Arbeitsbibliothek!

Schauen wir uns nun kurz die Toolbar und das Menü „SQL“ an:



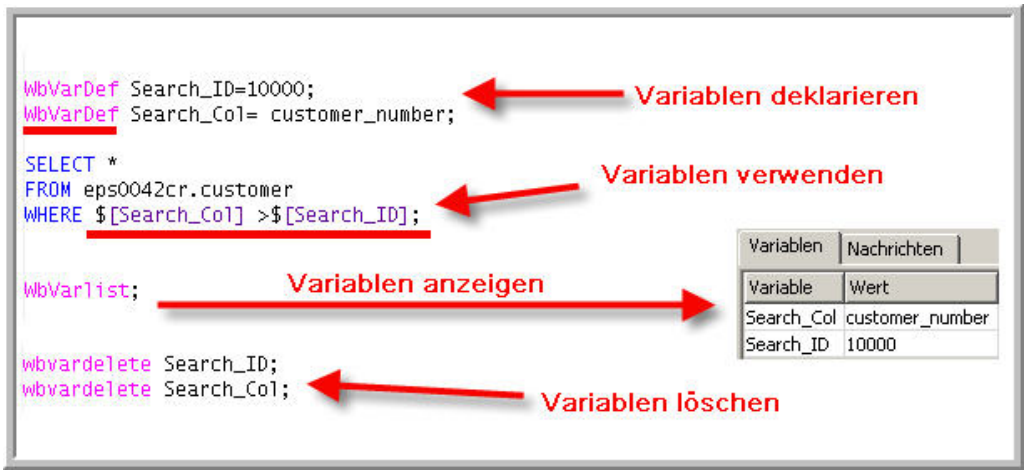
Toolbar und das Menü „SQL“

Die ganz links stehenden Icons der Toolbar sind für die Ausführung der SQL-Statements zuständig. In meiner installierten Version ist es derzeit so, dass eines der Icons fehlt und daher nur die Auswahl zwischen „Alle ausführen“ und „Aktuelles ausführen“ besteht.

Eine nette Eigenschaft der Skriptumgebung ist die Verlaufsaufzeichnung. Alle Veränderungen innerhalb Ihres Skriptes werden dokumentiert und daher ist es problemlos möglich innerhalb der Änderungen zu „blättern“. Probieren Sie die Funktion einfach aus; Sie werden die Vorteile schnell erkennen.

7.4.5.4.1 SQL Skriptumgebung – Variablen deklarieren

Die SQL Workbench/J erlaubt auch die Verwendung von Variablen im SQL-Skript; ein Feature, das der Prozedurumgebung des System i Navigators fremd ist.



Variablen verwenden

Möglich wird die Verwendung von Variablen durch integrierte Java-Funktionen.

Ich verwende im oben dargestellten Skript die integrierten Funktionen WbVarDef und WbVarDelete, um zwei Variablen zu deklarieren. Der ersten Variablen weise ich den Wert 10000 zu. Die zweite Variable enthält den Spaltennamen der WHERE-Bedingung. Bei der Verwendung von Variablen müssen Sie lediglich beachten, dass die Variablennamen case-sensitive sind. Sie müssen also auf korrekte Groß- und Kleinschreibung achten.

Die Workbench/J verfügt über viele individuelle Funktionen, die natürlich nicht zum SQL-Standard gehören und daher auch nur im Umfeld dieser SQL Workbench/J eingesetzt werden können. Ich werde in meinen Beiträgen sicherlich nicht alle Funktionen vorstellen, denn schließlich will ich nicht das Handbuch ersetzen. Sie sollten sich also bei Gelegenheit einfach selbst einen Überblick verschaffen: Das Handbuch enthält eine Beschreibung aller integrierten Funktionen.

7.4.5
Seite 18

Kommen wir aber wieder zu unseren Variablen zurück, die wir jetzt im SQL-Statement nutzen wollen. Verwenden können Sie die Variablen, indem Sie die Variable in eckige Klammern einschließen und ein $-Zeichen voranstellen. Die SQL Workbench/J expandiert daraufhin das SQL-Statement entsprechend, bevor die Abfrage zum Server geschickt wird. Sie können sogar die Variableninhalte zur Laufzeit des Statements bestimmen, wenn Sie dem Variablennamen ein Fragezeichen voranstellen:

Selektive Bedienerführung

Wir kennen dieses Verhalten aus der CL-Programmierung und nennen es selektive Bedienerführung. Testen Sie einfach mal den Einsatz von Variablen. Ich bin überzeugt, dass Ihnen sehr schnell viele Verwendungsmöglichkeiten einfallen.

7.4.5.4.2 SQL Skriptumgebung – Arbeiten mit BLOB-Daten

Die SQL Workbench/J unterstützt auch die Arbeit mit BLOB- oder CLOB-Spalten. Für Insert- und Update-Operationen können die ganz normalen SQL-Statements verwendet werden, wobei lediglich ein spezieller Platzhalter verwendet wird.

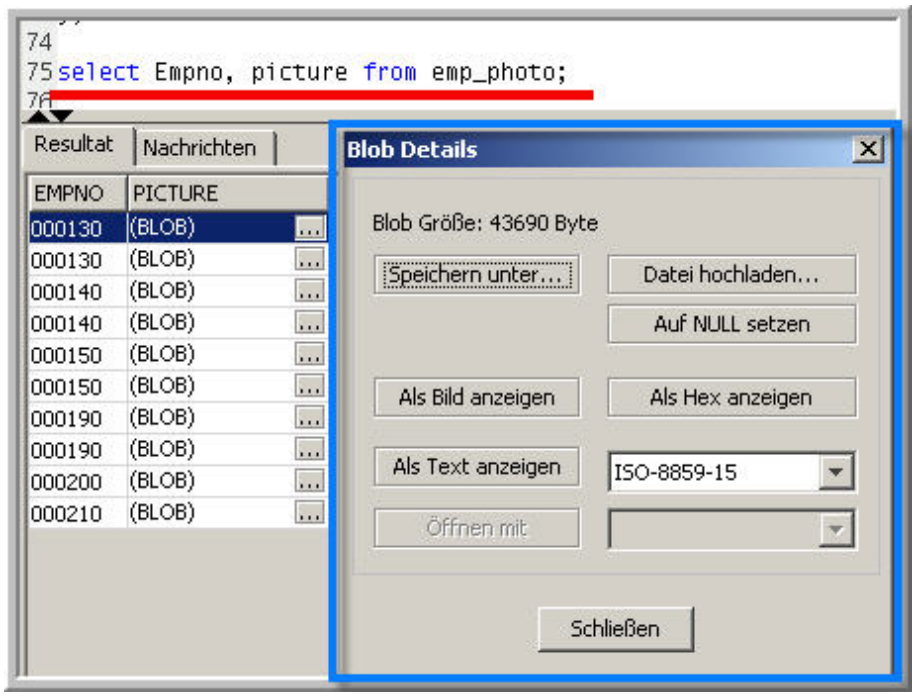
Schauen wir uns hierzu zwei Beispiele an.

```
UPDATE emp_photo
       SET picture = {$blobfile = E:\Toucan.jpg}
WHERE Empno = '000200';


INSERT INTO emp_photo
(
 Empno,
 photo_format,
 picture
)
VALUES
(
 '000210',
 'JPG',
 {$blobfile=E:\Toucan.jpg});
```

Um BLOB-Spalten zu verändern oder einzufügen, müssen Sie lediglich den Platzhalter {$blobfile = Pfadangabe} nutzen. Handelt es sich bei der Tabellenspalte um den Datentypen CLOB (Character Large Object), verwenden Sie hingegen den Platzhalter {$clobfile = Pfadangabe}. Die SQL Workbench/J schreibt das Insert- bzw. Update-Statement entsprechend um und sendet die Daten der im angegebenen Verzeichnis lokalisierten Datei zur Datenbank.

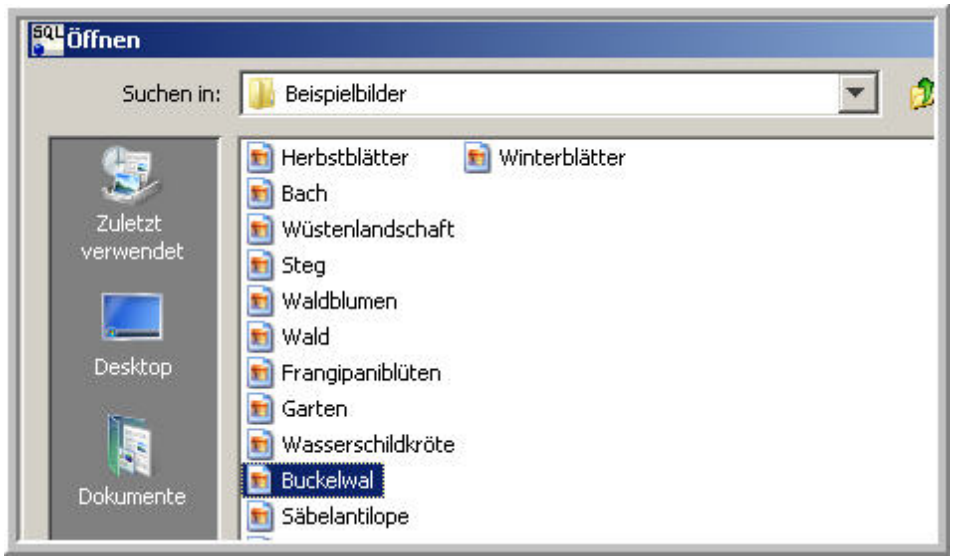
7.4.5
Seite 20

Wir haben mit den vorangegangenen SQL-Statements zum einen dem Mitarbeiter mit der Personalnummer 200 ein neues Bild zugewiesen, andererseits haben wir für einen Mitarbeiter zusätzliche Informationen erfasst. Wenn wir jetzt die Tabelle EMP_PHOTO selektieren, enthält das Result Set BLOB-Spalten. Die BLOB-Daten werden nicht direkt angezeigt, sondern enthalten ein Icon (=Button), wie folgende Abbildung zeigt:



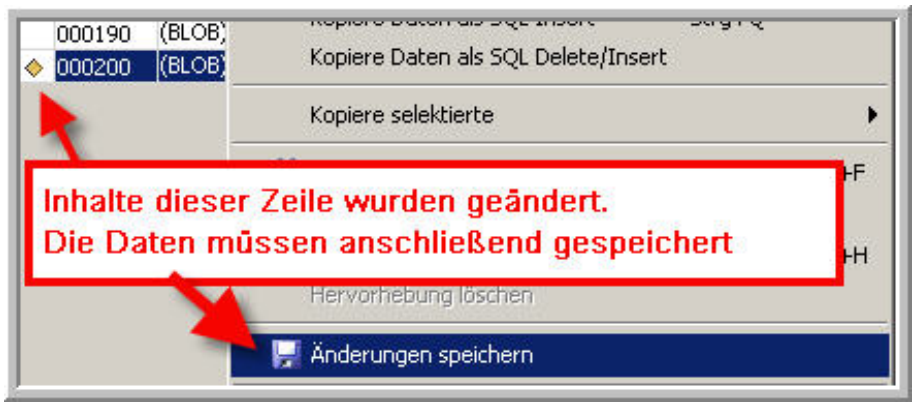
Blob-Spalten anzeigen

Wenn Sie auf das Icon eines Satzes klicken, öffnet sich das zweite Dialogfenster. Sie können die Datei bzw. das Bild über diesen Dialog anzeigen oder auf dem PC speichern. Sie können diesen Dialog auch nutzen, um neue Bilder hochzuladen. Dafür klicken Sie einfach nur auf den Button „Datei hochladen“ und suchen das Bild oder die Datei, die gespeichert werden soll.



Neue Bilder hochladen

Der veränderte Satz im Result Set wird automatisch markiert, so dass Sie sehen können, dass Spalteninhalte dieses Satzes verändert wurden. Es ist auch möglich, sich bereits zum jetzigen Zeitpunkt das neue Bild anzeigen zu lassen. Wenn Sie allerdings die Änderungen in der Datenbank festschreiben wollen, müssen Sie über das entsprechende Icon der Toolbar oder über das Kontextmenü die Daten auf dem Server speichern:



Geänderte Daten speichern

7.4.5.4.3 SQL Skriptumgebung – Daten im Result Set verändern

Dieses Verhalten ist neu für uns. Sie können tatsächlich die Daten eines Result Sets direkt bearbeiten und anschließend die Veränderungen auf dem Server speichern.

Damit diese Funktionalität gegeben ist, sollten Sie allerdings zuvor die Optionseinstellungen prüfen. Öffnen Sie hierfür das Menü „Extras Optionen“. Es gibt für die verschiedensten Themenbereiche Optionseinstellungen. So können Sie die Schrifttypen, Farben und viele weitere Details bestimmen. Wir schauen uns jetzt aber nur kurz den Bereich „Daten bearbeiten“ an:

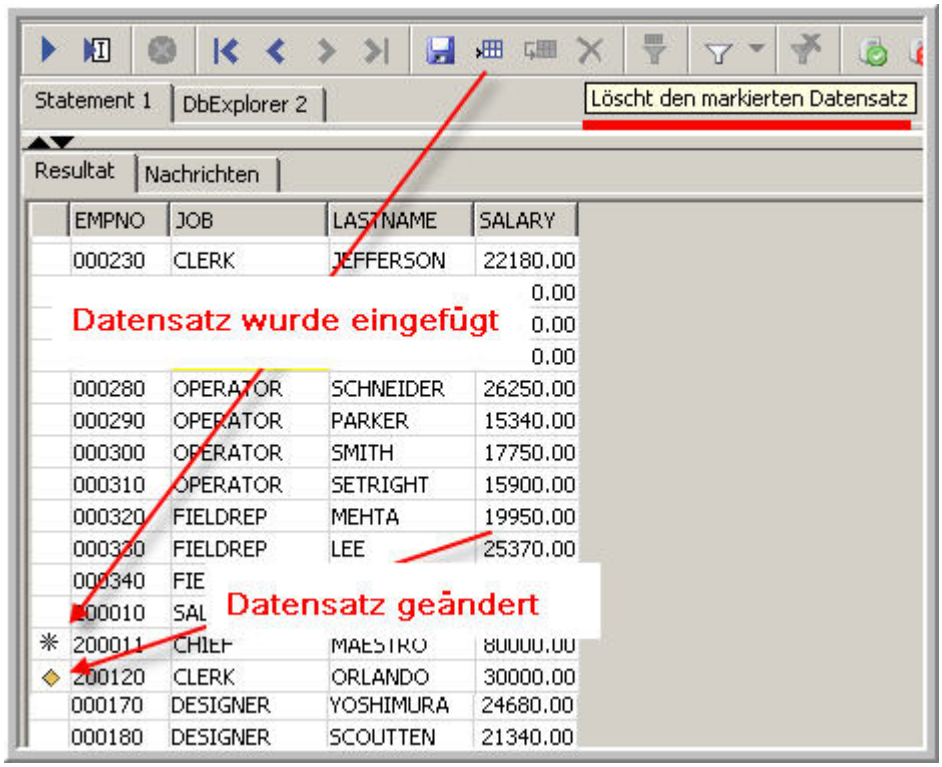
Optionen – Daten bearbeiten

Felder, die keine NULL-Werte enthalten dürfen, werden automatisch rot markiert, und ich möchte außerdem eine Bestätigung erhalten, bevor die Daten auf den Server übertagen werden.

Wenn Sie jetzt die Daten des Result Sets ändern, müssen Sie noch etwas berücksichtigen: Sie müssen die Tabellen qualifiziert ansprechen, ansonsten erhalten Sie stets folgende Fehlermeldung:

Keine Basistabelle gefunden

Sobald diese Minimalanforderungen erfüllt sind, lassen sich die Daten problemlos bearbeiten:



Result Set bearbeiten

Und wenn Sie mehr als eine Tabelle im Result Set haben, müssen Sie über einen zusätzlichen Dialog einfach festlegen, welche der Tabellen geändert werden soll:

Mehr als eine Tabelle im Result Set

Nur Spalten der ausgewählten Tabelle werden anschließend im INSERT-, UPDATE- und DELETE-Statement berücksichtigt. Dabei geht die SQL Workbench/J stets davon aus, dass genügend Informationen im Result Set verfügbar sind, um eindeutig auf den jeweiligen Satz zugreifen zu können. Weitere Informationen sollten Sie dem Handbuch entnehmen, das zwar nur in Englisch veröffentlicht wird; dafür aber gelungen ist.

7.4.5.4.4 SQL Skriptumgebung – Komfortable Skript-Funktionen

Sie sehen, die SQL Workbench/J beinhaltet mächtige Funktionen. Daneben gibt es aber auch weniger spektakuläre Features, die dennoch nützlich sein können. Hier einige Beispiele:

Während der Eingabe des SQL-Statements können Sie die Autovervollständigung nutzen.

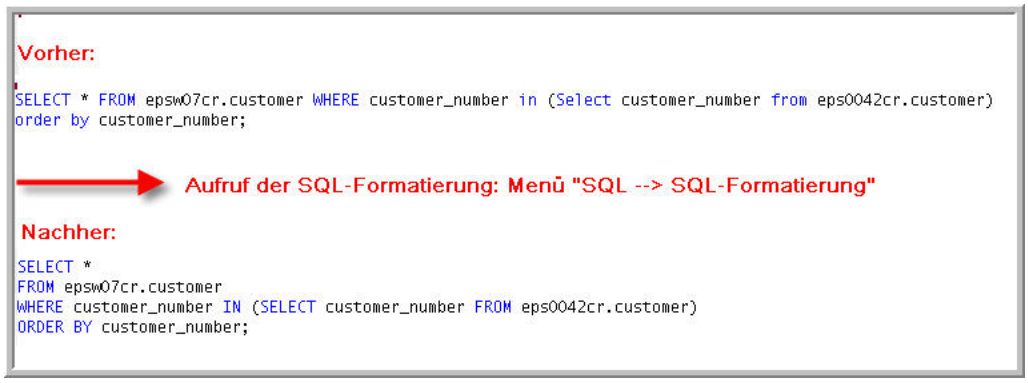
Hierzu geben Sie Ihr SQL-Statement wie gewohnt ein und nutzen dann aber die Tastenkombination STRG+Leertaste, um sich beispielsweise die Tabellen innerhalb eines Schemas anzeigen zu lassen oder die Spalten der ausgewählten Tabelle auszuwählen. Folgende Abbildung zeigt, was gemeint ist:

Autovervollständigung

In Abhängigkeit der Cursorposition werden entweder die vorhandenen Schemata, die Tabellen im gewählten Schema oder die Spaltennamen der entsprechenden Tabelle angezeigt. Ein kleines aber nützliches Feature!

Auch nicht zwingend notwendig, aber dennoch nützlich, ist die SQL-Formatierung.

Sie können SQL-Statements automatisch formatieren lassen. Schauen Sie sich die folgende Abbildung an, dann werden Sie schnell verstehen, was gemeint ist:



Automatische SQL-Formatierung

Die Regeln für die Formatierung bestimmen Sie teilweise selbst in den Optionen. Öffnen Sie hierfür das Menü „Extras ➜ Optionen ➜ SQL-Formatierung“.

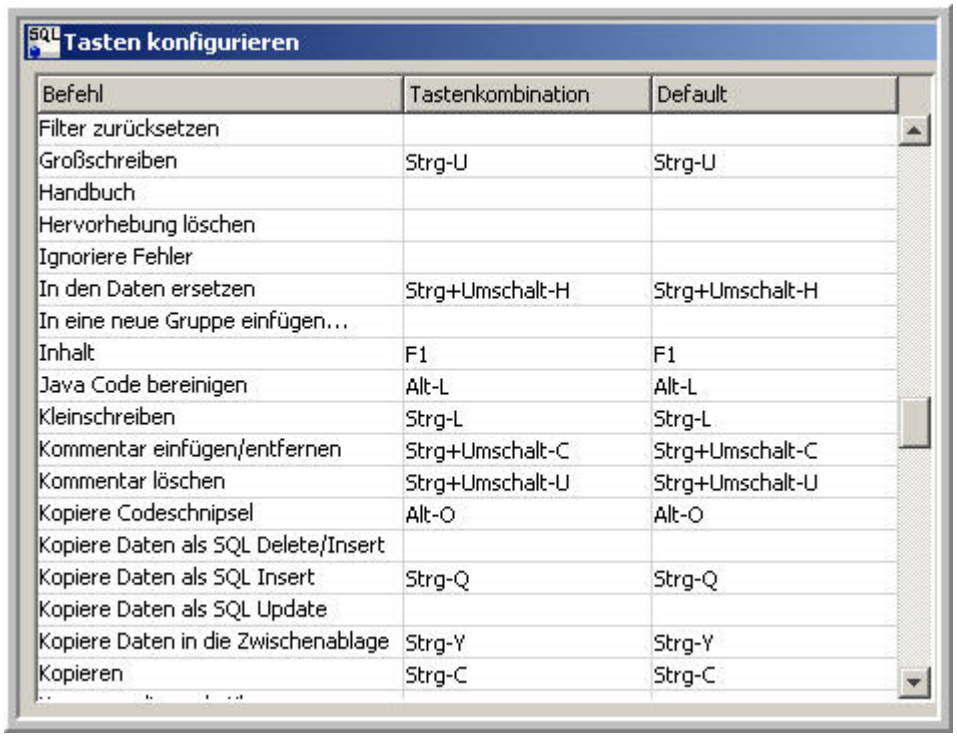
Optionen – SQL-Formatierung

Auch die schnelle Kommentierung ausgewählter SQL-Statements ist komfortabel. Stellen Sie sich vor, dass innerhalb des Skriptes einige Statements auskommentiert werden sollen. In der SQL-Workbench müssen Sie die Anweisungen lediglich markieren, wählen anschließend im Menü „Bearbeiten ➜ Kommentare einfügen/entfernen“ oder nutzen den Shortcut „STRG+Umschalt+C“ und sofort wird das entsprechende Statement auskommentiert:

```
--SELECT customer_number FROM eps0042cr.customer;
```

Kommentare einfügen und entfernen

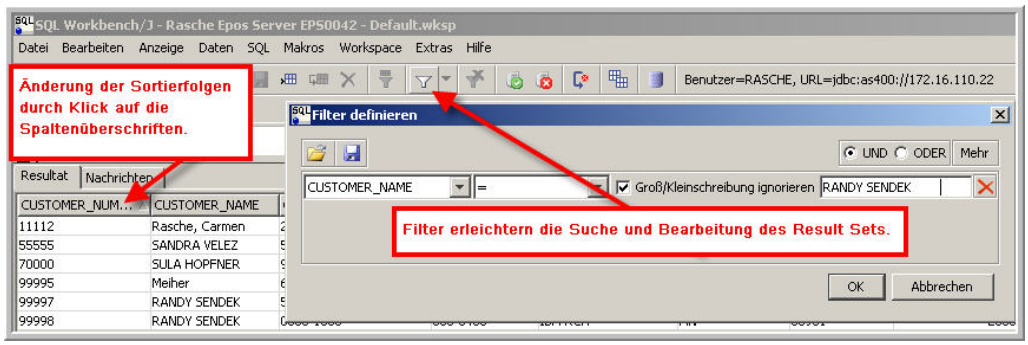
Die Shortcuts können Sie im Übrigen individuell bestimmen. Öffnen Sie hierzu das Menü „Extras ➜ Shortcuts“.



Shortcuts definieren

Alle Tastenkombinationen sind änderbar und natürlich können Sie auch eigene, zusätzliche Shortcuts festlegen.

Einige Icons der Toolbar beziehen sich auf das Result Set. So können Sie für die selektierte Datenmenge Filterkriterien aufbauen, was die Suche im Result Set oftmals deutlich vereinfacht. Auch ist es problemlos möglich, durch Klicken auf die Spaltenüberschriften die Sortierfolgen beliebig zu ändern. Die Daten werden hierfür nicht erneut vom Server abgerufen, sondern tatsächlich auf die Result Sets des Clients angewendet.



Filterkriterien im Result Set definieren

Ich hoffe, Sie konnten einen ersten Eindruck über die vielfältigen Funktionen der Skriptumgebung gewinnen. Ich habe sicherlich nicht alle Funktionen erwähnt, aber vieles ist selbsterklärend, wenn Sie sich einfach ein wenig Zeit nehmen und die Menüpunkte erforschen.

7.4.5.5 Die SQL Workbench/J – Datenbank-Explorer

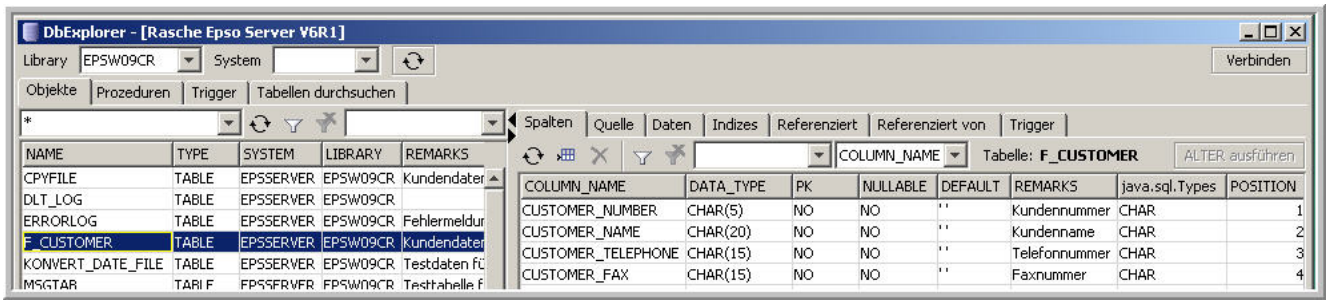
Die SQL Workbench/J ist nicht nur bei der Arbeit mit SQL Statements ein komfortables Tool, auch der Datenbank-Explorer muss sich nicht verstecken. Der Datenbank-Explorer zeigt die verfügbaren Datenbankobjekte innerhalb eines Schemas an. Tabellen, Views aber auch Trigger und Stored Procedures werden gelistet. Für die Arbeit mit den Datenbankobjekten stehen viele nützliche Funktionen zur Verfügung. Wenn Sie die SQL Workbench wie bisher öffnen, wird der Datenbank-Explorer automatisch in einem eigenen Register gestartet:

Der Datenbank-Explorer

Es gibt aber auch die Möglichkeit den Explorer separat zu starten, indem Sie dem Workbenchaufruf folgenden Parameter hinzufügen -dbexplorer. Der Aufruf sähe dann folgendermaßen aus:

```
„C:\Program Files\SQL Workbench 107\
SQLWorkbench.exe“ -dbexplorer
```

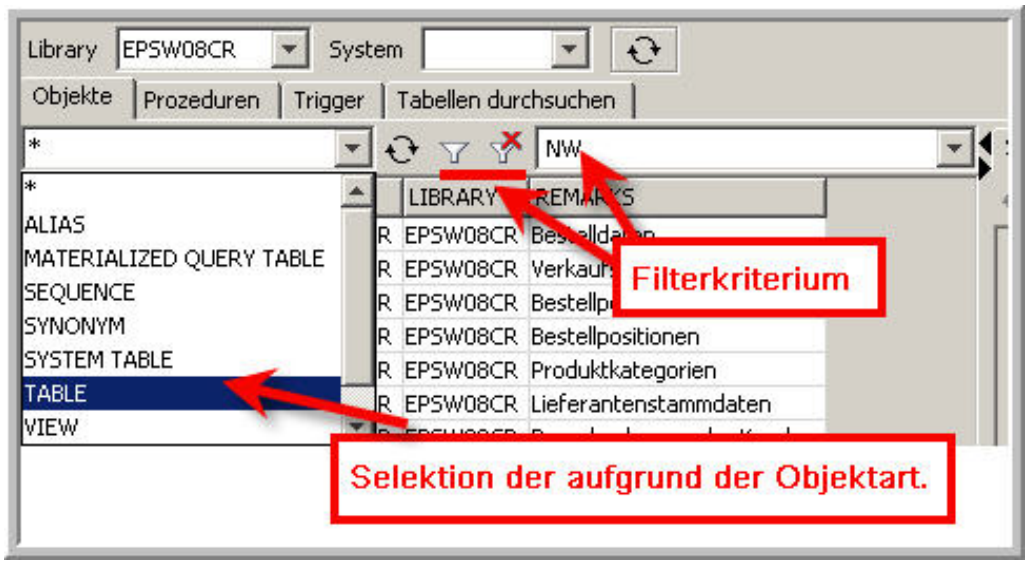
Durch den Direktaufruf steht die SQL-Skriptumgebung nicht zur Verfügung:



Datenbank-Explorer Direktaufruf

Deutlich ist zu erkennen, dass das Register „Statement 1“ in der zweiten Abbildung fehlt. Eigene SQL-Abfragen können somit nicht formuliert werden; der Anwender muss sich mit den Möglichkeiten des DB-Explorers begnügen, aber auch die sind vielfältig.

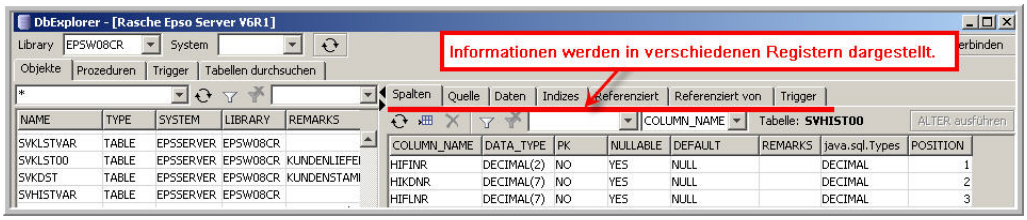
Im oberen Bereich des Bildschirms können Sie das Schema (=Bibliothek) und den Server auswählen, dessen Daten Sie bearbeiten wollen. Haben Sie sich für ein Schema entschieden, werden alle im Schema vorhandenen Objekte angezeigt. Nicht selten haben wir aber in einem Schema hunderte von Datenbankobjekten gespeichert: Tabellen (=Physische Dateien), logische Sichten (Logische Dateien), referentielle Integritätsbedingungen, Datenbanktrigger u. v. m. Hier hilft es Filterkriterien zu setzen. Sie können nach den Objektarten selektieren. Wollen Sie beispielsweise nur mit den Tabellen arbeiten, lassen Sie sich auch nur diese anzeigen.



Filterkriterien erleichtern die Arbeit

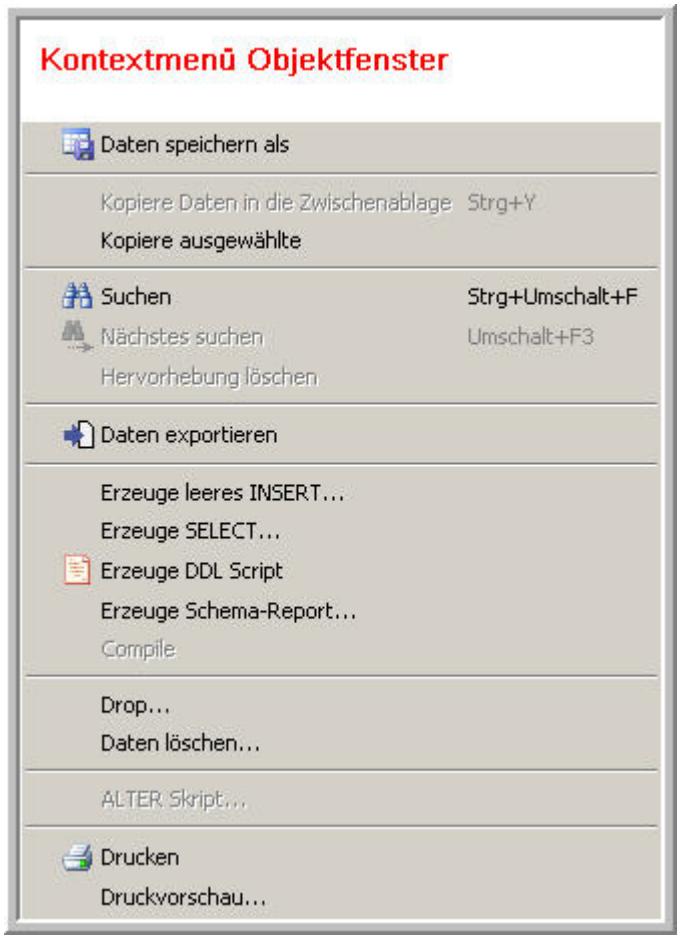
Doch dieses Kriterium allein ist häufig nicht ausreichend und daher können Sie zusätzlich noch das Filterfeld nutzen. Um die Objekte anhand Ihres Objektnamens zu filtern, geben Sie das Kriterium an, drücken ENTER oder klicken auf das Filter-Icon. Im Beispiel oben werden alle Tabellen angezeigt, deren Objektname die Zeichenfolge „NW“ beinhaltet. Soll ausgedrückt werden, dass der Objektname mit der Zeichenfolge „NW“ beginnen soll, müssen Sie das Zeichen ^ der Suchzeichenfolge voranstellen.

Schauen wir uns jetzt die verfügbaren Informationen und Bearbeitungsmöglichkeiten im DB-Explorer etwas genauer an:



Informationen werden in unterschiedlichen Registern abgebildet

Der Datenbank-Explorer gliedert sich in zwei große Hauptbereiche. Im linken Fensterbereich sehen Sie alle Objekte des entsprechenden Schemas (=Objektfenster), im rechten Fensterbereich sehen Sie stets Detailinformationen des markierten Objektes (=Detailfenster); aufgeteilt auf unterschiedliche Register. Beide Fensterbereiche stellen über entsprechende Kontextmenüs diverse Bearbeitungsmöglichkeiten zur Verfügung. Schauen wir uns zunächst einige Kontextfunktionen des Objektfensters an:



Kontextmenüs des Objektfensters

Erzeuge leeres INSERT

Die Funktion erzeugt ein leeres Insert-Statement für die selektierte Tabelle. Die Funktion ist nützlich für Programmierer, da sich auf diese Weise sehr schnell eine Insert-Anweisung generieren lässt. Aber auch SQL-Neulingen bleibt die Syntax erspart und Tipp-Fehler werden auf diese Weise ebenfalls verhindert.

Wirkungsweise „Erzeuge leeres INSERT“

Steht keine SQL-Skriptumgebung zur Verfügung, ist diese Bearbeitungsmöglichkeit dagegen nicht relevant, denn das erzeugte Insert-Statement kann in dieser Umgebung nicht ausgeführt werden.

Ähnlich verhält es sich, wenn Sie die Option „Erzeuge SELECT“ oder „Erzeuge DDL-Statement“ nutzen. Im ersten Fall wird ein SELECT-Statement erzeugt, im zweiten Fall wird das entsprechende CREATE TABLE-Statement generiert.

Erzeuge Schema-Report

So wird ein XML-Bericht der selektierten Tabellen erstellt. Das generierte XML-File kann in einem beliebigen Schema Ihrer Wahl gespeichert werden.

DROP

Die Funktion DROP ist selbsterklärend. Die markierte Tabelle wird gelöscht, vorausgesetzt Sie besitzen die notwendigen Objektberechtigungen.

Daten löschen

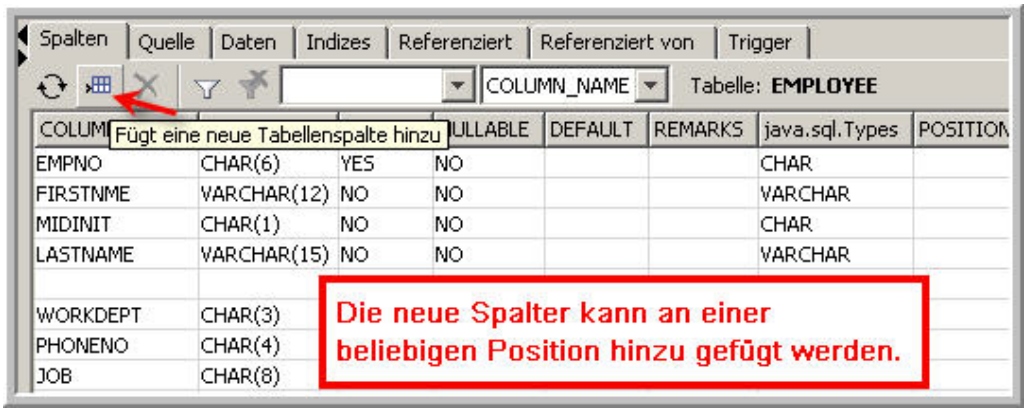
Auch dieser Bearbeitungspunkt ist selbsterklärend; sie entspricht dem CL-Befehl CLRPFM (clear physical file member) und löscht alle Datensätze der markierten Tabelle.

Aufmerksame Leser haben sicherlich bemerkt, dass ich es bislang versäumt habe, die Funktionen **„Daten speichern als“ und „Daten exportieren“** zu erläutern. Hinter beiden Funktionen verbergen sich in die Workbench/J integrierte Funktionen (WbExport, WbImport), die mächtige Datentransfers und Datenabgleiche ermöglichen. Die Funktionen sind so vielfältig, dass ich mich entschieden habe diese in einem separaten Kapitel zu behandeln.

Bislang haben wir uns nur das Objektfenster angeschaut. Nun betrachten wir die Detailinformationen. Wird eine Tabelle selektiert, erhalten Sie im rechten Fenster im Register „Spalten“ Informationen über die Spaltendefinitionen.

Die Detailinformationen einer Tabelle

Sie können die Spaltendefinitionen ändern, löschen aber auch neue Spalten hinzufügen. Dazu klicken Sie auf die Spalte, deren Attribute Sie ändern wollen, oder fügen eine Spalte mit Hilfe des entsprechenden Icons ein.

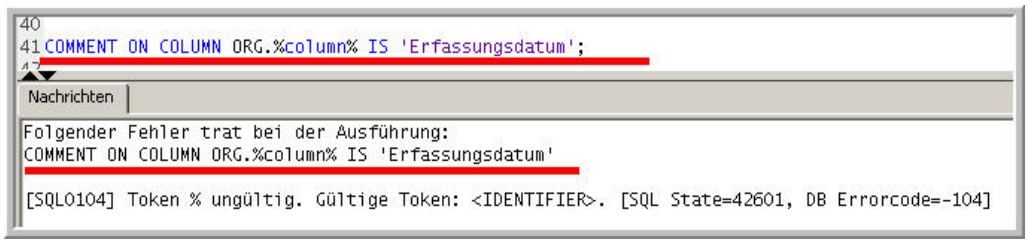


Hinzufügen neuer Spalten

Sind die Änderungen erfolgt, klicken Sie auf den Button „Alter ausführen“; das zugehörige ALTER TABLE-Statement wird angezeigt und kann, falls gewünscht, jetzt ausgeführt werden.

Leider funktionieren diese Möglichkeiten derzeit nicht in vollem Umfang mit einem System-i-Server. Sie können zwar problemlos die Datentypen manipulieren und Default-Werte verändern, aber keine Spalten hinzufügen oder löschen. Herr Keller bietet in seinem Handbuch an, die notwendigen SQL-Statements zu implementieren, wenn man ihm die entsprechenden Informationen zur Verfügung stellt. Dies habe ich bereits getan, bleibt abzuwarten, wann die Features verfügbar sind.

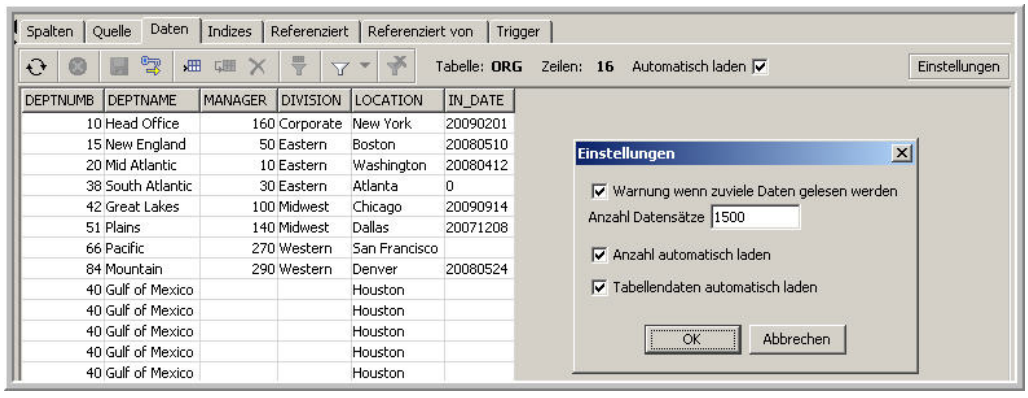
Die SQL Workbench/J generiert wie System i Navigator den Source Code einer Tabelle anhand der zur Verfügung gestellten Metadaten einer Tabelle. Manchmal liefert der Treiber allerdings nicht die korrekten Informationen. Hier müssen Sie also Vorsicht walten lassen, denn einige Statements werden nicht ausführbar sein, wie folgendes Beispiel zeigt:



Detailinformationen – Das Register „Quelle“

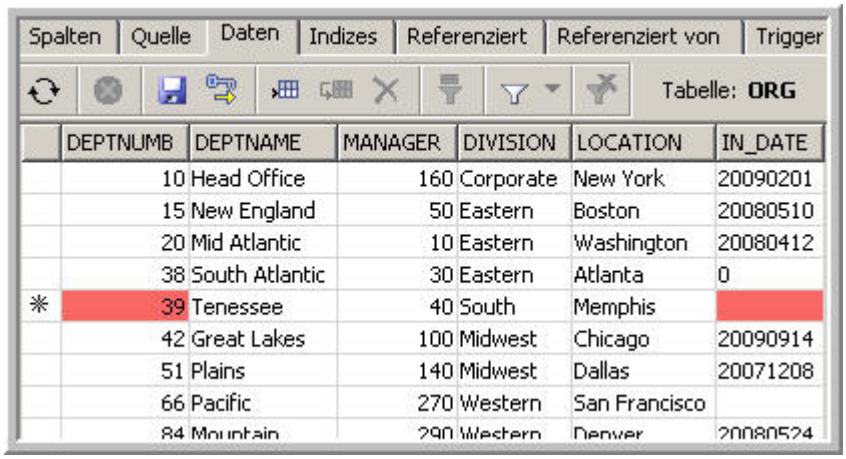
Es existiert zwar ein beschreibender Text für die Spalte IN_DATE, dieser wurde aber mit der Anweisung LABEL ON erstellt. Die DB2 UDB kennt natürlich auch die Anweisung COMMENT ON, allerdings bewirkt diese eine andere Art von Kommentar und ist syntaktisch falsch, da die DB2 nicht mit dem Prozentzeichen in dieser Form arbeitet. Sie müssen also die Quellen sorgfältig prüfen und ggf. ein klein wenig anpassen bevor sie lauffähig sind.

Am besten gefällt mir das Register Daten. Im Register „Daten“ sollten Sie als erstes spezifische Einstellungen vornehmen, um beispielsweise die Anzahl der zu lesenden Datensätze zu begrenzen. Des Weiteren werden die Anzahl Zeilen innerhalb der Tabelle ermittelt und die Daten der Tabelle automatisch geladen. Dies sollten Sie ggf. unterbinden, um nicht unnötig das Netz zu belasten. Fordern Sie die Daten nur an, wenn sie auch wirklich benötigt werden.



Das Register Daten – spezifische Einstellungen

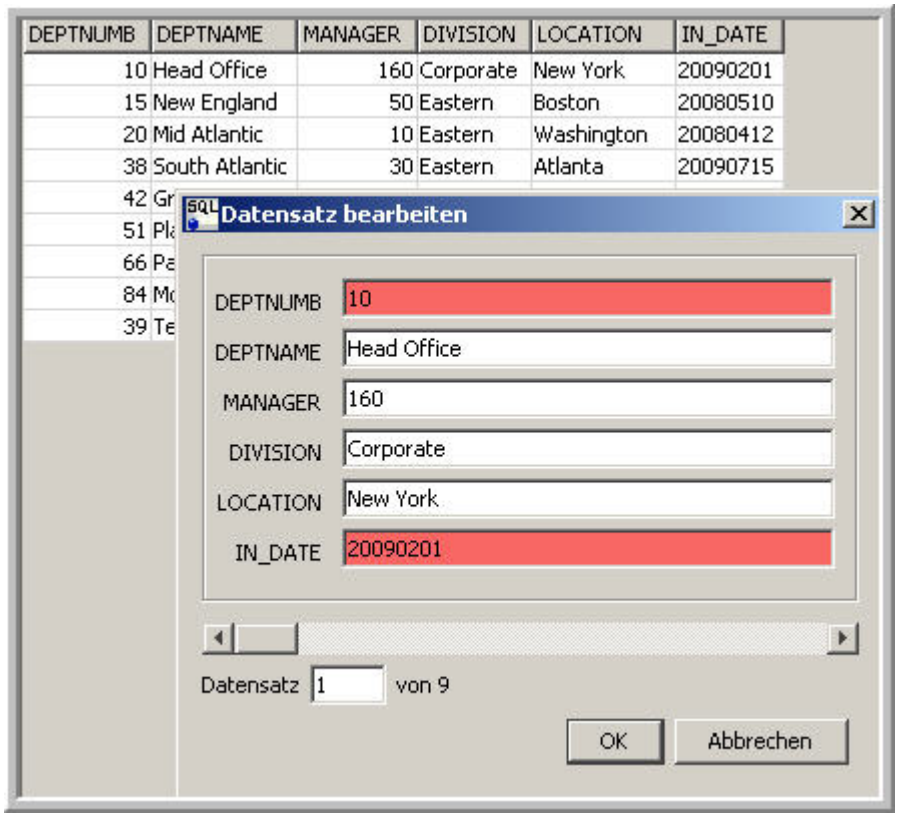
Die angezeigten Daten können Sie anschließend beliebig manipulieren, indem Sie Dateninhalte verändern, neue Datensätze einfügen oder bestehende Daten verändern. Während Ihrer Arbeit werden die geänderten Daten markiert.



	DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION	IN_DATE
	10	Head Office	160	Corporate	New York	20090201
	15	New England	50	Eastern	Boston	20080510
	20	Mid Atlantic	10	Eastern	Washington	20080412
	38	South Atlantic	30	Eastern	Atlanta	0
*	39	Tenessee	40	South	Memphis	
	42	Great Lakes	100	Midwest	Chicago	20090914
	51	Plains	140	Midwest	Dallas	20071208
	66	Pacific	270	Western	San Francisco	
	84	Mountain	290	Western	Denver	20080524

Register Daten – Datensatz einfügen

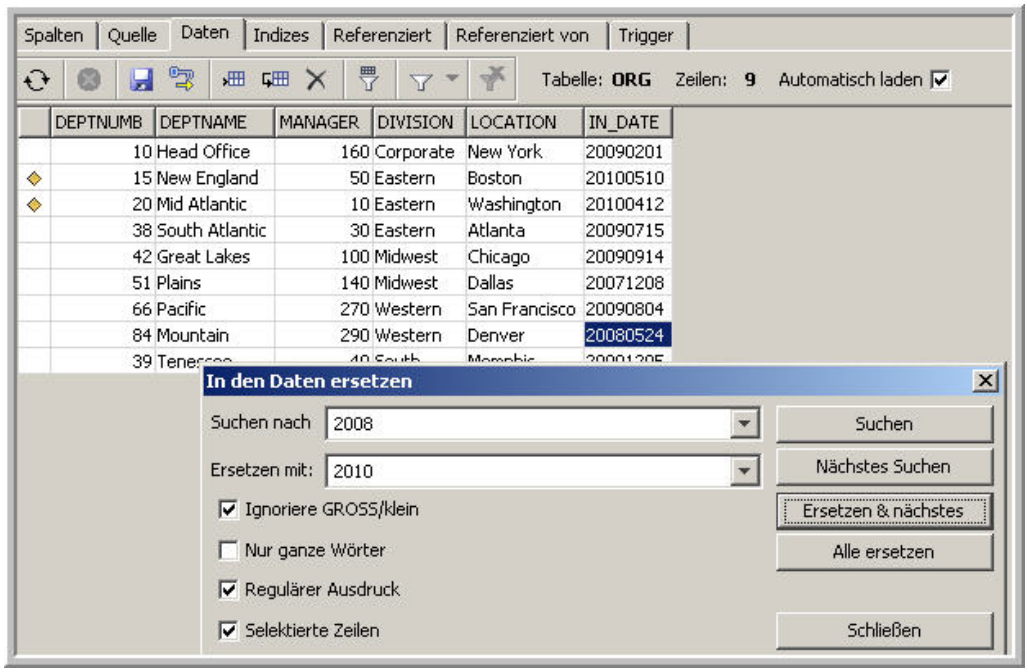
Sie müssen Ihre Änderungen auch nicht direkt im Register ausführen, denn über die Kontexttaste steht Ihnen der Menüpunkt „Datensatz Dialog“ zur Verfügung, der eine komfortable Oberfläche für die Bearbeitung der Daten bereitstellt:



Der Dialog – Datensatz bearbeiten

7.4.5
Seite 34

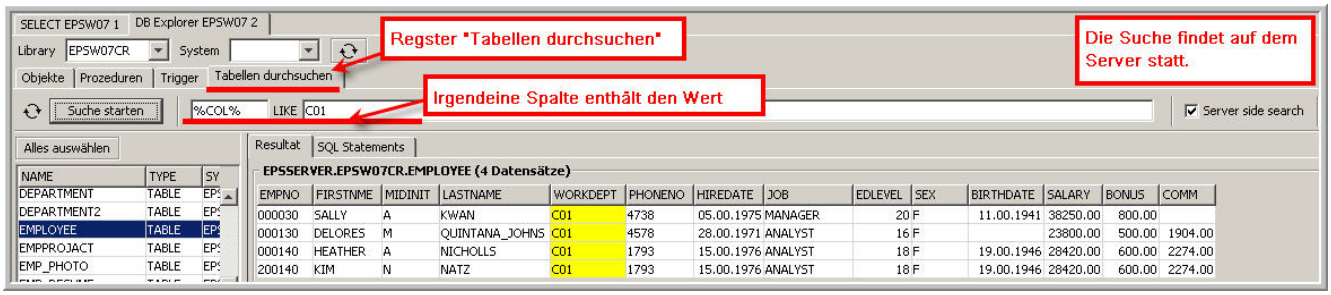
Ich kann nicht alle Kontextfunktionen dieses Registers erläutern, aber einige Funktionen sind durchaus erwähnenswert. So finden Sie im Kontextmenü des Datenregisters die Funktion „In den Daten ersetzen".



Die Kontextfunktion „Suchen und Ersetzen"

Hier können Sie ähnlich einem Word-Dokument innerhalb der Daten bestimmte Daten suchen und diese global oder step-by-step ersetzen. Eine Funktion, die ich mir auch im System i Navigator wünschen würde. Wenn Sie den Dialog öffnen, müssen Sie ggf. die Schlüsselspalte der Tabelle bestimmen, wenn kein Primärschlüssel für die Tabelle definiert wurde, aber anschließend funktioniert der Dialog fehlerfrei.

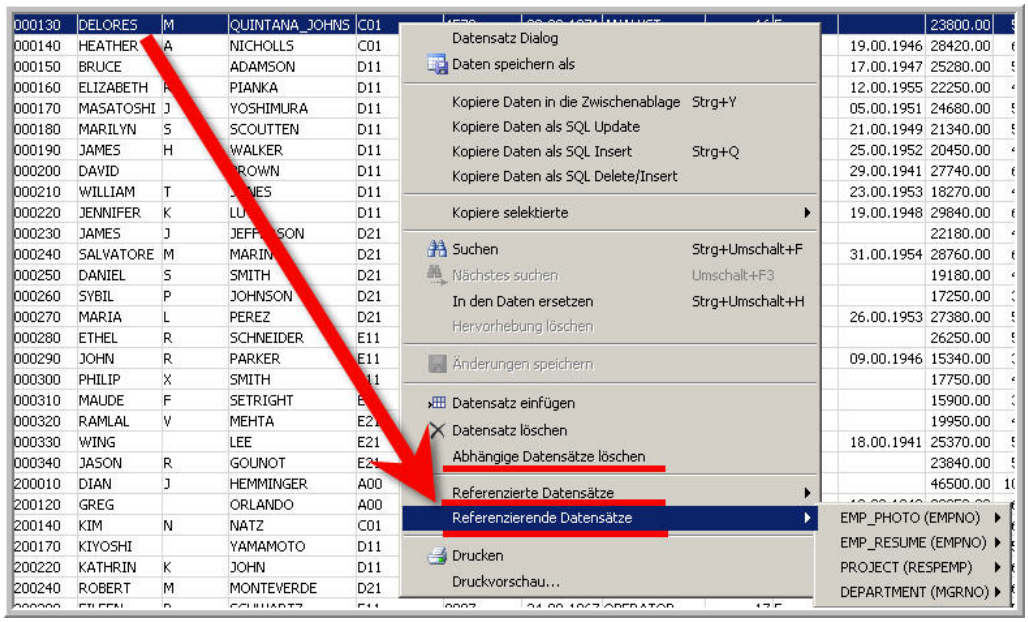
Es gibt aber noch eine weitere Möglichkeit die Datenbasis einer Tabelle zu durchsuchen. Im Objektfenster finden Sie das Register „Tabellen durchsuchen". Wenn Sie dieses Register öffnen, sehen Sie den folgenden Dialog:



Tabellen durchsuchen

Sie können eine einzelne Tabelle oder mehrere Tabellen markieren und einen Wert in einer bestimmten Spalte oder in einer beliebigen Spalte der Tabelle suchen. In meinem Beispiel suche ich den Wert „C01“ in irgendeiner (%COL%) Spalte der markierten Tabelle „Employee“. Soll nur in der Spalte „WORKDEPT“ gesucht werden, müsste ich an Stelle des Wertes %COL% den Spaltennamen eintragen. Auf diese Weise lassen sich wirklich einfach identische Werte in unterschiedlichen Tabellen suchen. Achten Sie aber darauf, dass Sie die Suche Serverseitig ausführen, andernfalls würde ihr PC entsprechend stark belastet werden.

Kommen wir aber wieder zurück zum Register „Daten“. Hier gibt es einen weiteren Funktionsblock, den ich gelungen finde:



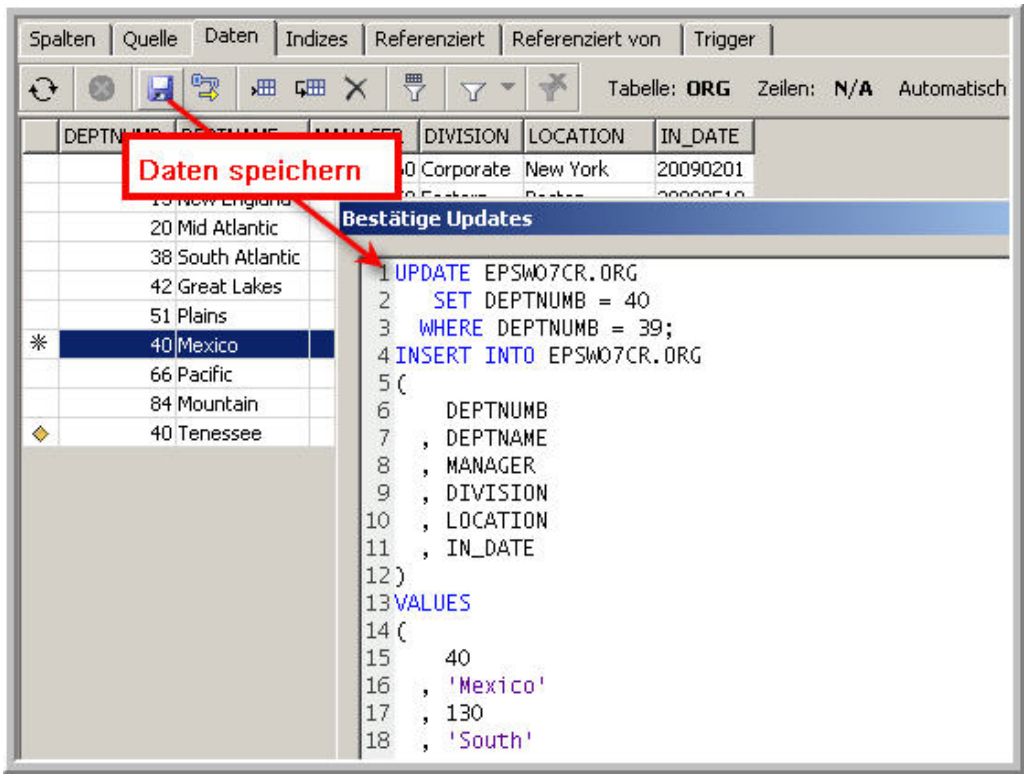
Abhängige Tabellensätze anzeigen

Sind referentielle Integritätsbedingungen definiert, können für einen spezifischen Datensatz die abhängigen Child-Sätze aber auch zugehörige Parent-Sätze angezeigt, und falls notwendig, auch gelöscht werden.

Zum Funktionsblock „Kopiere Daten“ findet man ebenfalls kein Pendant im System i Navigator. Sie sollten diese Funktionen einmal in Ruhe ausprobieren, schnell werden Sie gute Verwendungsmöglichkeiten dafür finden.

Egal wie und welche Änderungen Sie an den Daten ihrer markierten Tabelle vornehmen, die Änderungen werden zunächst nur als SQL-Statement festgehalten. Erst wenn Sie auf das Icon „Speichern“ klicken, wird das generierte SQL-Skript angezeigt und jetzt können Sie die Änderungen durchführen oder das Skript nochmals manuell anpassen, um dann in einem separaten Schritt die Veränderungen umzusetzen.

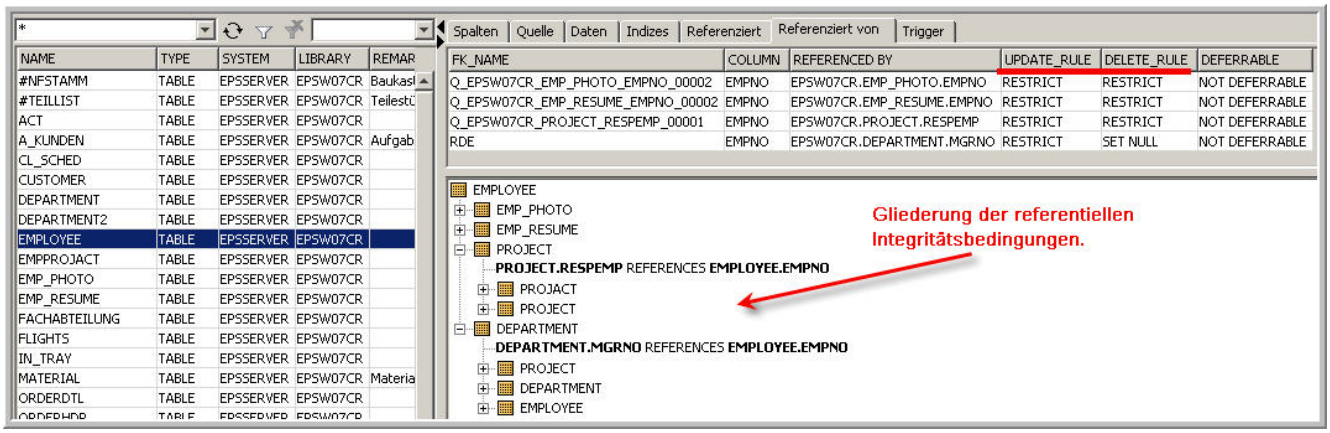
7.4.5
Seite 36



Das Register Daten – geänderte Daten speichern

Die Bearbeitung der Datenbasis ist meiner Meinung nach wirklich komfortabel und gut gelungen, wer viel und oft mit seinen Daten arbeiten muss, sollte sich mit den entsprechenden Funktionen vertraut machen. Es lohnt sich!

Das Register „Index“ bietet keine Überraschungen. Gut gelungen ist aber wieder die Anzeige abhängiger Tabellen durch entsprechende referentielle Integritätsbedingungen:



Gliederung referentieller Integritätsbedingungen

Es werden nämlich nicht nur die Bedingungen und deren UPDATE- und DELETE-Regeln angezeigt. Die Beziehungen werden zusätzlich in Form eines Gliederungsschemas zwischen den Tabellen verdeutlicht.

7.4.6 Eclipse Datenbank-Tools

Die Tatsache, dass System-i-Anwendungsentwickler mit den Nachfolgeprodukten von WDSC – etwa mit RDi (Rational Developer for IBM i) – keine Möglichkeit mehr haben, Datenbanken bzw. deren Inhalte direkt zu bearbeiten, darf als unglücklich bezeichnet werden. Wunderlich mutet diese Tatsache auch deshalb an, weil IBM nicht ohne Grund die Verwendung SQL-basierter Datenbanken der Nutzung klassischer, auf physischen Dateien basierender Datenbanken vorzieht.

Während in WDSC die Datenperspektive zum Einsatz von SQL genutzt werden konnte, fehlt diese in RDi-Paketen.

Nun hat IBM die Entwicklerwerkzeuge auf der Basis der Eclipse-Funktionen und Plug-in-Technologien erstellt. Das bietet uns die Möglichkeit, Erweiterungen von Basisprodukten – wie RDi – durch die Installation zusätzlicher Plug-ins zu nutzen.

Ein solches Plug-in wird direkt von Eclipse geliefert und bietet unterschiedliche Funktionen zum Arbeiten mit Datenbanken und SQL: Dabei ist der Einsatz des Eclipse Data Tools nicht allein auf die Verwendung mit der DB2/UDB des System i beschränkt, sondern auch für die Nutzung mit anderen Datenbanken geeignet.

Die Eclipse Data Tools Platform (kurz DTP) kann kostenlos von den Eclipse-Webseiten heruntergeladen werden. Der folgende Link verweist auf die entsprechenden Eclipse-Webseiten:

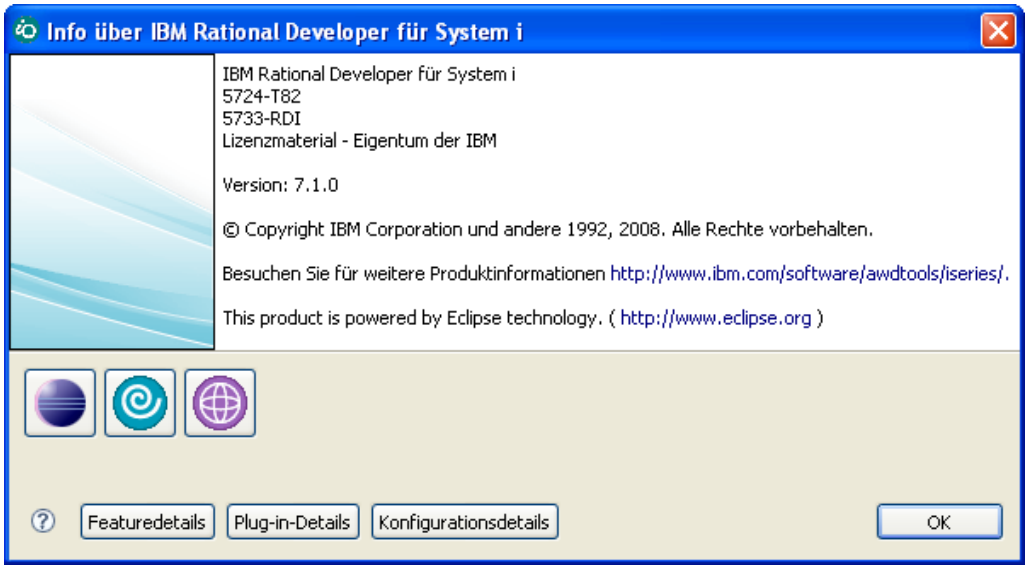
http://www.eclipse.org/datatools/downloads.php

Achten Sie UNBEDINGT darauf, dass Sie die korrekte Version von DTP herunterladen und installieren. Insbesondere die Kompatibilität mit der installierten Eclipse-Version ist wichtig. Die aktuell genutzte Version können Sie in RDi wie folgt ermitteln:

Selektieren Sie die Menüoption „Hilfe / Info über IBM Rational Developer for System i“.

7.4.6
Seite 2

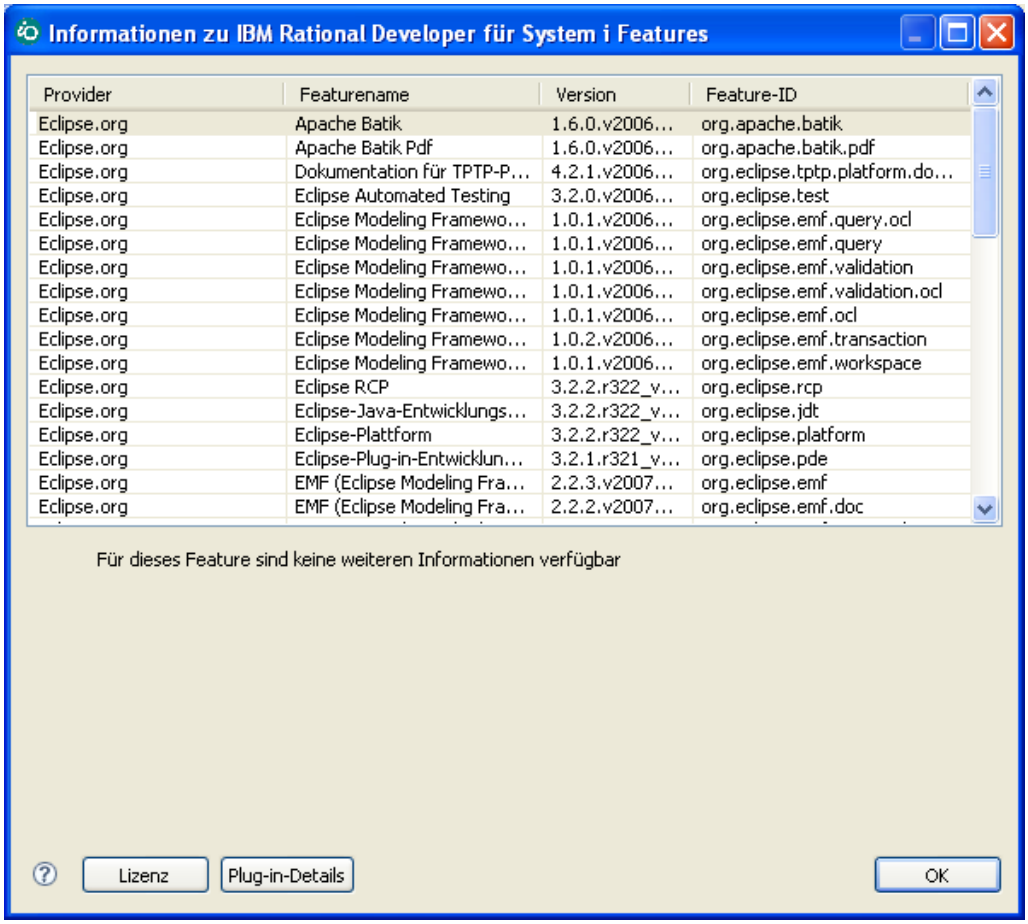
Daraufhin erscheint eine Anzeige wie sie in der folgenden Abbildung zu sehen ist:



Installierte Version

Wählen Sie die Option „Featuredetails“.

Damit erhalten Sie eine Übersicht der installierten Features. Unter anderem finden Sie dort auch die installierte Eclipse-Version. In unserem Fall ist dies die Version 3.2.2.



Eclipse-Version

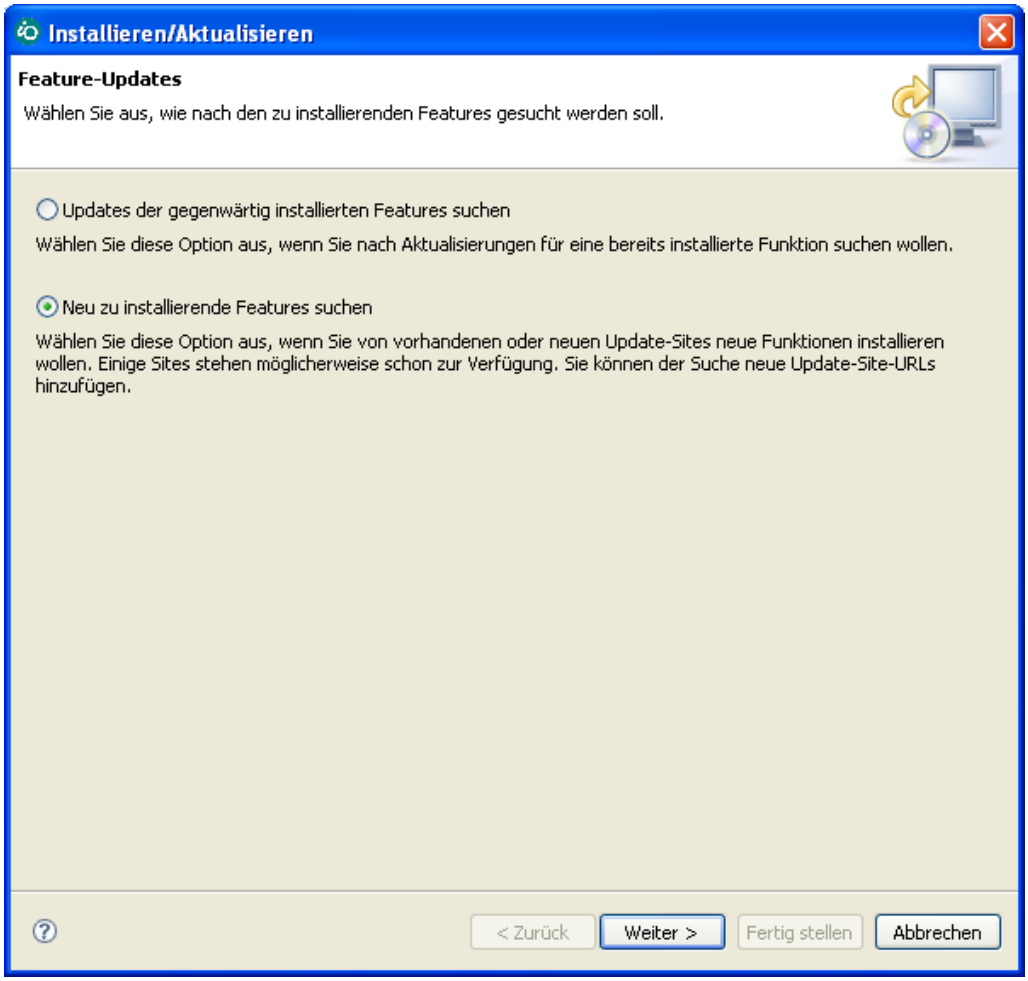
Das bedeutet, dass wir bei dieser RDi-Installation eine DTP-Version nutzen können, die mit der Eclipse-Plattform 3.2.2 kompatibel ist.

Nach dem Herunterladen des DTP muss es entpackt werden. Das Ziel für das Entpacken kann beliebig bestimmt werden.

Nun bereiten wir das Installieren vor. Dazu wählen wir zunächst innerhalb von RDi die Menüoption „Hilfe / Software Updates / Suchen und installieren“.

7.4.6
Seite 4

Daraufhin wird ein neues Fenster geöffnet, in dem wir die Option „Neu zu installierende Features suchen“ anklicken.



Neu zu installierende Features suchen

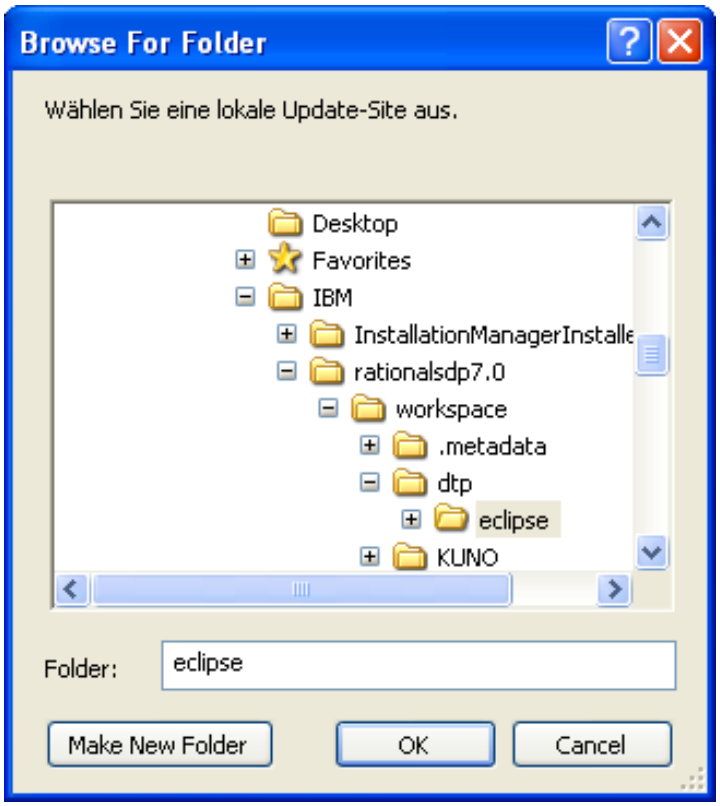
Bestätigen Sie die Auswahl mit einem Klick auf die Schaltfläche „Weiter“.

Nun werden die verfügbaren Update-Bereiche angezeigt. Dort fehlt der DTP-Bereich noch. Durch einen Klick auf die Schaltfläche „Neue lokale Site“ kann dieser für die Installation hinzugefügt werden.

Klicken Sie auf die Schaltfläche „Neue lokale Site“.

Wählen Sie den Speicherort der entpackten Installationsdaten für DTP aus.

Wählen Sie als Verzeichnis den Folder „Eclipse“.

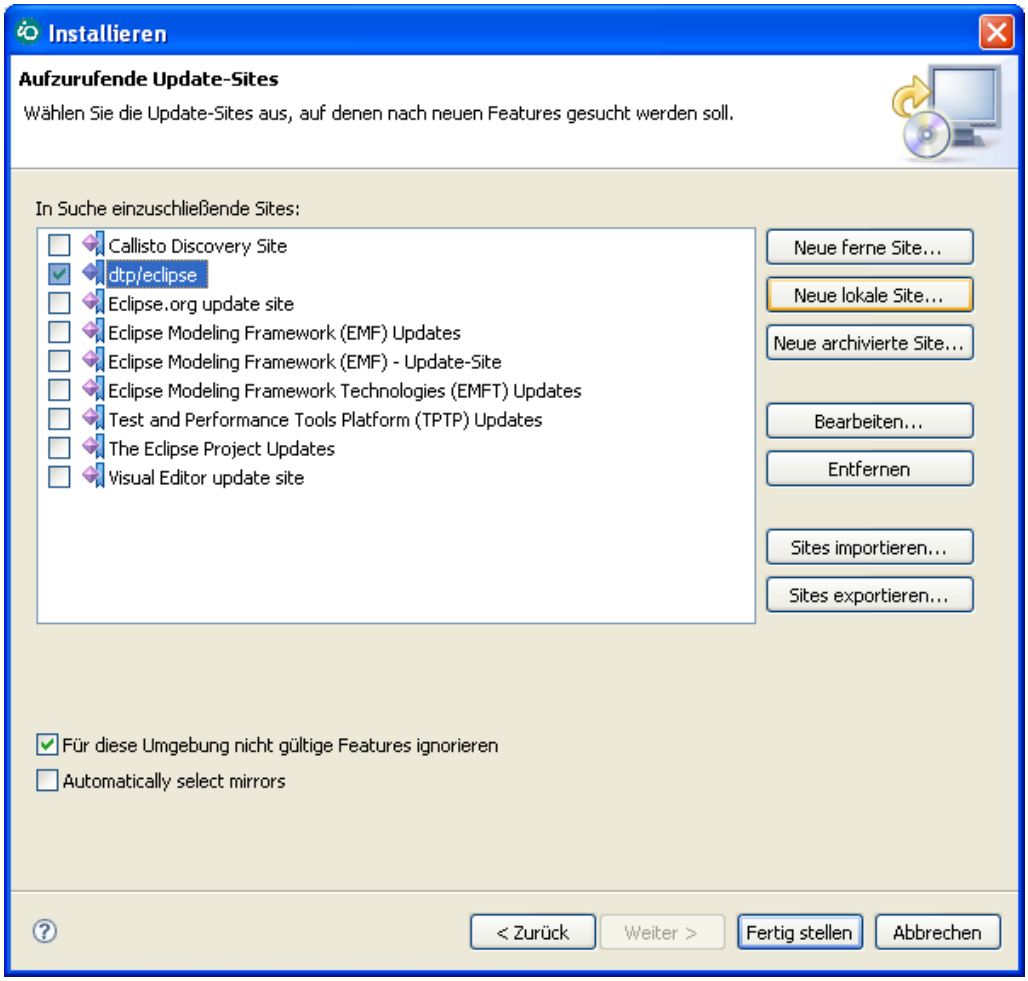


Auswahl Eclipse-Installation

Bestätigen Sie die Auswahl mit einem Klick auf „OK“.

7.4.6
Seite 6

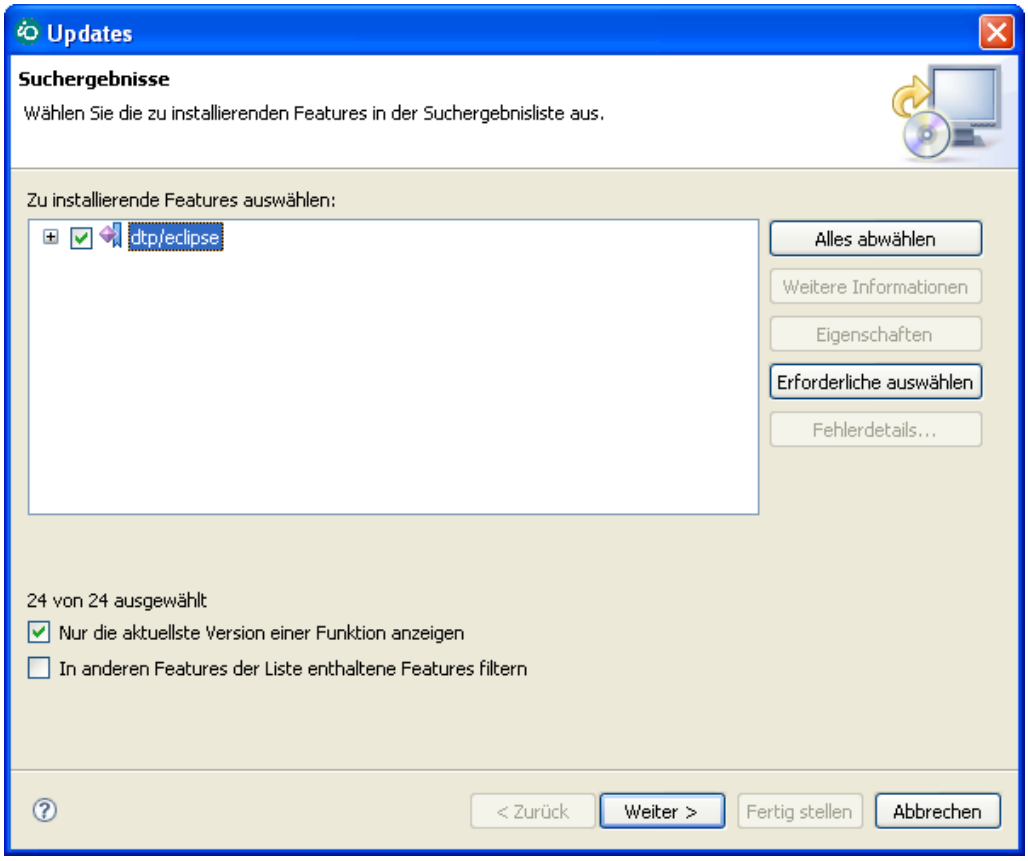
Damit erscheint jetzt der DTP-Bereich auch in der Auflistung der zu installierenden Sites:



Neue Update Site

Der Installationsprozess wird jetzt mit einem Klick auf die Schaltfläche „Fertig stellen“ gestartet.

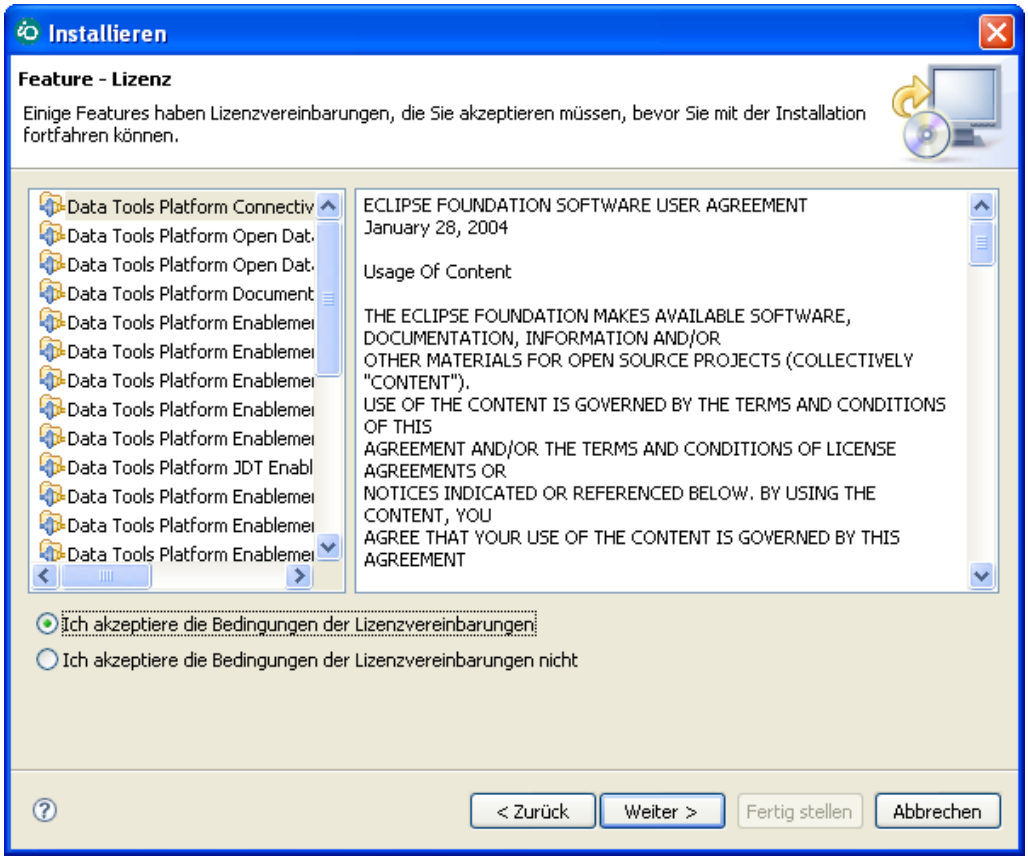
Nach einem kurzen Vorbereitungslauf wird das neu zu installierende Feature aufgelistet. Zur Installation muss es noch markiert werden. Dazu genügt ein Klick in das Kästchen links neben der Bezeichnung des Features.



Zu installierende Features

7.4.6
Seite 8

Mit einem Klick auf die Schaltfläche „Weiter“ werden zunächst die Lizenzbestimmungen für die Eclipse Data Tools angezeigt. Diese müssen für die Fortsetzung der Installation akzeptiert werden.

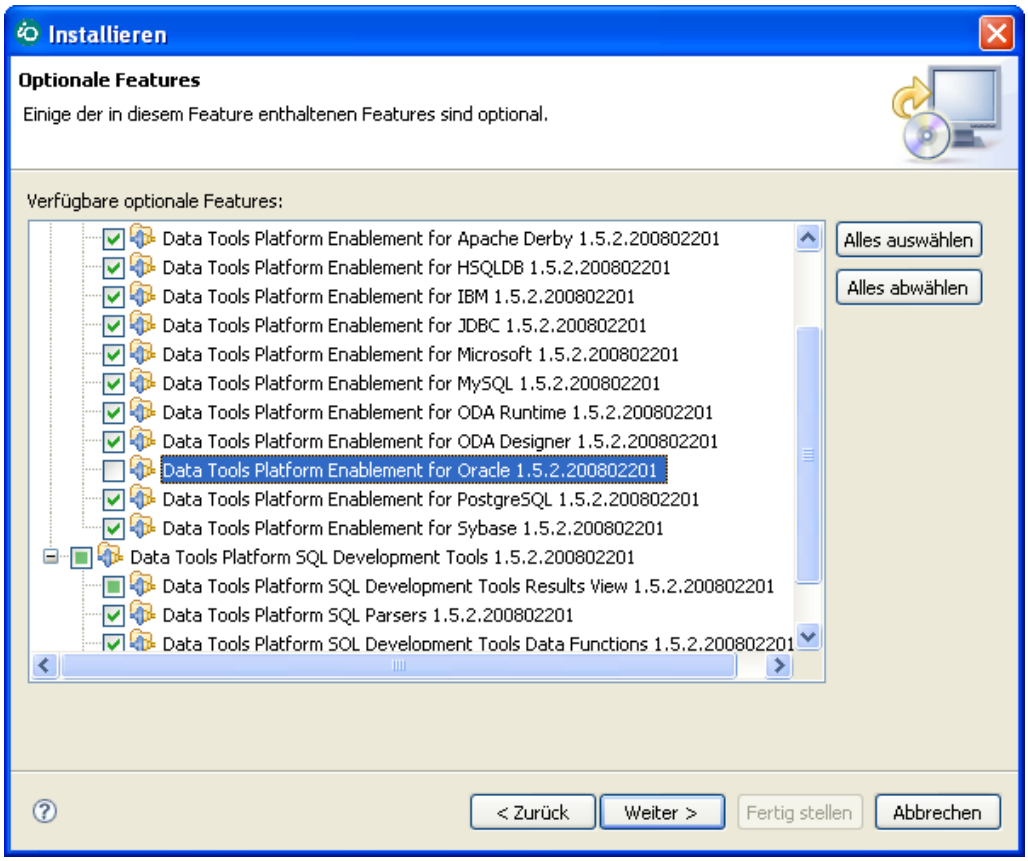


Lizenz

Akzeptieren Sie die Lizenzbestimmungen

Bestätigen Sie mit einem Klick auf „Weiter“.

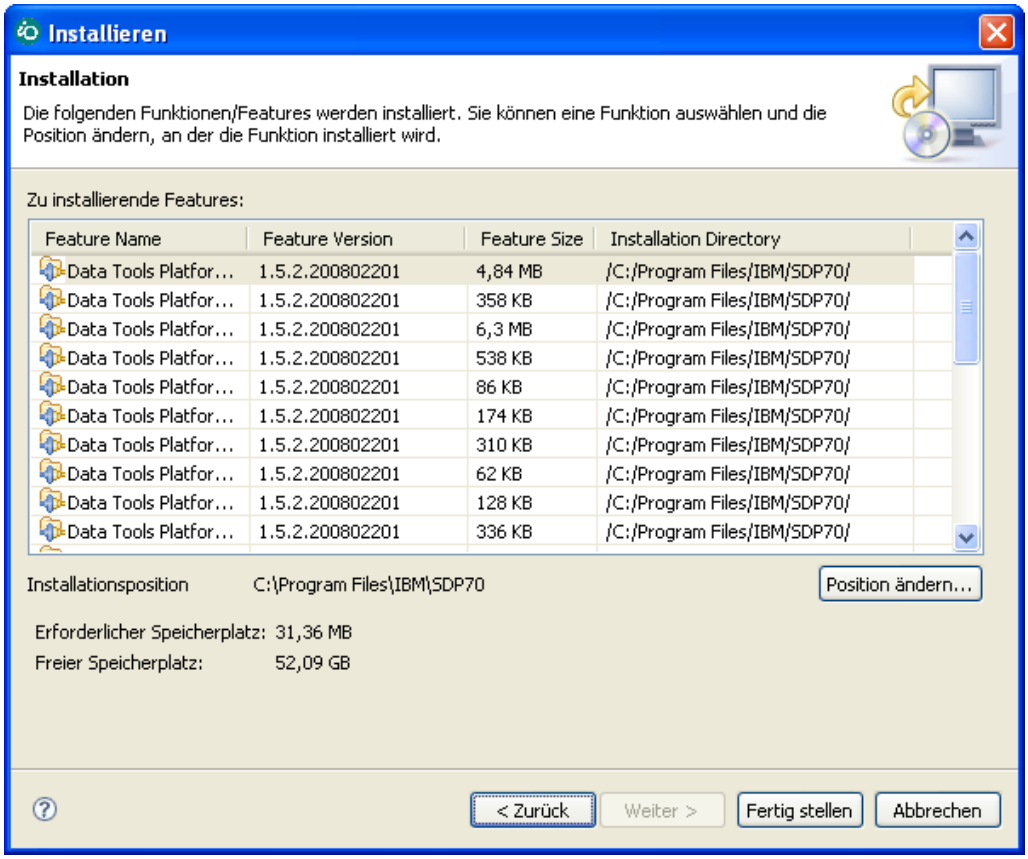
Die optional zu installierenden Komponenten werden aufgelistet. Unter anderem finden wir hier verschiedene Datenbankoptionen. Sollten Sie beispielsweise keine Oracle-Datenbanken mit RDi nutzen wollen, dann kann diese Option deaktiviert werden. So habe ich es exemplarisch auch in der folgenden Abbildung gemacht.



Data-Tools-Auswahl

Mit einem Klick auf „Weiter“ werden nun die zu installierenden Details angezeigt. Der eigentliche Installationsprozess wird mit einem abschließenden Klick auf die Schaltfläche „Fertig stellen“ gestartet.

7.4.6
Seite 10



Installationsposition

Nach der Installation muss RDi erneut gestartet werden. Dies kann wahlweise sofort oder auch später erfolgen. Um die neuen Funktionen nutzen zu können, muss der Neustart aber erfolgt sein.

Damit ist die Grundinstallation von DTP abgeschlossen. Jetzt müssen wir die Treiber für die zu verwendenden Datenbanken definieren. Diese werden über die Benutzervorgaben gesteuert. Die Verwaltung der Benutzervorgaben erfolgt über den Menüeintrag „Fenster / Benutzervorgaben“.

Wählen Sie die Menüoption „Fenster / Benutzervorgaben“.

In den Benutzervorgaben sind verschiedene Einstellungen möglich. Die Treiberangaben für DTP finden wir unter dem Abschnitt „Connectivity / Driver Definitions“.

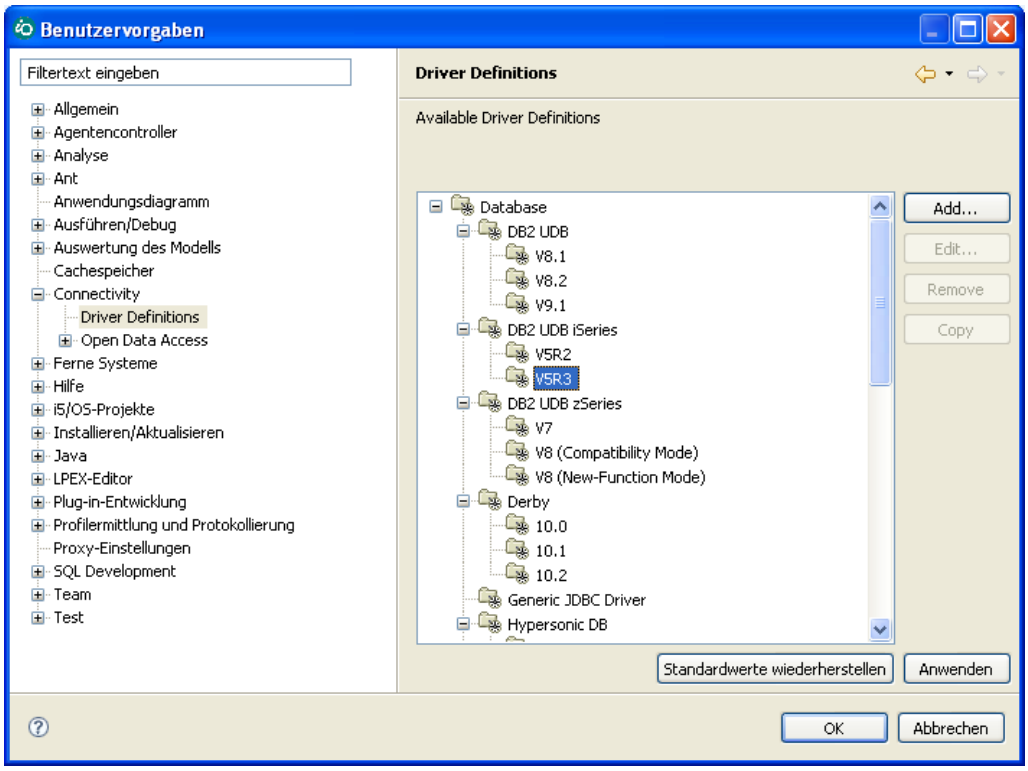
Erweitern Sie den Eintrag „Connectivity“.

Wählen Sie die Option „Driver Definitions“.

Selektieren Sie die für Ihr System zutreffende Version der DB2/UDB für System i bzw. iSeries.

Anmerkung

Je nach verwendeter Version von DTP kann es sein, dass nicht die aktuelle Version der DB2/UDB angezeigt wird. Nutzen Sie in einem solchen Fall den aktuellsten der aufgelisteten Datenbanktreiber!



DB2 UDB iSeries

Klicken Sie nach der Auswahl des Treibers auf die Schaltfläche „Add“.

Nun werden die verfügbaren Treiber aufgelistet. In unserem Beispiel stehen uns verschiedene Treiber zur Verfügung. Wir wählen den Treiber „AS/400 Toolbox for Java “ aus.

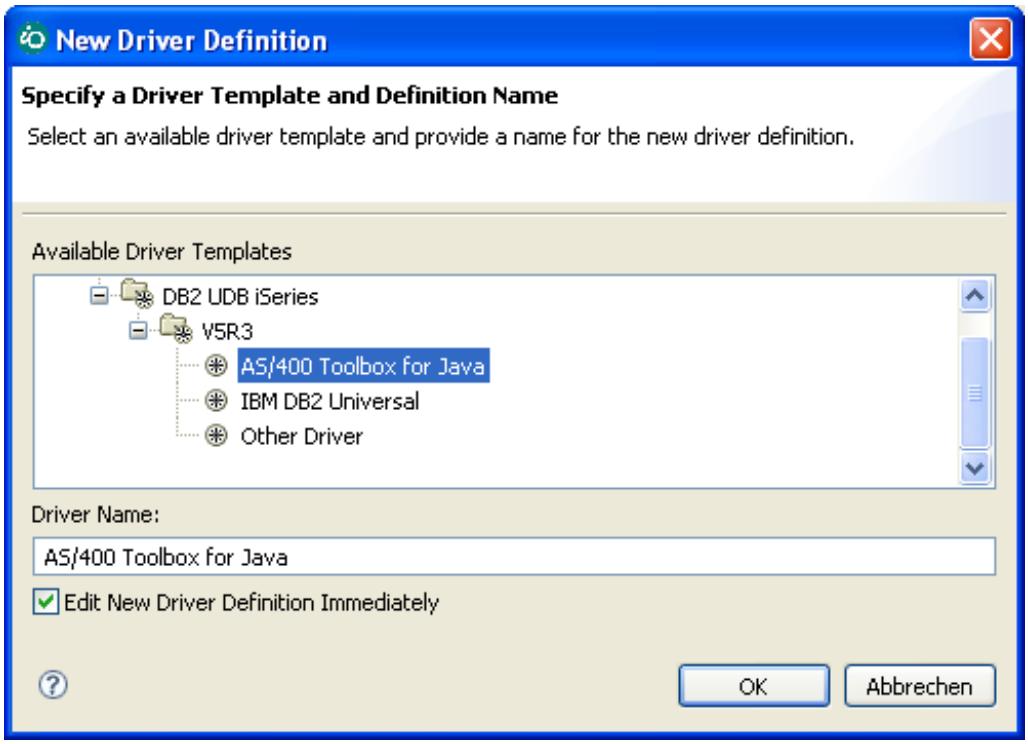
Erweitern Sie den Eintrag der Version der DB2/UDB (in unserem Beispiel V5R3) durch einen Klick auf das „+“-Zeichen.

Selektieren Sie in der Auflistung der verfügbaren Treiber den Treiber „AS/400 Toolbox for Java“. Damit wird dieser in das Feld „Driver Name“ übernommen.

Aktivieren Sie die Option „Edit New Driver Definition Immediately“.

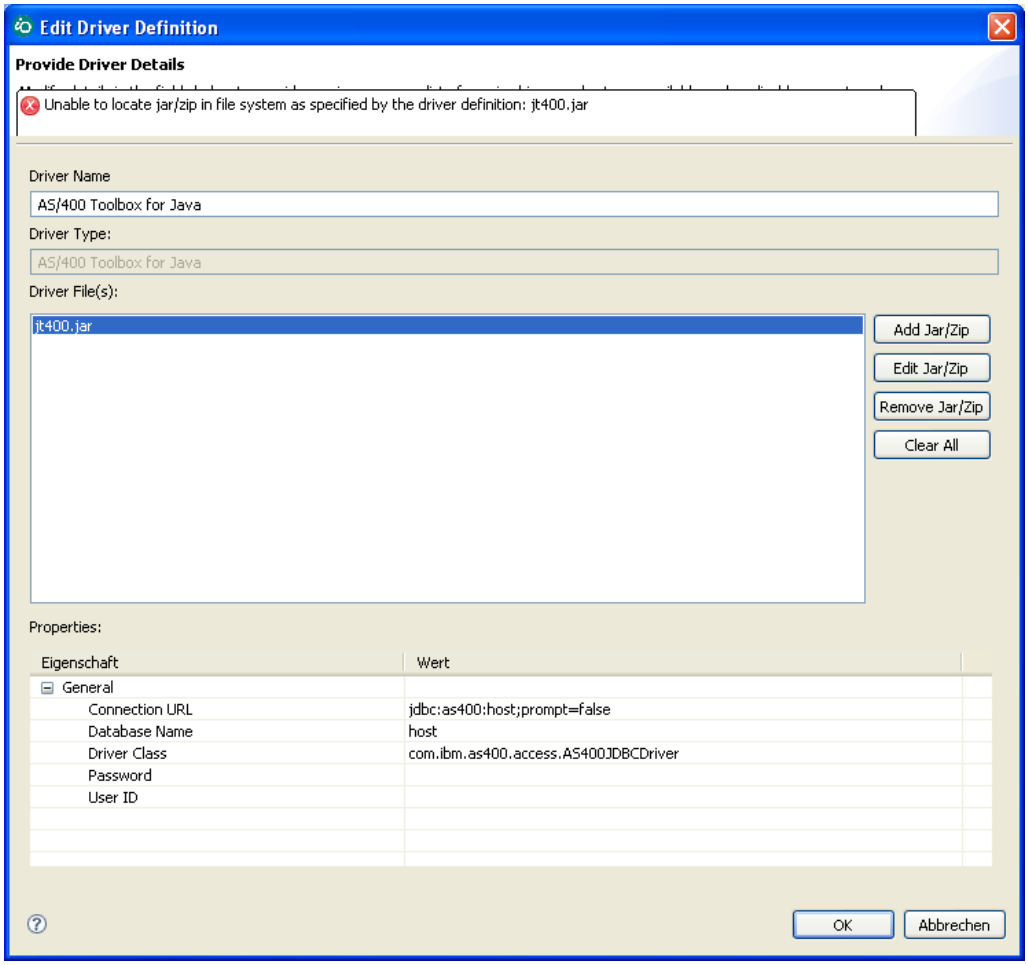
7.4.6
Seite 12

Bestätigen Sie die Auswahl mit einem Klick auf die Schaltfläche „OK“.



Toolbox für Java

Leider ist die Location dieses Treibers variabel. Das erklärt auch die Fehlermeldung, die wir in der nächsten Anzeige sehen. „Unable to locate jar/zip in file system“ sagt hier aus, dass die Location nicht angegeben wurde. Das ändern wir jetzt. Dazu geben wir den Speicherort der jt400.jar an.



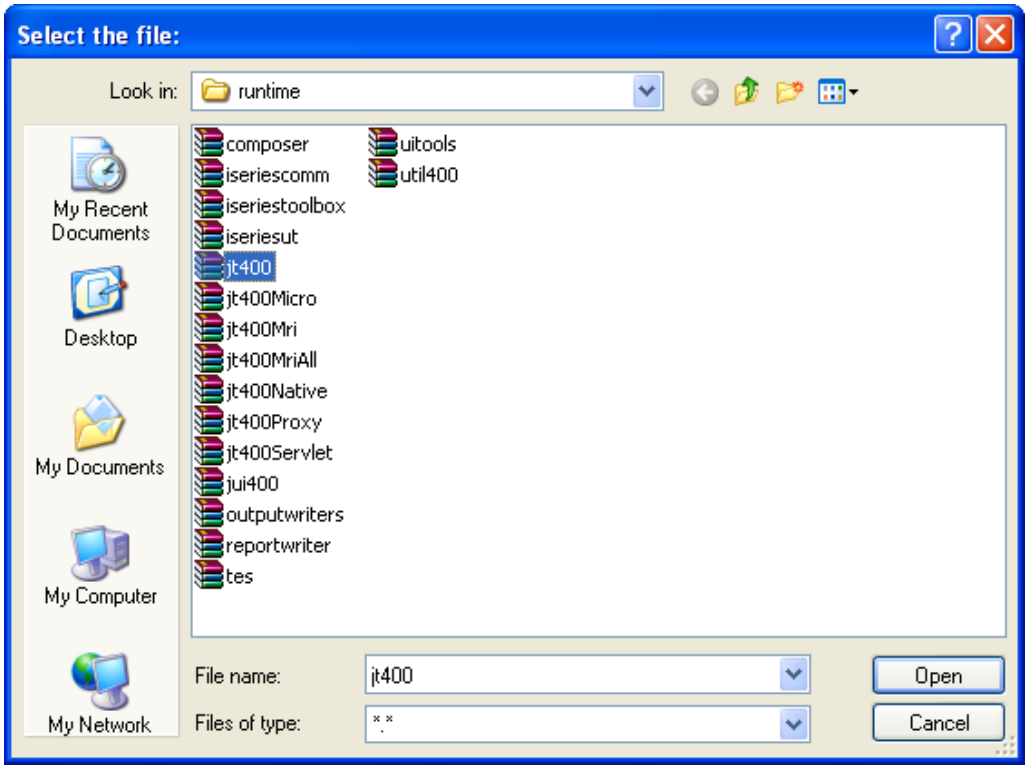
Fehlerhinweis

Markieren Sie den Eintrag „jt400.jar“ in der Auflistung der Driver Files.

Klicken Sie auf die Schaltfläche „Edit Jar/Zip“.

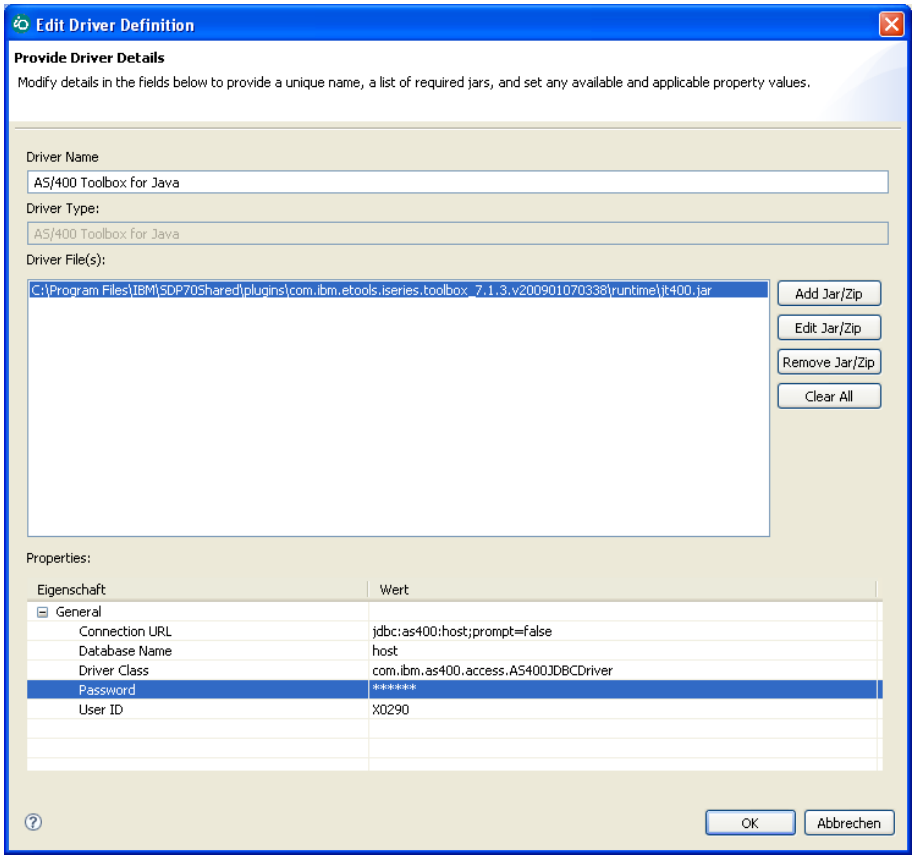
Suchen Sie eine möglichst aktuelle Version der jt400.jar und übernehmen Sie diese.

7.4.6
Seite 14



JT400-Auswahl

Jetzt muss noch der Zugriff auf die DB2/UDB definiert werden. Dazu geben wir den Datenbanknamen, den Benutzer und das zugehörige Kennwort im Bereich „Properties“ ein. Ein Beispiel dazu finden Sie in der folgenden Abbildung.



Treiberdetails

Bestätigen Sie die Eingaben mit einem Klick auf „OK“.

In der Anzeige der Benutzervorgaben („Driver Definition“ klicken wir abschließend auf die Schaltfläche „Anwenden“ und dann auf „OK“.

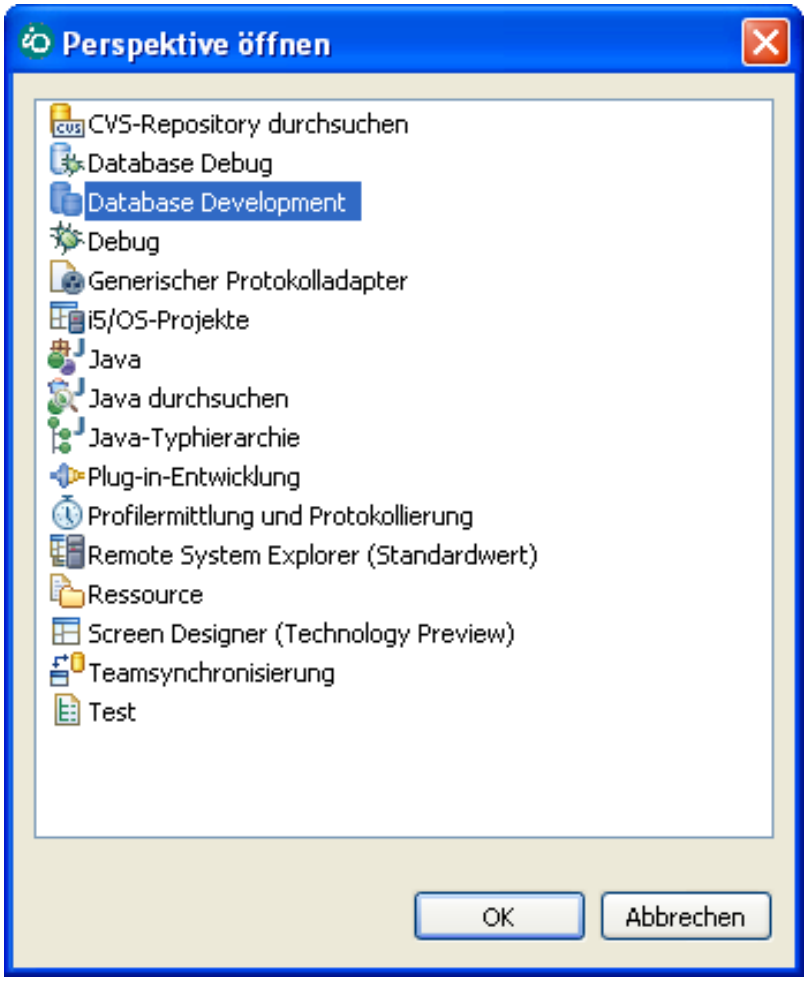
Damit haben wir die Voraussetzungen für den Zugriff auf die DB2/UDB des System i abgeschlossen.

Unter anderem finden wir jetzt in der Übersicht der Perspektiven die Perspektive „Database Development“, die wir für die weitere Arbeit nutzen werden.

Wählen Sie die neue Perspektive „Database Development“ aus.

Das können Sie z. B. über die Menüoption „Fenster / Perspektive öffnen / andere“ durchführen.

7.4.6
Seite 16



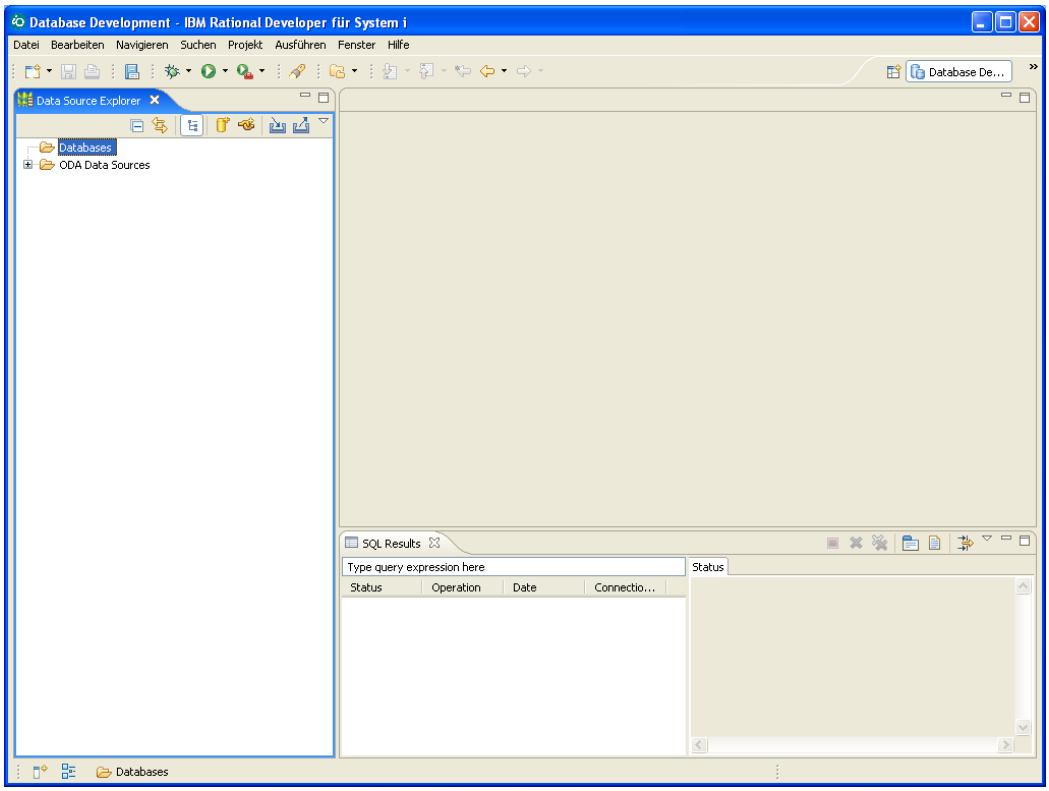
Perspektive auswählen

Die neue Datenperspektive besteht aus verschiedenen Anzeigenbereichen. Links befindet sich der „Data Source Explorer“ – die Grundlage für das Arbeiten mit den Inhalten einer Datenbank.

SQL-Result-Anzeigen liefern Ergebnisse zu SQL-Anweisungen. Und im Editierbereich können wir SQL-Eingaben vornehmen und SQL-Quellcodes bearbeiten.

7.4.6.1 Erstellen eines Verbindungsprofils

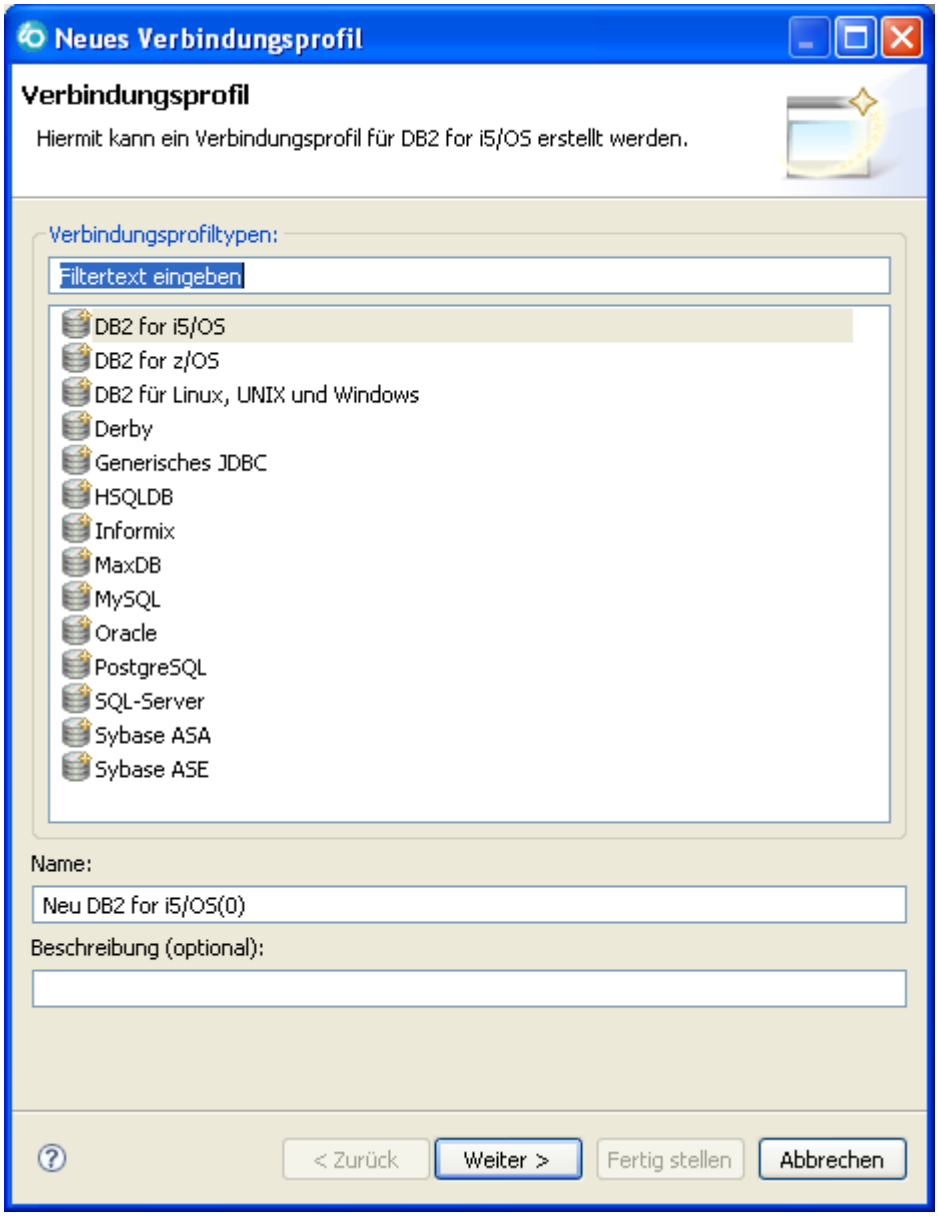
Als Erstes müssen wir in dieser Perspektive ein Verbindungsprofil „Connection Profile“ erstellen.



Data Source Explorer

- Wählen Sie in der Sicht „Data Source Explorer“ den Eintrag „Databases“ mit der rechten Maustaste aus.
- Selektieren Sie die Option „Neu“.
  Mit dieser Auswahl gelangen wir in eine Übersicht der verfügbaren oder besser gesagt der in RDI installierten Datenbanktreiber für die unterschiedlichen Datenbanken. Unter anderem finden wir dort auch die notwendigen Treiber für die DB2/UDB für i5/OS.
- Selektieren Sie den Eintrag „DB2 for i5/OS“.

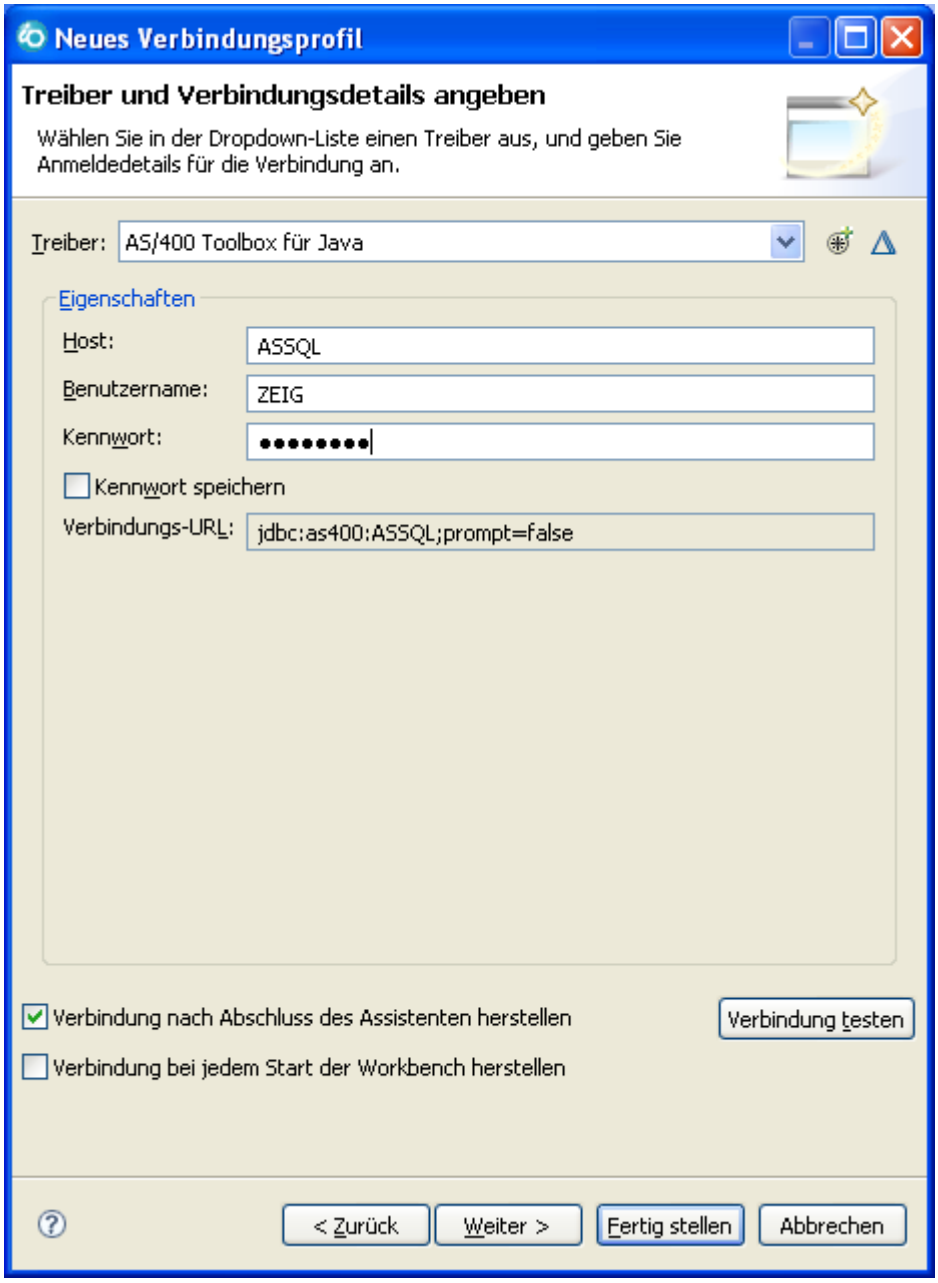
7.4.6
Seite 18



Verbindungsprofil wählen

- Bestätigen Sie die Auswahl mit einem Klick auf „Weiter“.

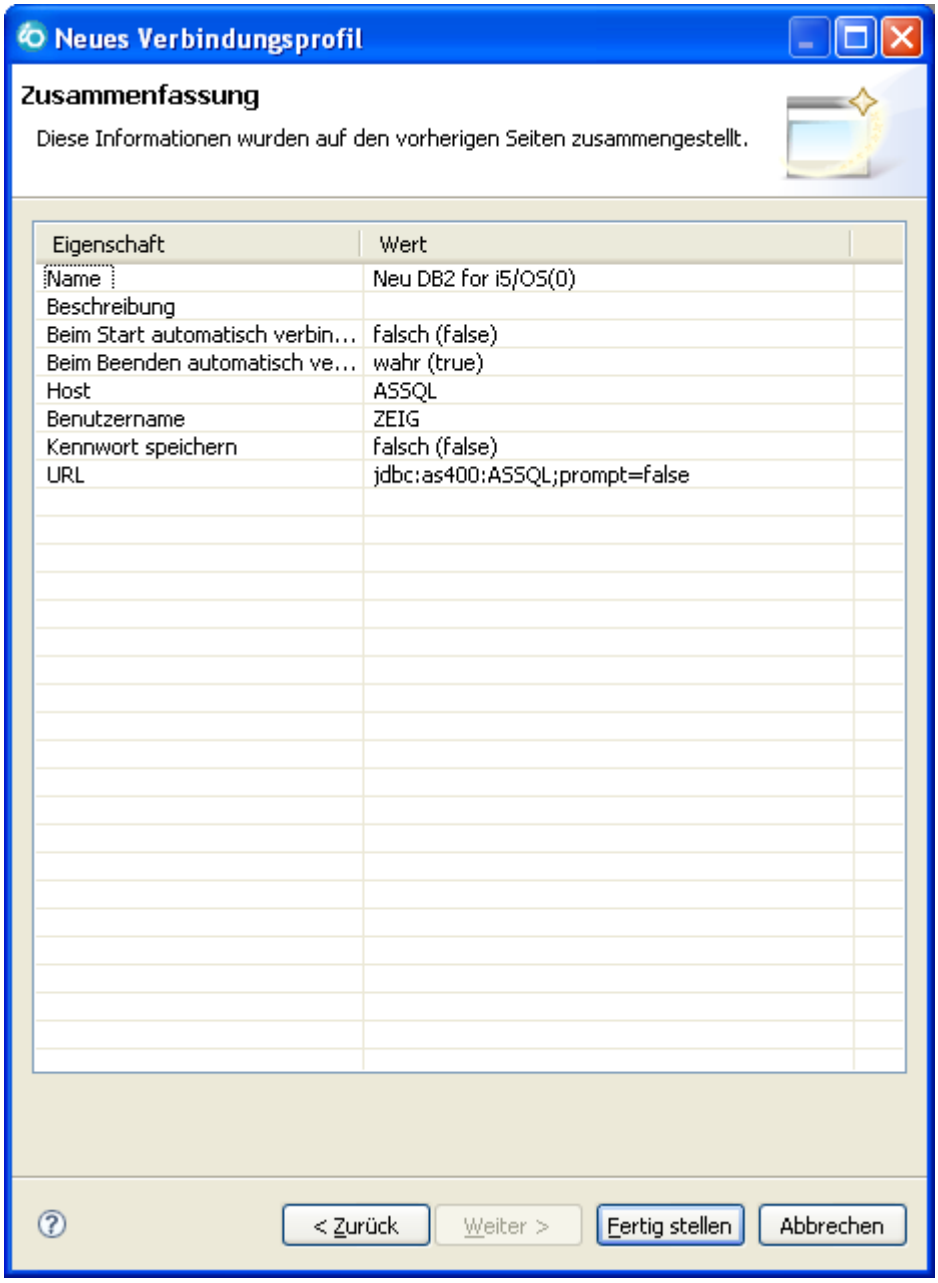
In der nächsten Anzeige werden wir zur Eingabe der Verbindungsmerkmale in Form des Hostnamens und eines Benutzers samt Kennwort aufgefordert. Geben Sie hier den Hostnamen für das System i und ein IBM-i-Benutzerprofil an.



Verbindungsdetails

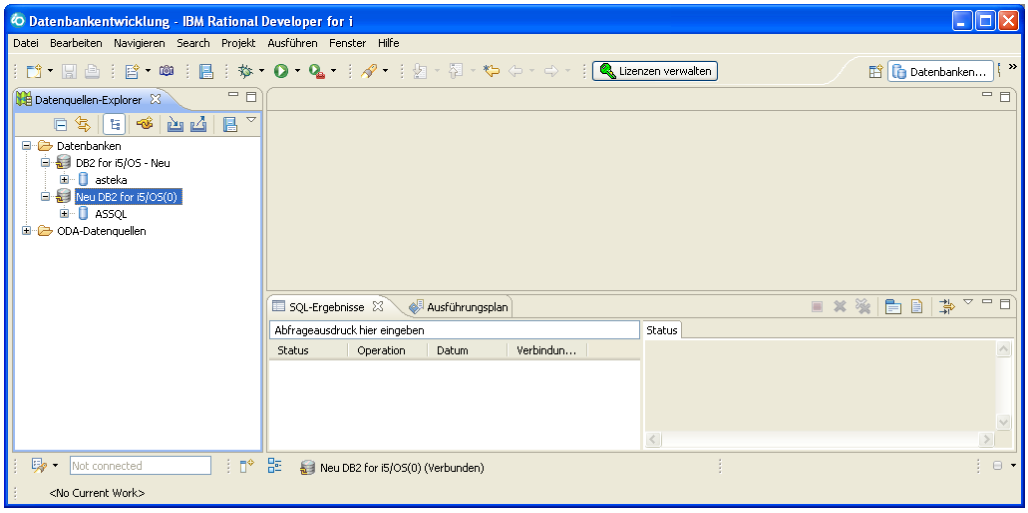
- Mit einem Klick auf „Weiter“ können wir die Angaben nochmals einsehen. Anschließend erscheint eine Anzeige, wie sie in der nachfolgenden Abbildung zu sehen ist.

7.4.6
Seite 20



Zusammenfassung

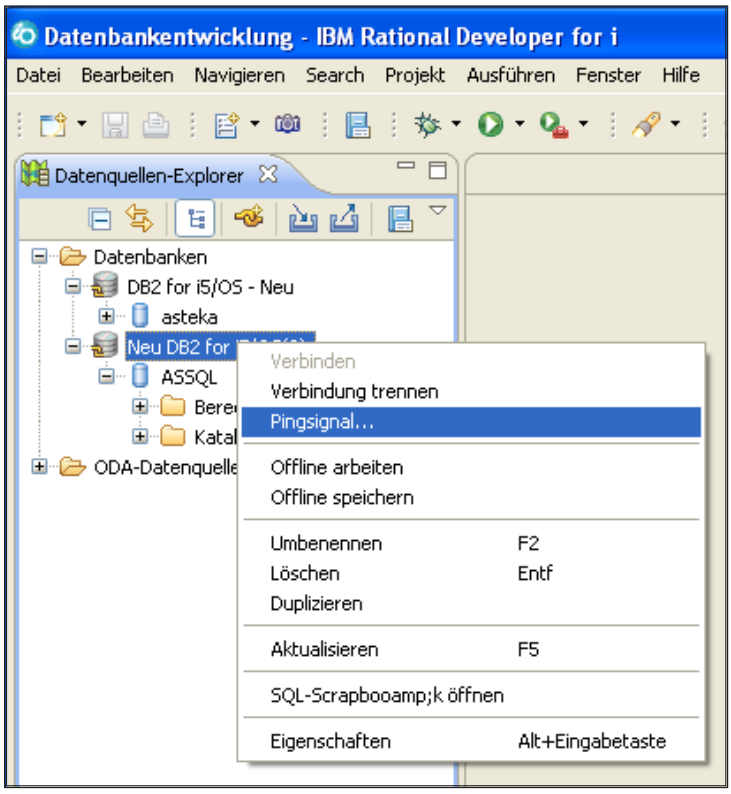
- Mit einem Klick auf „Fertigstellen“ wird die neue Datenbankverbindung generiert. Das Ergebnis sehen wir in der Navigationsübersicht. Dort sollte sich nun die neue Datenbankverbindung befinden.



Neue Verbindung wurde definiert

Es bietet sich an, die Verbindung zu testen. Dazu können wir uns einer Pingfunktion bedienen, die in RDI integriert ist.

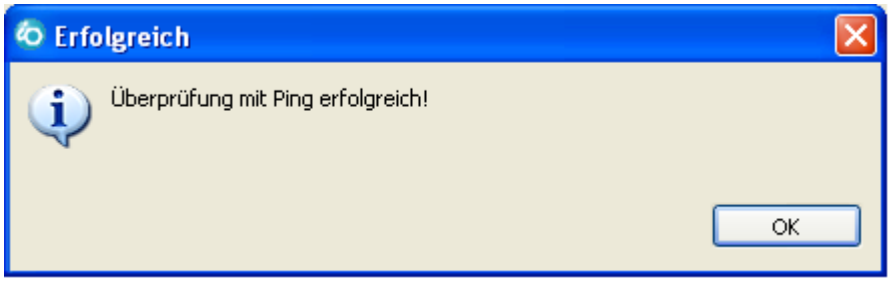
- Die Verbindung testen wir, indem wir im linken Navigationsbereich mit der rechten Maustaste auf den neuen Eintrag klicken. Daraufhin erscheint eine Auswahl.
- Selektieren Sie den Eintrag „Pingsignal ...“.



Pingsignal

7.4.6
Seite 22

Wenn die Konfiguration IP-seitig in Ordnung ist, sollte eine Nachricht erscheinen, wie sie in der folgenden Abbildung zu sehen ist.



Erfolgreiche Verbindung

Mit der Erstellung des Verbindungsprofils sind die ersten Konfigurationsschritte für das Arbeiten mit dem Datentool von Eclipse abgeschlossen. Es bildet die Grundlage für das Arbeiten mit der jeweiligen Datenbank.

Auf Basis der „jeweiligen Datenbank“ lassen sich die eigentlichen Arbeiten ausführen, die den Anwendungsentwickler in die Lage versetzen, die Funktionen und Werkzeuge in diesem Tool für die Anwendungsentwicklung und die Arbeit mit der DB2/UDB des System i zu verwenden.

7.4.6.2 Arbeiten mit SQL

Die Voraussetzungen für das Arbeiten mit der Datenbank des System i haben wir mit den vorhergehenden Schritten geschaffen. Jetzt gilt es, die SQL-Tätigkeiten mit den RDi-Funktionen nutzen zu können.

Achtung !

Grundsätzlich an dieser Stelle der Hinweis, dass RDi bzw. die Datenperspektive mit allen Inhalten der Datenbank arbeiten. Damit geht einher, dass bei der Verbindungserstellung und beim Arbeiten mit den Datenbankinhalten unter Umständen eine größere Anzahl von Informationen zwischen dem Hostsystem, auf dem die Datenbank läuft, und dem RDi-Client übertragen werden muss.

Eclipse bietet für diesen Fall die Möglichkeit, auf ein Subset der Datenbankinformationen zuzugreifen. Dieses Subset kann man individuell mit Hilfe von Filtern definieren. Der Vorteil: Schnellere Zugriffszeiten bei der Beschaffung der Grundinformationen zur Datenbank bzw. zu deren Elementen.

Mit Hilfe von Filtern lassen sich über die Eigenschaften der Datenbankverbindung auch gezielt Einschränkungen für die benötigten Inhalte der Datenbank definieren.

Unter anderem stehen in der Datenperspektive folgende Filter für unterschiedliche Datenbankbereiche zur Verfügung:

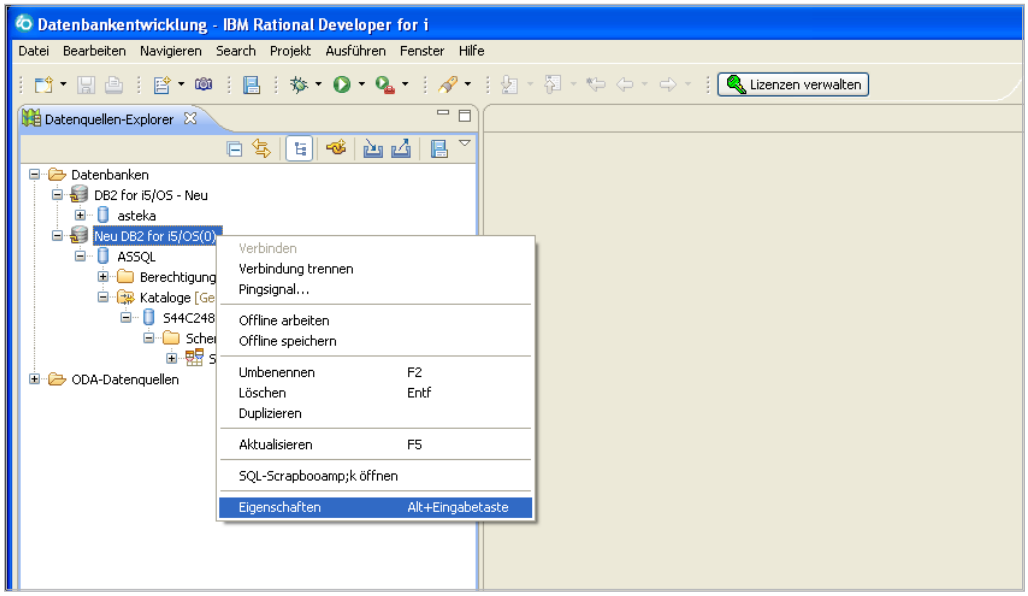
- Schemafilter
- Tabellenfilter

Gehen wir einmal davon aus, dass wir zunächst mit dem Inhalt einer Bibliothek bzw. eines Schemas arbeiten wollen. Dazu können wir einen Schemafilter verwenden, der als Einschluss- oder Ausschlussfilter definiert werden kann.

Den Filter erstellen wir auf Basis des Verbindungsprofils. In dessen Eigenschaften finden wir u. a. auch die verschiedenen Filterdefinitionen.

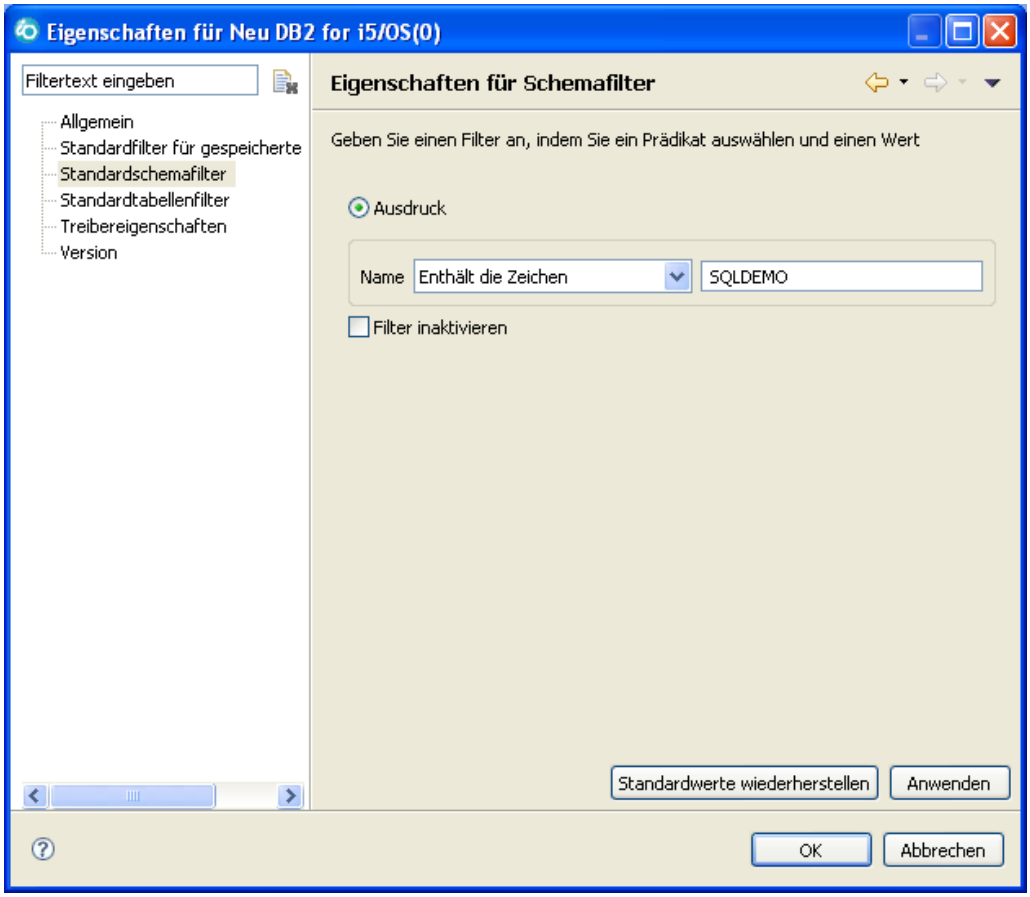
7.4.6.2.1 Erstellen eines Filters

Um einen Schemafilter zu erstellen, klicken wir mit der rechten Maustaste im Navigationsbereich auf das Verbindungsprofil. Es öffnet sich ein Fensterbereich, in dem wir u. a. auch die Eigenschaften des Profils verwalten können.



Eigenschaften des Verbindungsprofils

- Klicken Sie in der Navigationssicht mit der rechten Maustaste auf das Verbindungsprofil.
- Wählen Sie im neuen Anzeigenbereich die Option „Eigenschaften“.
- Selektieren Sie im nächsten Fenster den Bereich „Standardschemafilter“.
- Deaktivieren Sie die Option „Filter inaktiviert“.
- Wählen Sie die Option Name mit „Enthält die Zeichen“.
- Geben Sie im Filterdefinitionsbereich den Namen eines Schemas an. In unserem Beispiel habe ich das Schema „SQLDEMO“ verwendet, das die Beispieltabellen und SQL-Definitionen beinhaltet.



Schemafilter

Anmerkung

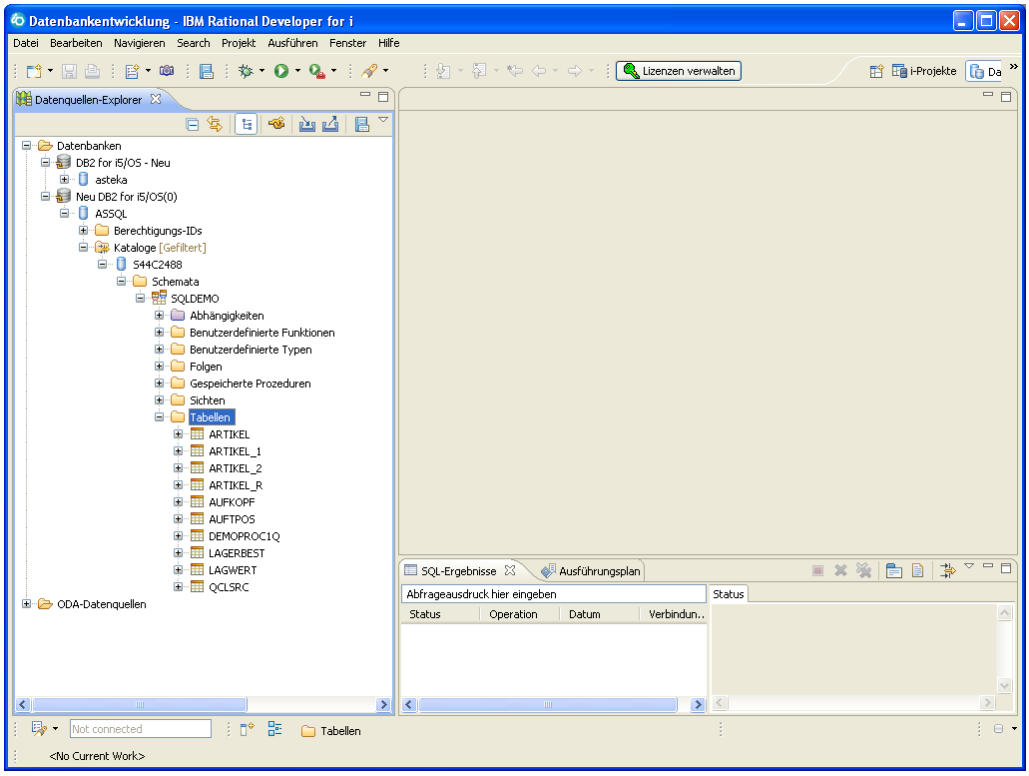
RDI bietet neben der Option „Enthält die Zeichen“ auch noch andere Selektionsoptionen, so z. B.:

- Beginnt mit den Zeichen
- Endet mit den Zeichen
- Beginnt nicht mit den Zeichen
- Endet nicht mit den Zeichen
- Enthält nicht die Zeichen

Bestätigen Sie die Auswahl mit einem Klick auf die Schaltfläche „Anwenden“ und anschließend auf „OK“.

Mit diesen Teilschritten verändert sich der Eintrag in der Navigationsleiste leicht. Wenn Sie den entsprechenden Bereich aktualisieren, finden Sie im Unterordner „Kataloge“ den Hinweis „gefiltert“. Daran erkennt man sofort, dass in diesem Bereich nur ein Subset der Datenbankinhalte dargestellt wird. Unterhalb des Bereichs „Schema“ finden wir nun die Inhalte der Schemata, die den Selektionskriterien entsprechen. In unserem Beispiel ist dies der Inhalt des Schemas „SQLDEMO“. Hier finden wir z. B. unterhalb des Abschnitts „Tabellen“ die in dem Schema SQLDEMO enthaltenen Tabellen.

- Erweitern Sie den Eintrag „Kataloge“.
- Erweitern Sie den Eintrag für das Schema (in unserem Beispiel SQLDEMO).
- Erweitern Sie den Eintrag „Tabellen“.

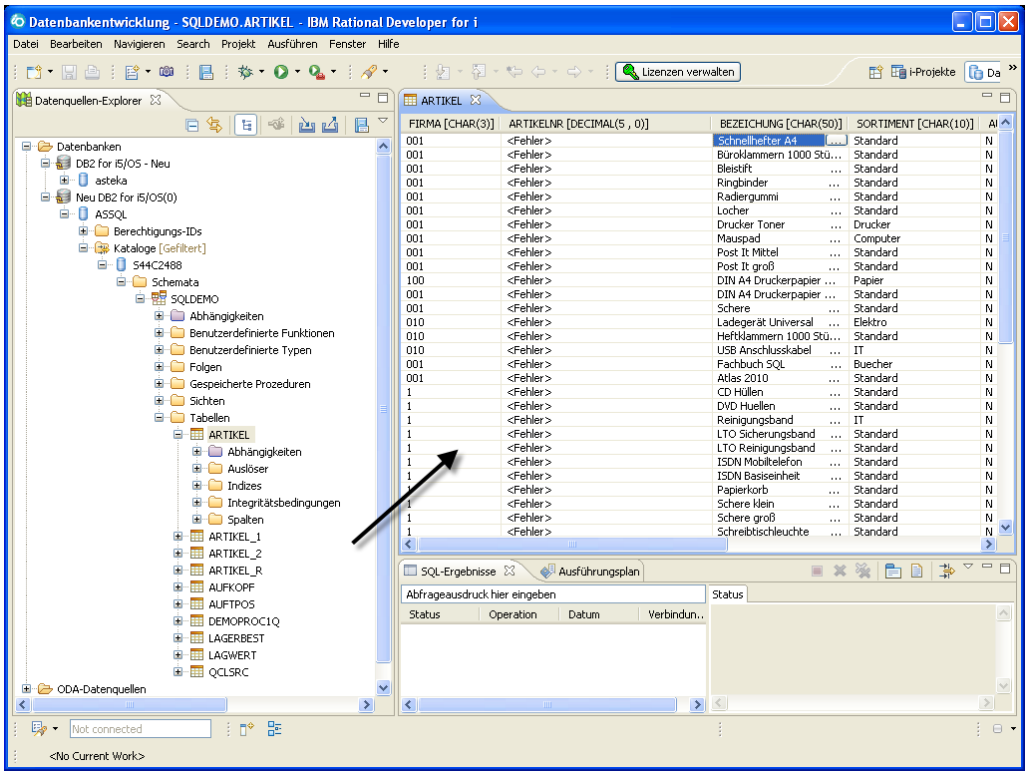


Tabellen in einem Schema

Anmerkung

Durch einen definierten Tabellenfilter kann der hier angezeigte Bestand an Tabellen auch eingeschränkt sein. Die Tabellenfilter werden ebenfalls im Bereich „Eigenschaften“ des Verbindungsprofils definiert. Die Definition verläuft dabei vergleichbar mit der Definition der Schemafilter.

Als erstes Beispiel für die Arbeit mit der Datenbank beschäftigen wir uns mit dem Inhalt einer Tabelle. Diese wird mit der rechten Maustaste markiert, anschließend lässt sich festlegen, wie mit dem Inhalt der Tabelle verfahren werden soll. RDI bietet u. a. die Möglichkeit, mit dem Inhalt der Tabelle direkt zu arbeiten. Die folgende Abbildung zeigt den zum Bearbeiten ausgewählten Inhalt der Tabelle „Artikel“.



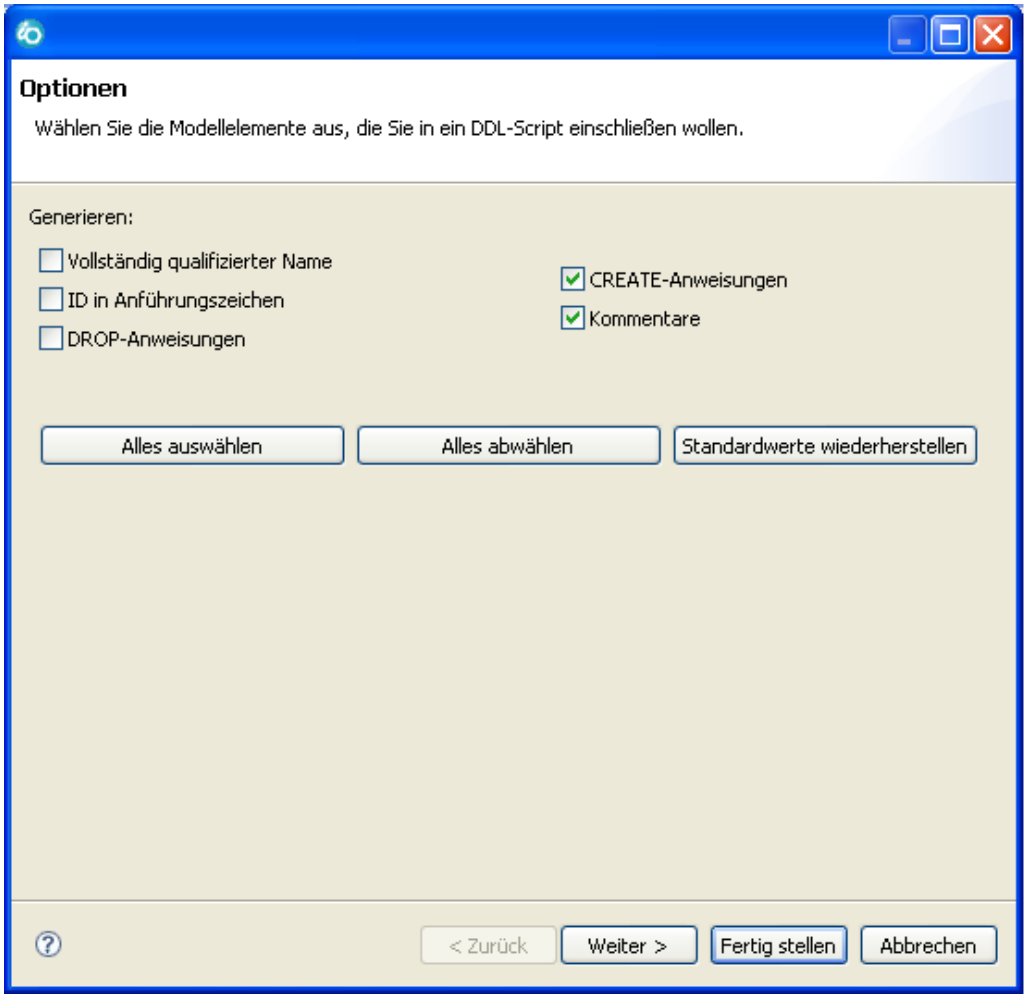
Bearbeiten des Inhalts einer Tabelle

Die Datenbank-Tools von Eclipse bieten verschiedene Werkzeuge für das Arbeiten mit einer Datenbank. System-i-Anwender werden den SQL-Editor mit den nativen Werkzeugen der 5250-Ebene vergleichen und sich gewiss mit dessen Verwendung anfreunden können. Alternativ dazu steht ein weiteres Tool in Form des SQL-Scrapbook zur Verfügung.

Schauen wir uns den SQL-Editor bzw. den Assistenten für SQL einmal genauer an. Dazu erstellen wir zunächst eine einfache Tabelle.

- Klicken Sie mit der rechten Maustaste auf den Eintrag „Tabellen“ in der Navigationssicht.
- Wählen Sie die Option „Neue Tabelle“.
- Im Definitionsbereich für die Optionen können wir unterschiedliche Vorgaben machen, wie die nachfolgende Abbildung zeigt. Passen Sie die Vorgaben an Ihre Wünsche bzw. Anforderungen an.

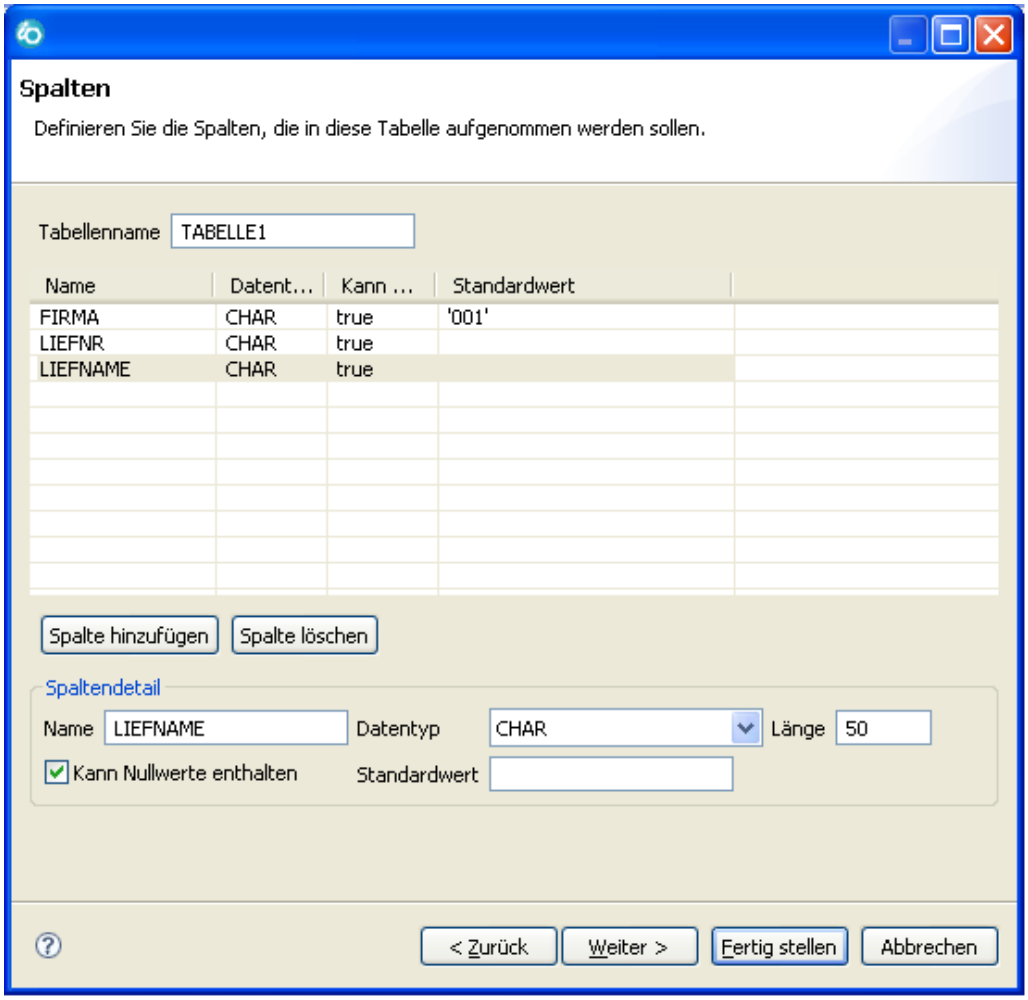
7.4.6
Seite 28



SQL-Optionen

- Bestätigen Sie die Angaben mit einem Klick auf die Schaltfläche „Weiter“.

Damit gelangen wir in den Anzeigenbereich des Assistenten für die Erstellung einer neuen Tabelle. Hier müssen wir zunächst den Namen der Tabelle angeben. In unserem Beispiel verwende ich „Tabelle1“. Über die Schaltfläche „Spalte hinzufügen“ können wir die gewünschten Spalten angeben. Diese verfügen natürlich über die gewohnten Merkmale wie Spaltenname, Art und Länge.

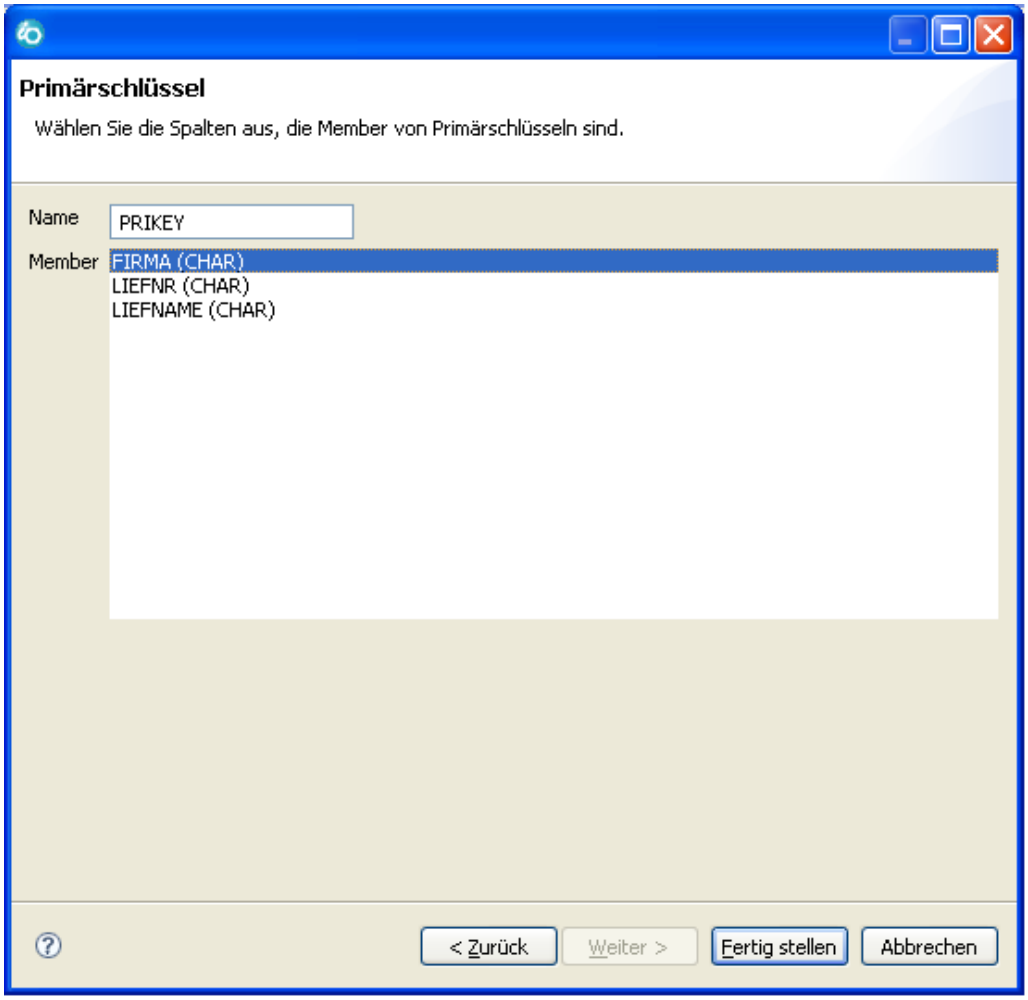


Erstellen einer neuen Tabelle

- Geben Sie als Tabellennamen „Tabelle1“ an.
- Klicken Sie auf die Schaltfläche „Spalte hinzufügen“.
- Überschreiben Sie im Feld „Name“ den vorgeschlagenen Spaltennamen mit dem gewünschten Namen für die neue Spalte.
- Geben Sie einen Datentyp und die zugehörige Länge ein.
- Wenn alle Spalten angegeben wurden, klicken Sie auf „Weiter“.

In der nächsten Anzeige bietet uns der Erstellungsassistent die Option, für die neue Tabelle einen Primärschlüssel zu definieren.

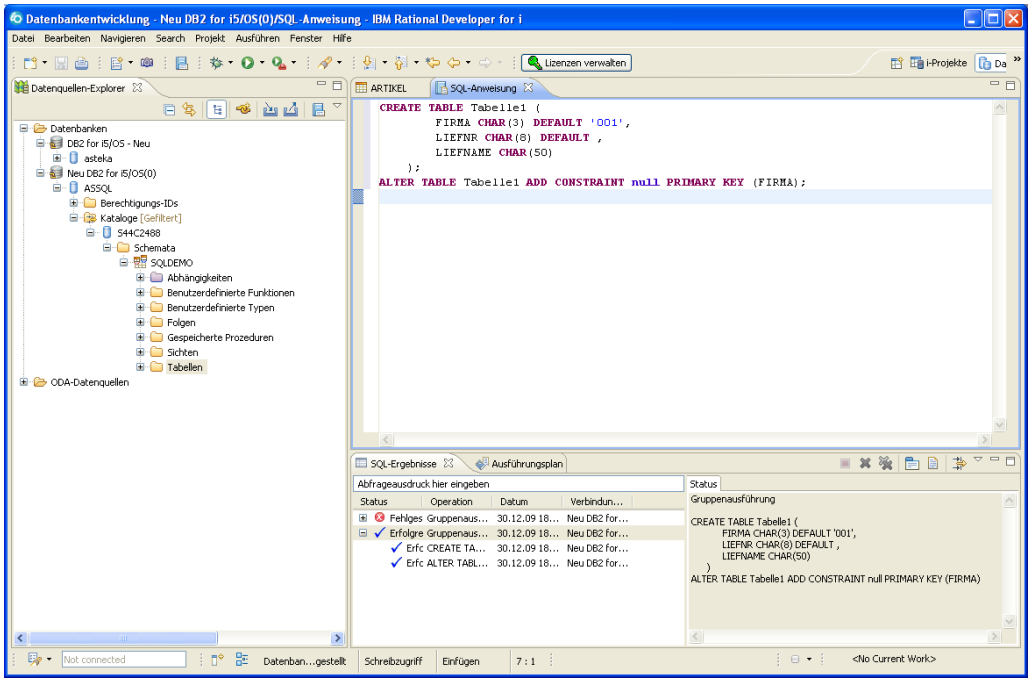
7.4.6
Seite 30



Primärschlüssel

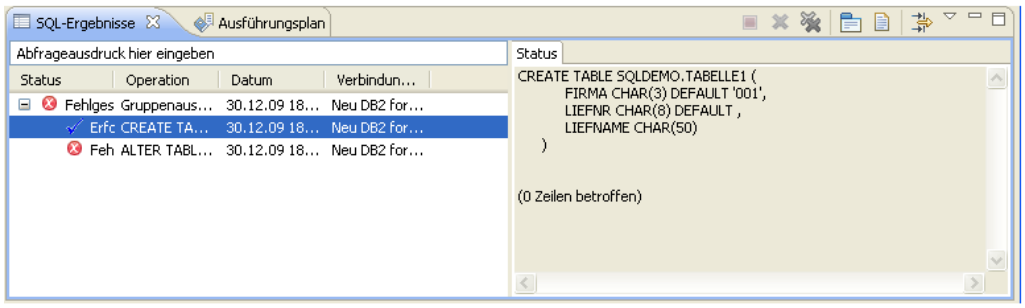
- Wählen Sie die Firmennummer als Primärschlüssel.
- Bestätigen Sie die Auswahl mit einem Klick auf „Fertig stellen“.

Damit verlassen wir den SQL-Assistenten und gelangen in den SQL-Editorbereich. Dieser beinhaltet unsere vorherigen Definitionen, die bei Bedarf hier noch angepasst werden können.



SQL-Syntax im Editor

Mit den Vorgaben lässt sich jetzt die neue Tabelle erstellen. Dazu kann man die Option „Alle Ausführen“ wählen. Damit wird die SQL-Anweisung auf dem Hostsystem (dem System i) abgesetzt und ausgeführt. Ein Statusbereich im unteren Abschnitt der Anzeige liefert das Ergebnis der SQL-Ausführung. Fehler werden mittels eines roten Symbols dargestellt. Mit einem Klick auf den jeweiligen Eintrag erhalten wir rechts im Statusbereich die Details zu dem selektierten Eintrag.

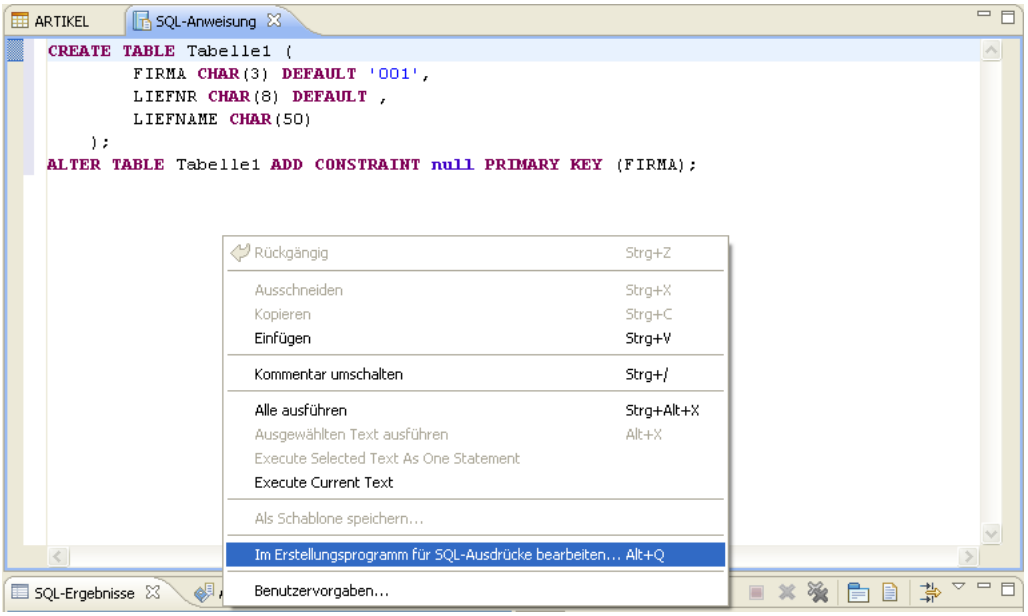


Ergebnis der SQL-Ausführung

7.4.6.3 SQL-Erstellungsprogramm

Alternativ zum SQL-Editor bietet die Datenbankperspektive auch ein SQL-Erstellungsprogramm, das hier und da auch als „SQL Query Builder“ bezeichnet wird. Dieses Erstellungsprogramm (eine andere Form des Erstellungsassistenten) bietet verschiedene Zusatzfunktionen, die insbesondere SQL-Neulingen die Arbeit erleichtern dürften.

Das SQL-Erstellungsprogramm kann aus dem SQL-Editorbereich heraus gestartet werden. Dazu genügt ein Klick mit der rechten Maustaste und die Auswahl der Option „Im Erstellungsprogramm für SQL-Ausdrücke bearbeiten“.



Erstellungsprogramm für SQL-Ausdrücke starten

Damit gelangen wir in die Startseite des SQL-Erstellungsprogramms. Hier ist per Standard der Assistent für die Verwendung einer SELECT-Anweisung gestartet. Natürlich kann auch jede andere Art einer SQL-Anweisung mit dem Erstellungsprogramm verwendet werden.

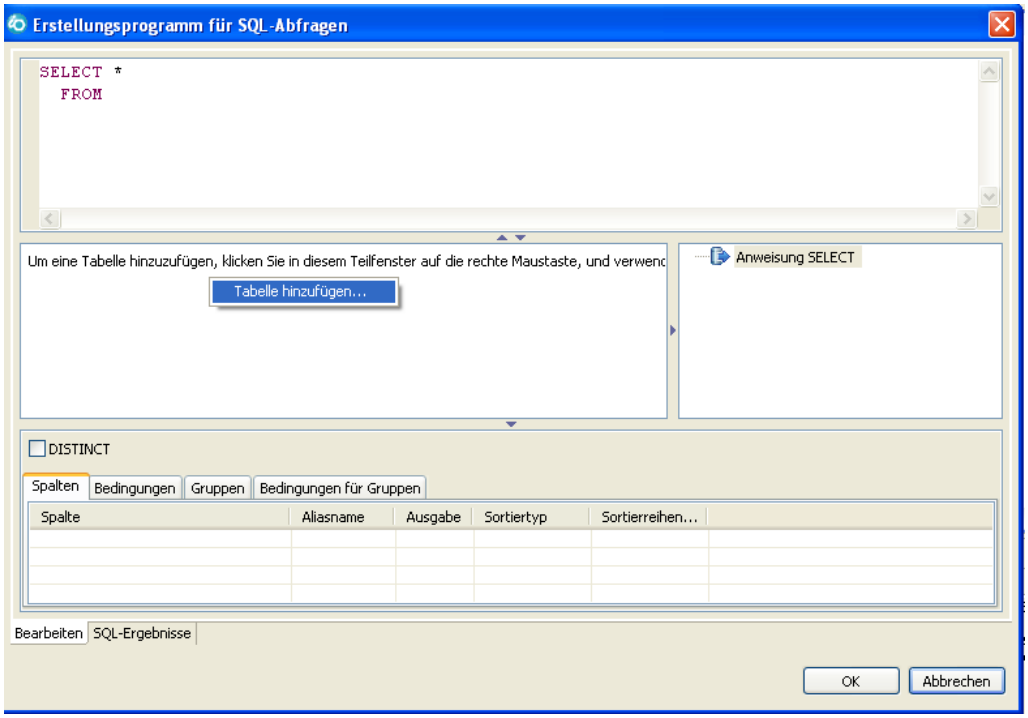


Tabelle auswählen

Schauen wir uns nun den Assistenten am Beispiel einer einfachen SELECT-Anweisung an, die den Inhalt der Tabelle „Artikel“ betrifft.

- Positionieren Sie den Cursor in den Abschnitt „Um eine Tabelle hinzuzufügen...“ und klicken Sie dort mit der rechten Maustaste.
- Erweitern Sie in der nächsten Anzeige das Schema.

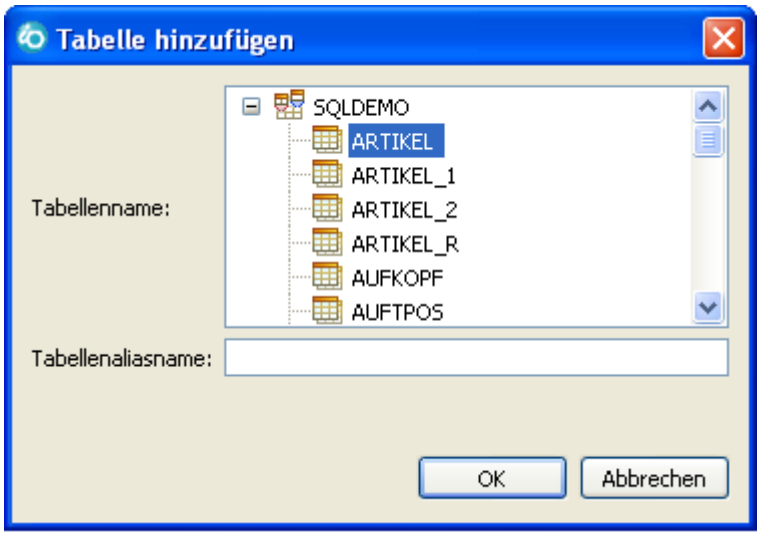
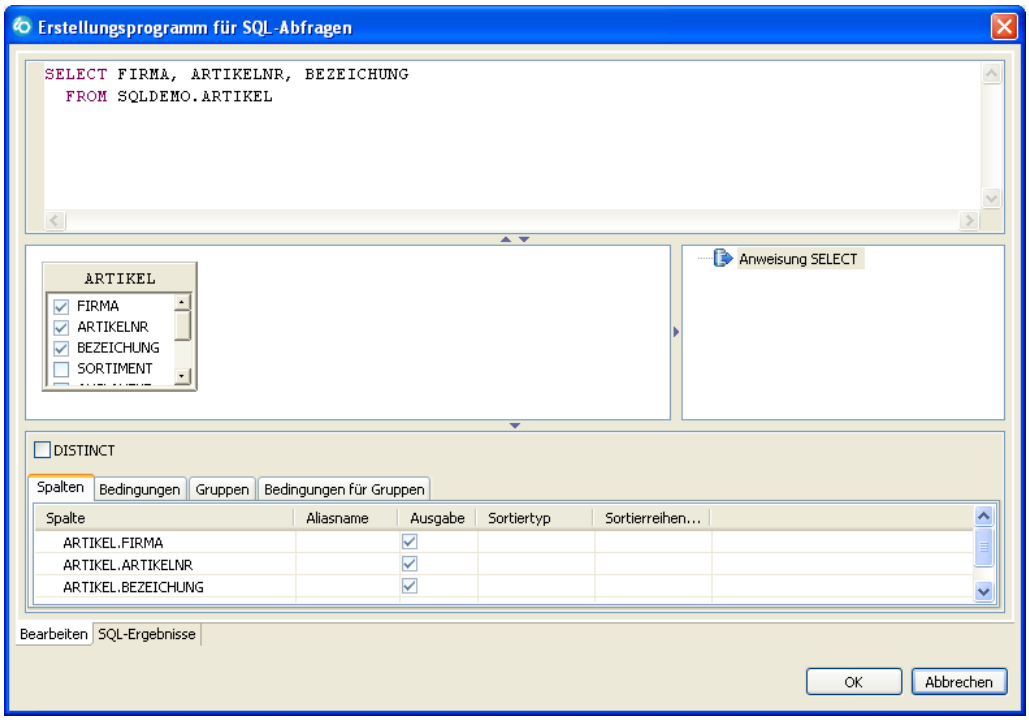


Tabelle auswählen

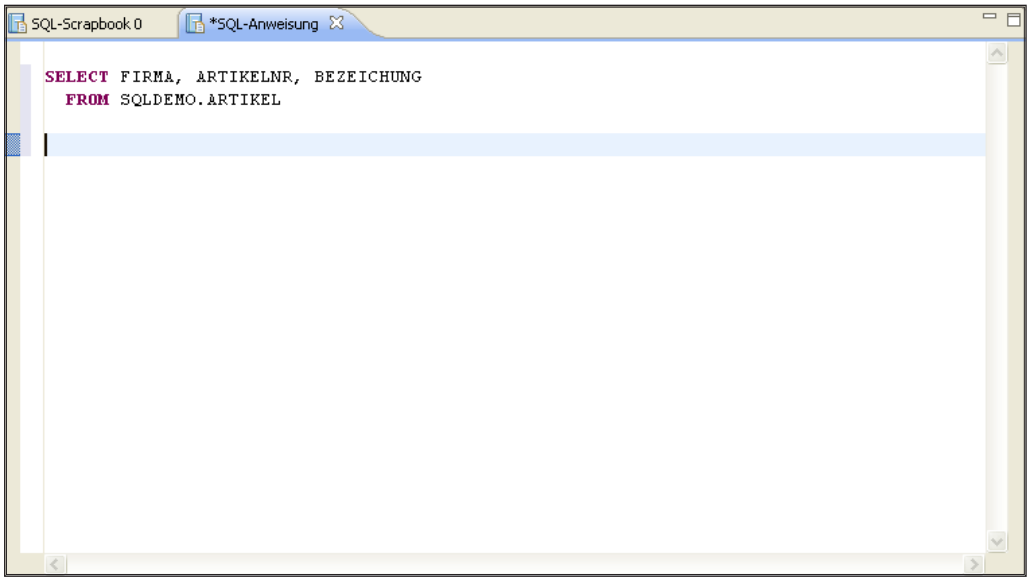
7.4.6
Seite 34

- Markieren Sie die Tabelle „Artikel“.
- Bestätigen Sie die Auswahl mit einem Klick auf „OK“.
  Damit werden die Spalten der Tabelle in diesem Abschnitt angezeigt. Diese lassen sich hier zur weiteren Verarbeitung selektieren.
- Wählen Sie einige Spalten mit einem Mausklick aus. Beachten Sie dabei die Veränderung in den Bereichen mit der eigentlichen SELECT-Anweisung und im Abschnitt „Spalten“ im unteren Bereich der Anzeige.



Markierte Spalten

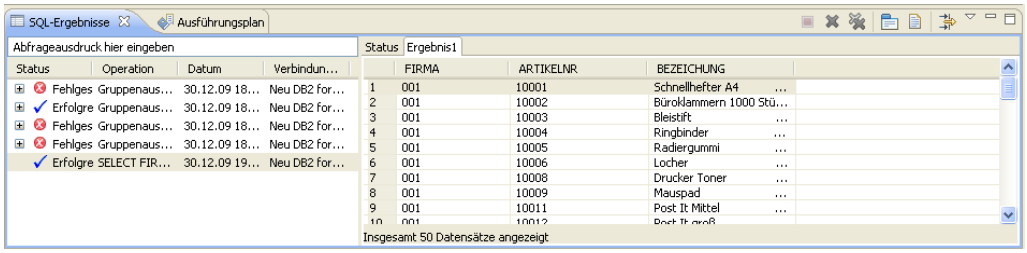
- Klicken Sie auf „OK“.
  Damit wechseln wir wieder in den SQL-Editorbereich, in dem jetzt die SELECT-Anweisung mit den ausgewählten Bereichen zu sehen ist.



Select-Anweisung

Mit der Tastenkombination „Strg + Alt + x“ kann der SQL-Code ausgeführt werden. (Alternativ dazu steht auch hier die Option „Alle ausführen“ nach einem Klick mit der rechten Maustaste zur Verfügung.)

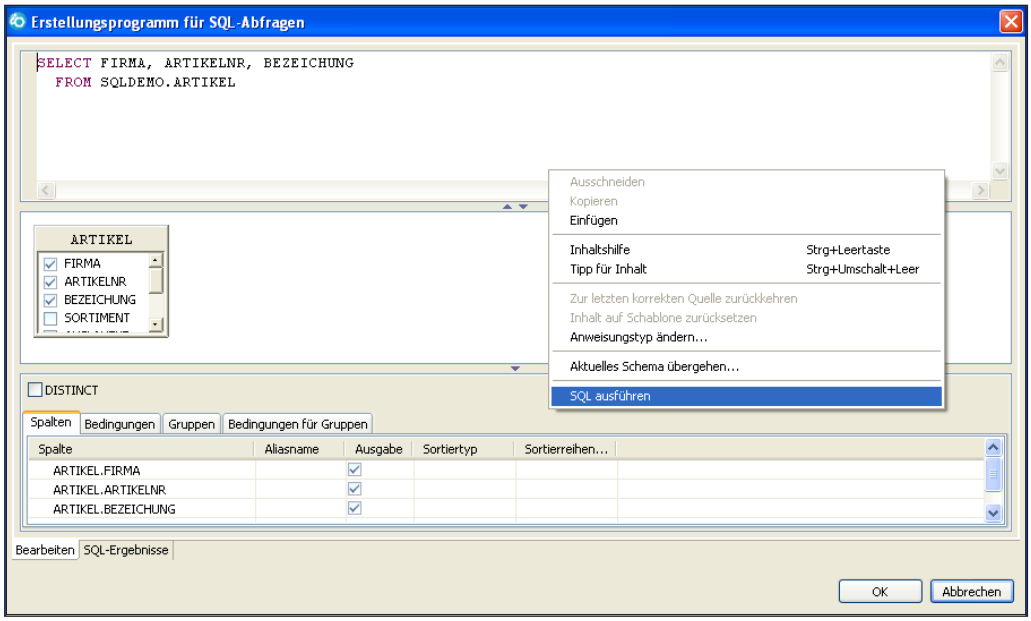
- Wählen Sie die Tastenkombination „Strg + Alt + x“. Das Ergebnis der Ausführung sehen wir unten rechts in der Ergebnissicht:



Ergebnis der SELECT-Anweisung

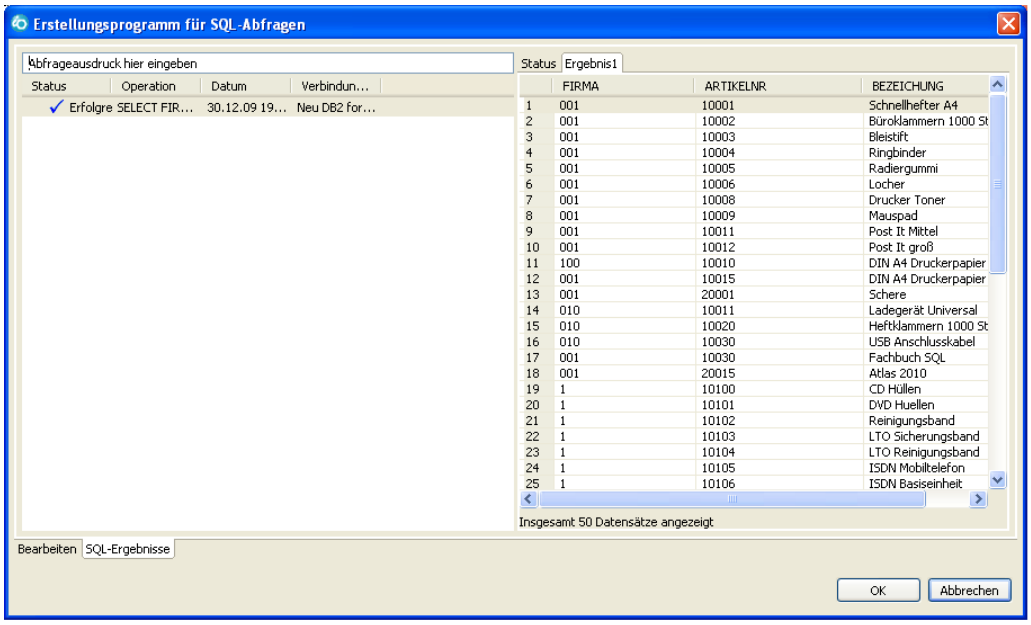
Anmerkung

Alternativ zu dieser Form der SQL-Ausführung kann im Erstellungsprogramm der SQL-Code nach einem Klick mit der rechten Maustaste in einem beliebigen Bereich innerhalb des Fensters für die Erstellung der SQL-Abfragen und der Auswahl der Option „SQL ausführen“ ausgeführt werden.



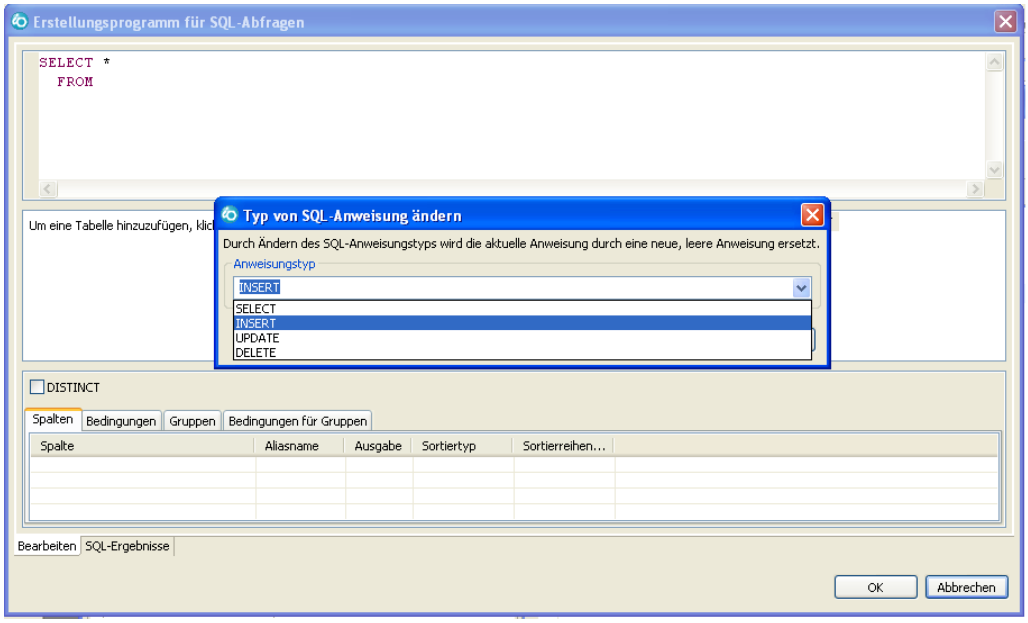
SQL ausführen

Das Ergebnis wird dann wie folgt ausgegeben:



Ergebnis der SQL-Ausführung

Um im Erstellungsprogramm eine andere Anweisung anzugeben, klickt man einfach mit der rechten Maustaste in den mit SELECT vorbelegten Bereich und wählt die Option „Anweisungstyp ändern“. Daraufhin erscheint ein Auswahlfenster mit den möglichen SQL-Anweisungen.



Anweisungstyp ändern

Anmerkung

Natürlich kann das Erstellungsprogramm auch zum Bearbeiten von SQL-Code genutzt werden. Dazu öffnet man den SQL-Code im Standard-SQL-Editor, markiert den SQL-Code und wählt nach einem Klick mit der rechten Maustaste innerhalb des Fensters der SQL-Abfragedefinition die Option „Im Erstellungsprogramm für SQL Ausdrücke bearbeiten“. Alternativ dazu kann man auch die Tastenkombination „Alt + Q“ nutzen.

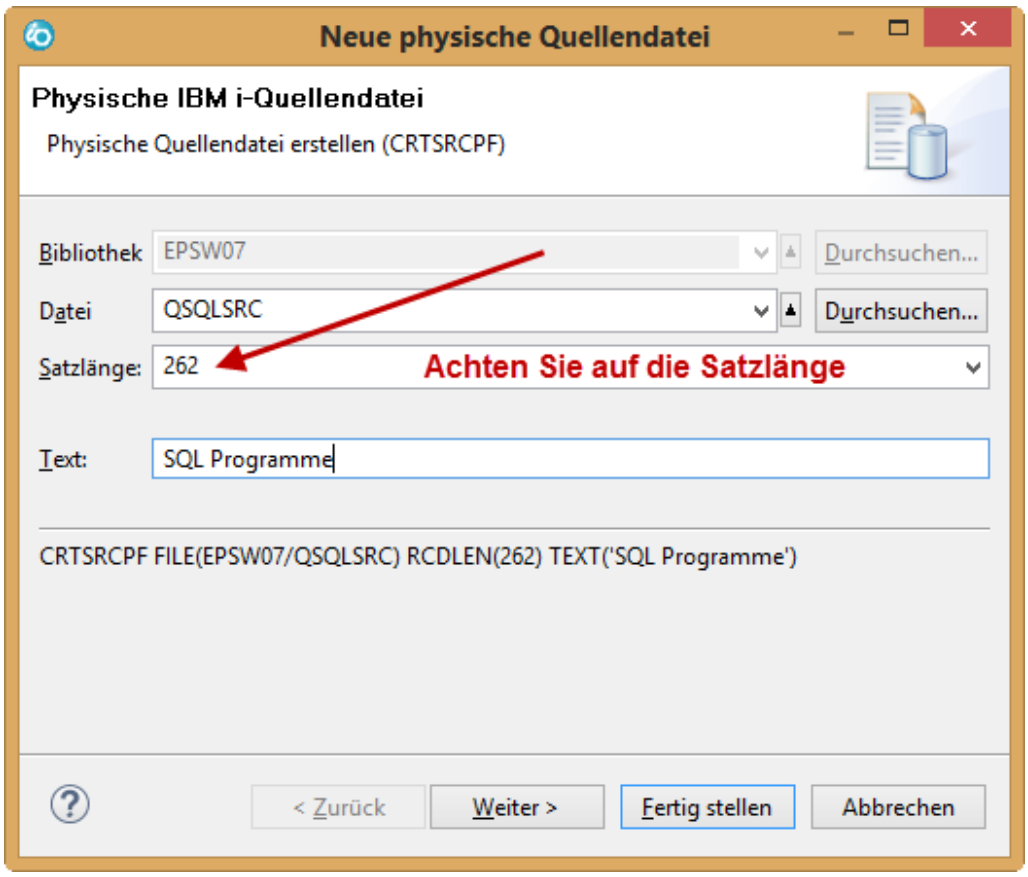
7.4.6
Seite 38

7.4.7 RDI und die SQL-Objekterstellung

Sie können Ihre SQL-Sourcen und Objekte aber auch mittels RDI erstellen. Dafür müssen die Quellen entweder in einer Quellendatei oder im IFS gespeichert werden. Schauen wir uns die Möglichkeiten nacheinander an.

7.4.7.1 Kompilierbefehle für Quellenteildateien erstellen

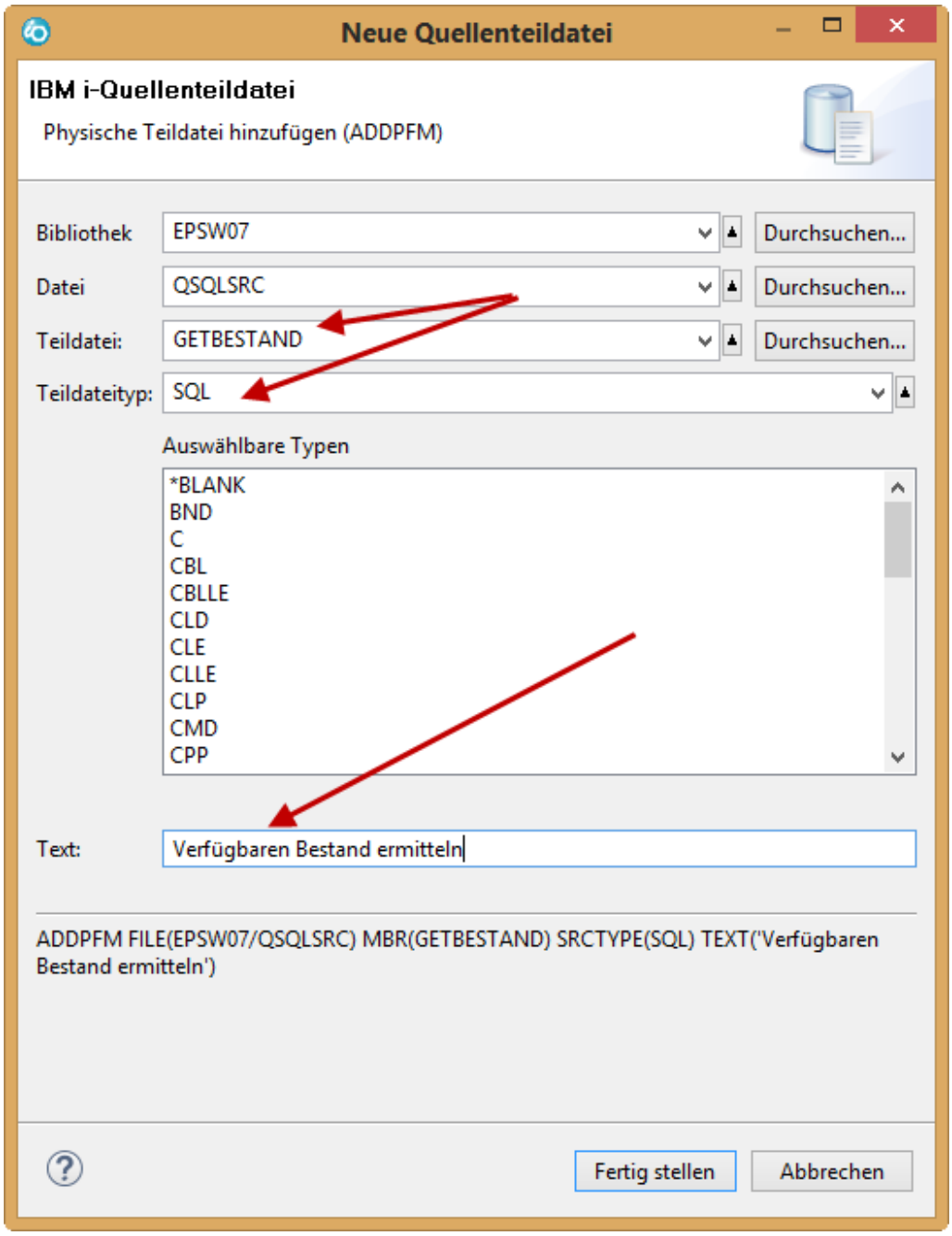
Sollen unsere SQL-Quellen in einer Quellendatei verwaltet werden, muss diese ggf. zunächst erstellt werden:



Die Quellendatei QSQLSRC

7.4.7
Seite 2

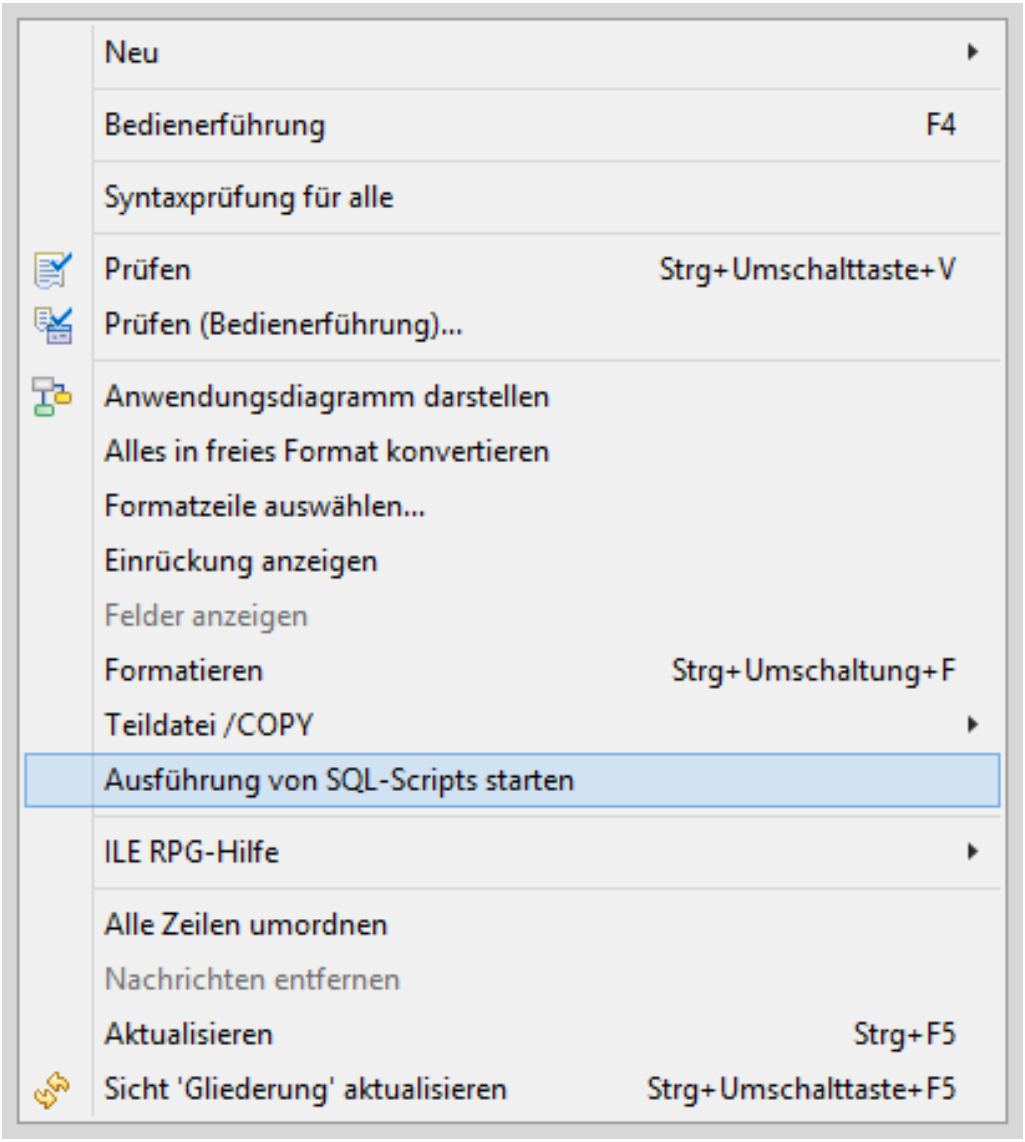
Es empfiehlt sich, für die Quellendatei eine Satzlänge größer 112 zu verwenden. So sind Sie in der Zeilenlänge weniger limitiert. Ich entscheide mich in der Regel für eine Satzlänge von 262 Zeichen, so dass ich 250 Zeichen (262 Zeichen abzüglich 12 Zeichen Folgenummern und Änderungsdatum) für die Quelle zur Verfügung habe. Nun erstellen Sie eine Teildatei:



Teildatei erstellen

Als Teildateiart verwende ich SQL, damit der Editor die Syntax richtig darstellen kann. Sobald ich auf „Fertig stellen“ klicke, öffnet sich der LPEX-Editor:

Hier haben Sie jetzt – genau wie in der Prozedurumgebung – die Möglichkeit, Ihren Quellcode zu schreiben. Leider gibt es keine Bedienerführung, die Ihnen dabei behilflich ist. Aber mit RDI 9.5.1 können Sie die SQL-Skriptumgebung von Access Client Solution (ACS) öffnen und dort die verschiedenen SQL-Statements testen. Dazu öffnen Sie QUELLE ➜ „Ausführung von SQL Skripts“ starten.

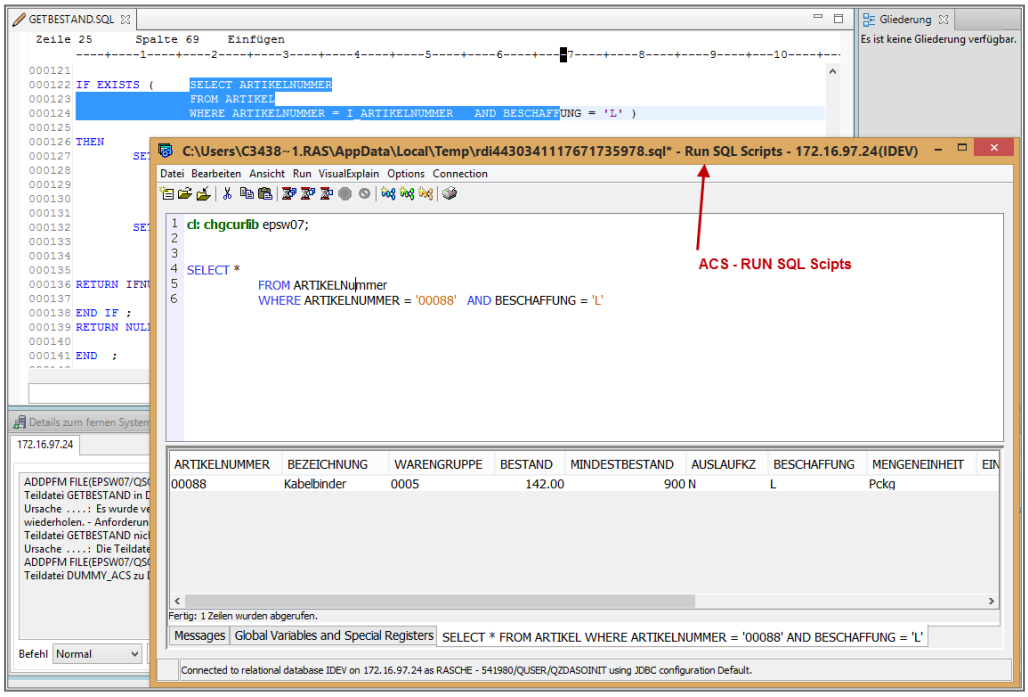


ACS RUN SQL Scripts aufrufen

Etwas befremdlich ist derzeit, dass Sie die Skriptumgebung nur öffnen können, wenn Sie eine Teildatei der Art SQLRPGLE geöffnet haben. Ich habe mir zu diesem Zweck eine Dummy-Teildatei angelegt, die ich einmalig für diese Verbindung öffne. Ist das Skript-Fenster geöffnet, können Sie die Teildatei einfach wieder schließen.

7.4.7
Seite 4

Anschließend steht Ihnen die Skriptumgebung zur Verfügung. Auch die SQL-Statements können Sie bei SQL-Prozeduren, Funktionen etc. leider nur per Drag&Drop in die Skriptumgebung übernehmen.

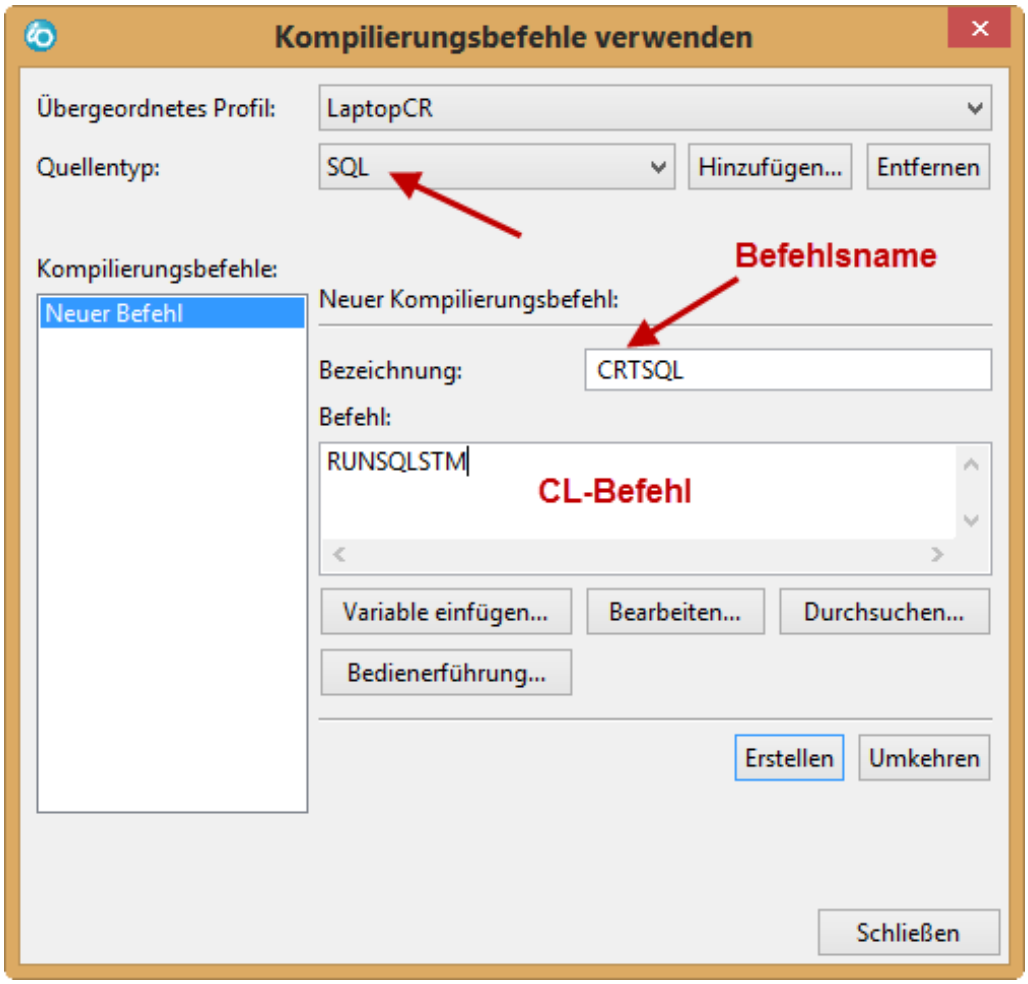


ACS-Run SQL Scripts

Keine schöne Lösung, aber wir brauchen ja auch noch ein paar Dinge, auf die wir uns freuen können!

Als Nächstes müssen wir uns noch einen Kompilierbefehl definieren, damit wir unsere Funktion auch erstellen können. Dazu öffnen wir: KOMPILIEREN ➜ KOMPILIEREN ➜ KOMPILIERUNGSBEFEHLE VERWENDEN.

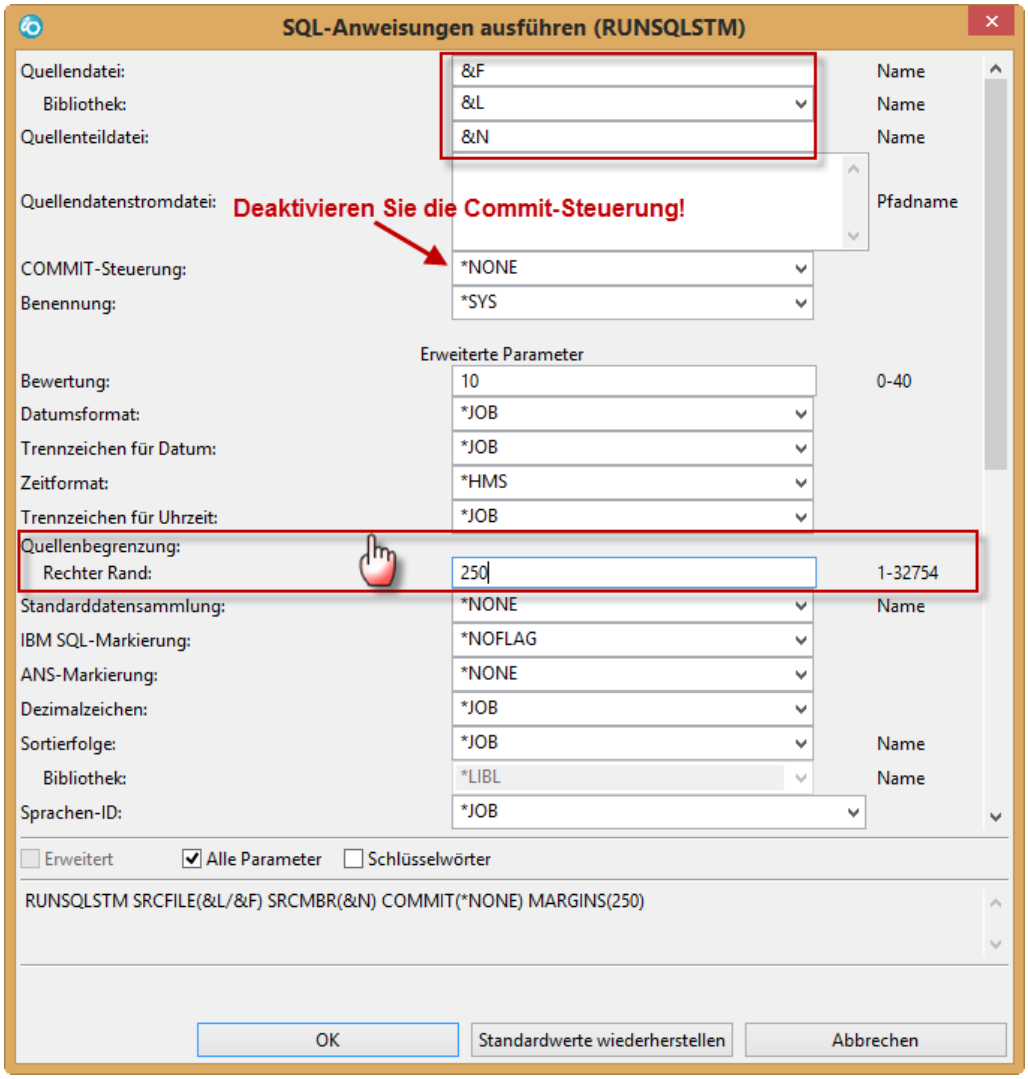
Anschließend sollten Sie den folgenden Dialog sehen:



Kompilierbefehl erstellen

Ich nutze für meine SQL-Quellen den Teildateityp SQL. Wenn Sie dem folgen, müssen Sie den Quellentyp SQL einmalig hinzufügen. Nun vergeben Sie noch einen Namen für Ihren Befehl – im Beispiel habe ich mich für CRTSQL entschieden. Als CL-Befehl verwenden wir RUNSQLSTM. Für den CL-Befehl können Sie die Bedienerführung anfordern, um die weiteren Parameter zu setzen:

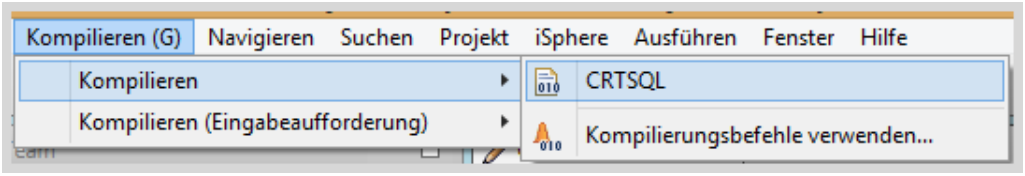
7.4.7
Seite 6



Der Befehl RUNSQLSTM

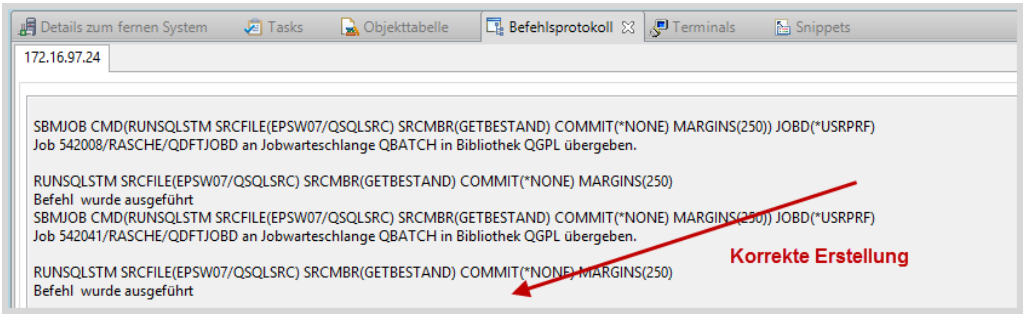
Der Name der Quellendatei, der Bibliotheks- und der Teildateiname werden mit entsprechenden Platzhaltern versehen. Schließlich soll unser Kompilierbefehl ja beliebige Teildateien umwandeln. Beachten müssen Sie lediglich noch die Satzlänge der Quellendatei. Hier handelt es sich um die Länge der Zeile ohne Folgenummern und ohne Änderungsdatum. Und auch die Commit-Steuerung sollten Sie deaktivieren, sonst vergessen Sie am Abend, Ihre Arbeit mit Commit zu beenden und wundern sich am nächsten Morgen, warum die am Vortag erstellten SQL-Objekte nicht mehr vorhanden sind!

Sind alle Eingaben gemacht, bestätigen Sie die Erstellung. Anschließend können Sie Ihre SQL-Quellen im RDI direkt umwandeln.

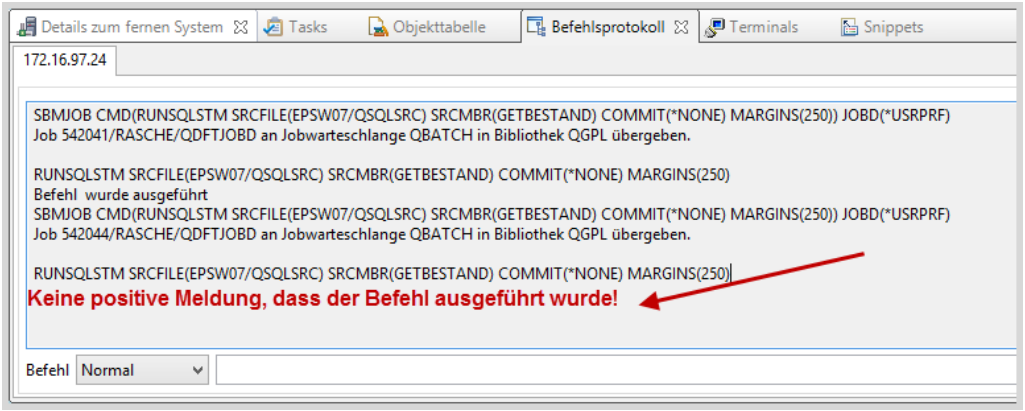


Kompilierbefehl nutzen

Leider erhalten Sie auch keine unmittelbaren Ergebnisse über die erfolgreiche Umwandlung. Sie müssen sich dazu das Befehlsprotokoll ansehen:



Das Befehlsprotokoll – alles okay



Das Befehlsprotokoll – Fehler aufgetreten

7.4.7
Seite 8

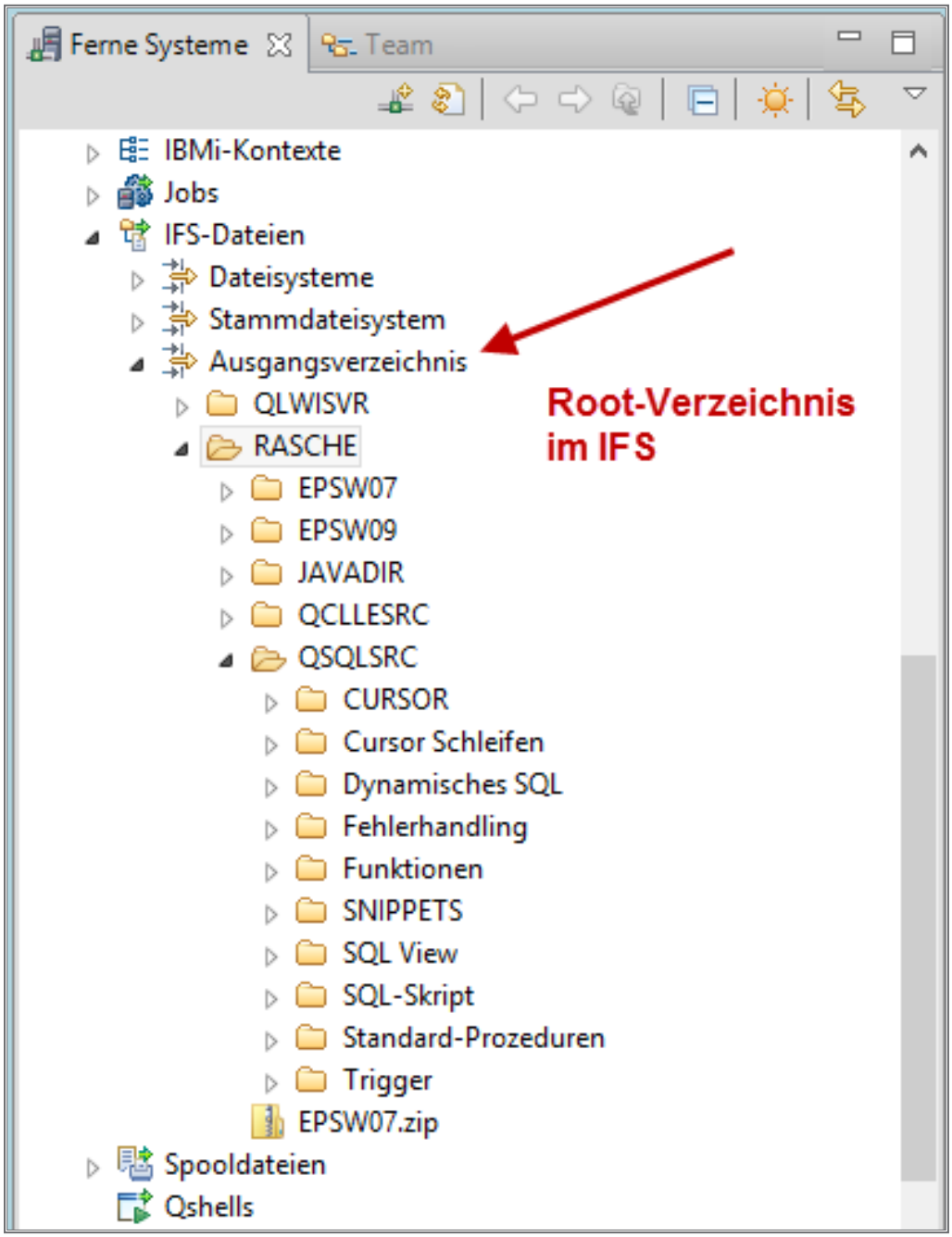
Fehlt die Meldung: „Befehl wurde ausgeführt“, sind Fehler aufgetreten und Sie müssen das Spoolfile öffnen, um die Fehler zu identifizieren.

```
  44

                    ***** E N D E  D E R  Q U E L L E *****
5770SS1 V7R2M0 140418         SQL-Anweisungen ausführen      GETBESTAND                 28.12.16 13:23:51  PAGE    3

Satz   *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8  FLGNR. Letzte Änderung

MSG ID  WTK  SATZ  TEXT
CPC2198  0      1  Position 1 Aktuelle Bibliothek in EPSW07 geändert.

SQL0199  30     29  Position 19 Schlüsselwort FROM nicht erwartet. Gültige
                    Token: ) ,.


                 Nachrichtenzusammenfassung
 Gesamt   Info  Warnung     Fehler  Wertigk. Beendigung
    2      1       0          0       1          0


Fehler der Wertigkeitsstufe 30 in Quelle gefunden.

                    ***** E N D E  D E R  L I S T E *****
```

Die Spooldatei

7.4.7.2 Benutzeraktionen für IFS-Teildateien erstellen

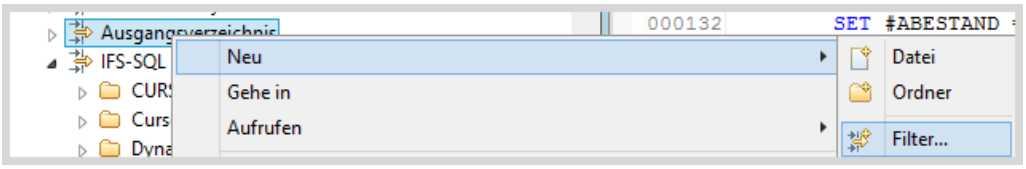
RDI zeigt Ihnen das Root-Verzeichnis Ihres Servers im Menü „Ausgangsverzeichnis“. Sie sehen in der folgenden Abbildung, dass ich in meinem Root-Verzeichnis „RASCHE“ das Verzeichnis „QSQLSRC“ mit diversen Unterverzeichnissen erstellt habe. Im IFS kann ich problemlos meine Teildateien nach verschiedenen Kriterien ordnen. Ein entscheidender Vorteil, wie ich finde!



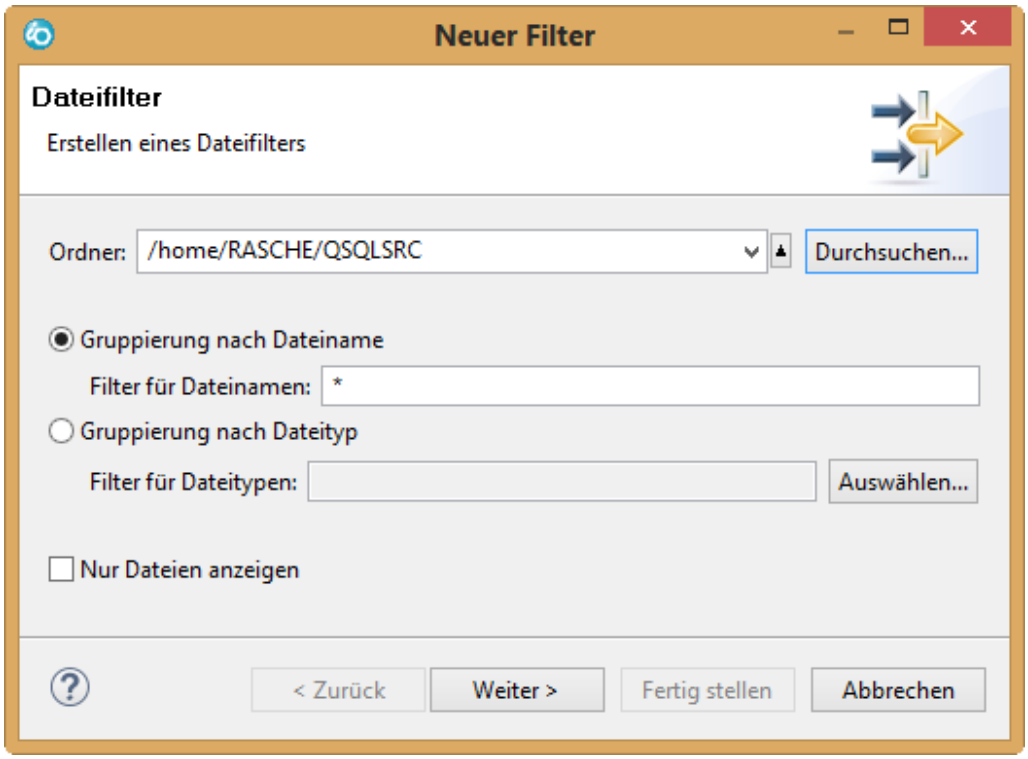
RDI – IFS Root-Verzeichnis

7.4.7
Seite 10

Falls Ihr Root-Verzeichnis nicht so übersichtlich ist wie meines, können Sie natürlich auch einen Filter für das relevante Verzeichnis erstellen:



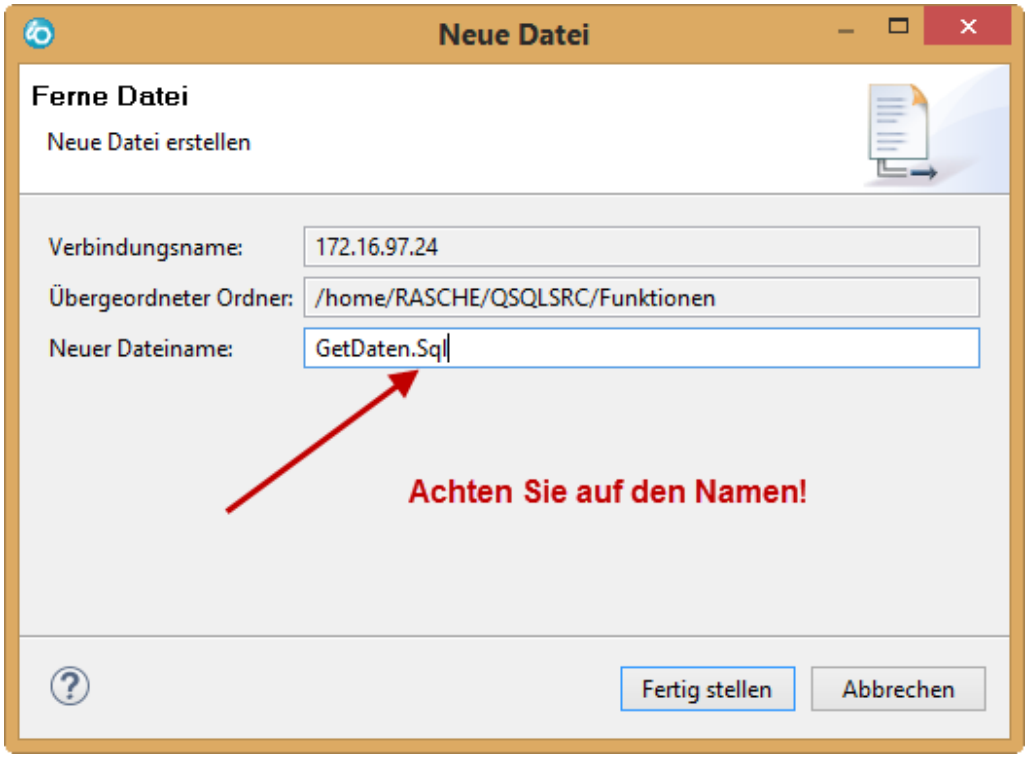
Filter im Root-Verzeichnis erstellen



Der IFS-Filter

Abschließend geben Sie dem Filter einen sprechenden Namen, so dass Sie dann problemlos – ohne langes Suchen – auf die IFS-Quellen zugreifen können.

Wenn Sie jetzt eine neue Datei im entsprechenden Verzeichnis erstellen, müssen Sie darauf achten, dass die Dateiextension Bestandteil des Namens ist.

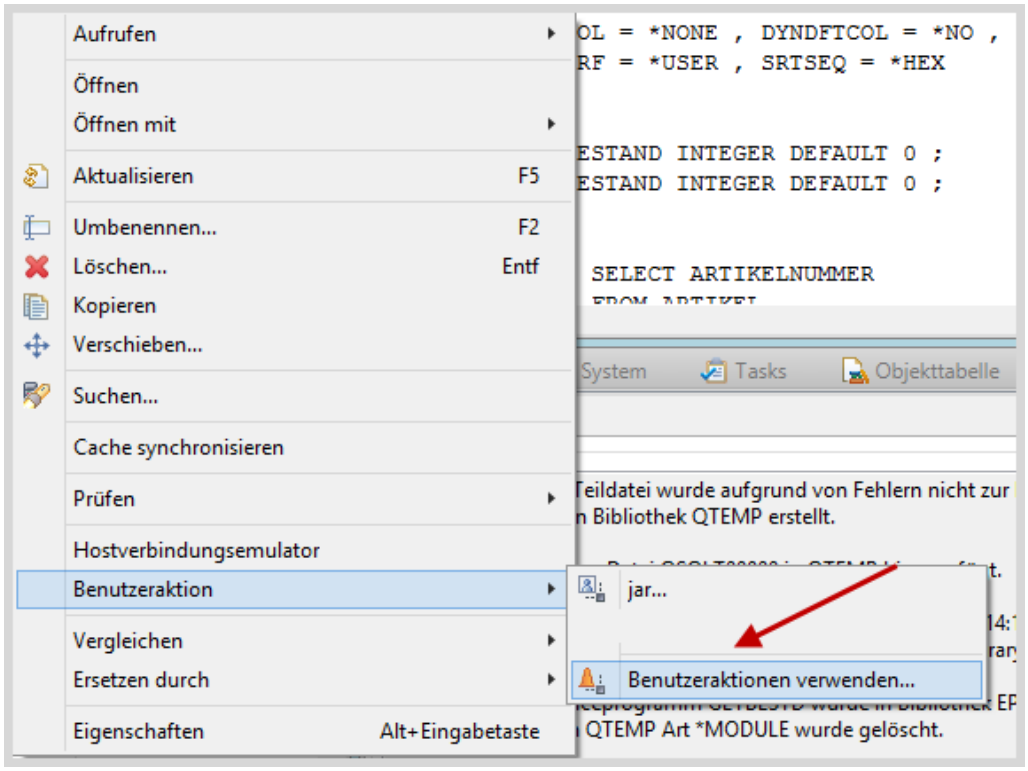


Neue Datei im IFS erstellen

Schade ist jetzt, dass der LPEX-Editor keine farbliche Darstellung der SQL-Quelle unterstützt, wenn es sich beispielsweise um eine ferne SQL-Quelle handelt. Warum eigentlich nicht? Diese Frage sollte man bei Gelegenheit einmal der IBM stellen.

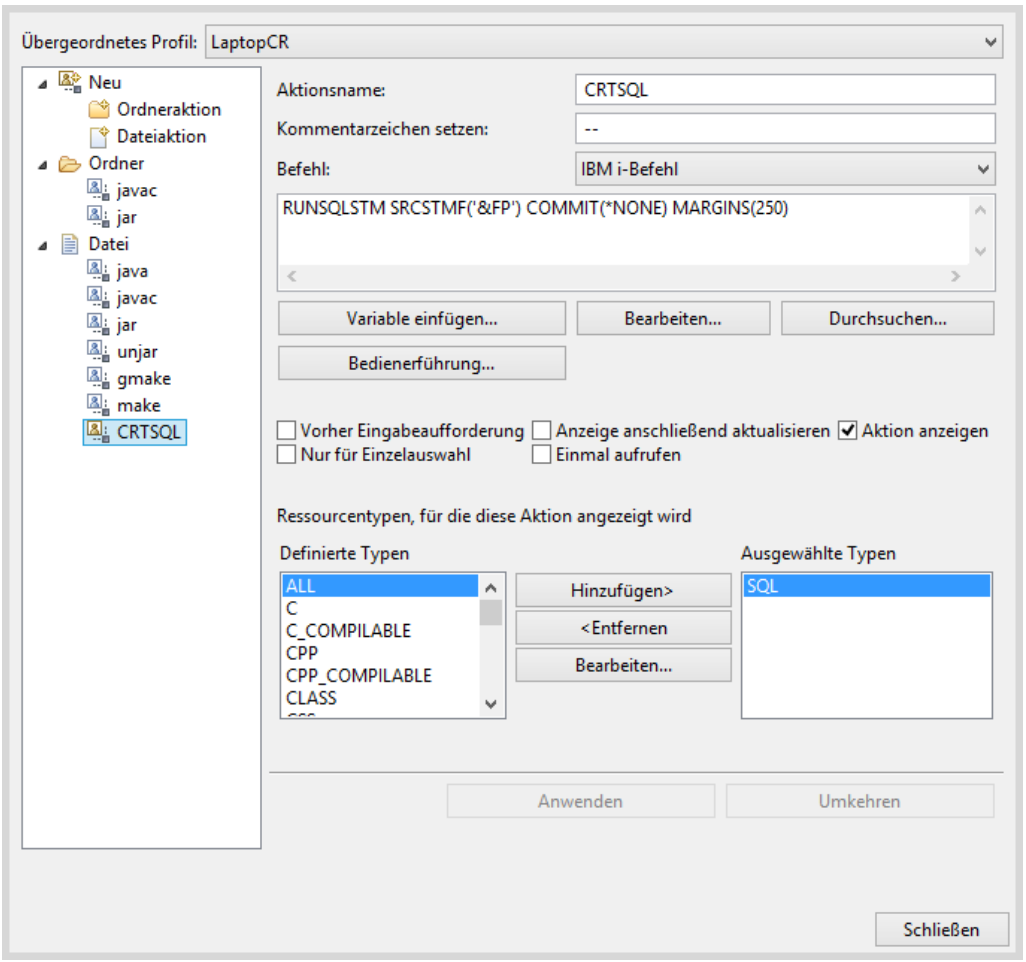
7.4.7
Seite 12

Um die ferne Datei auszuführen, benötigen wir dieses Mal eine „Benutzeraktion“.



Benutzeraktion definieren – Teil 1

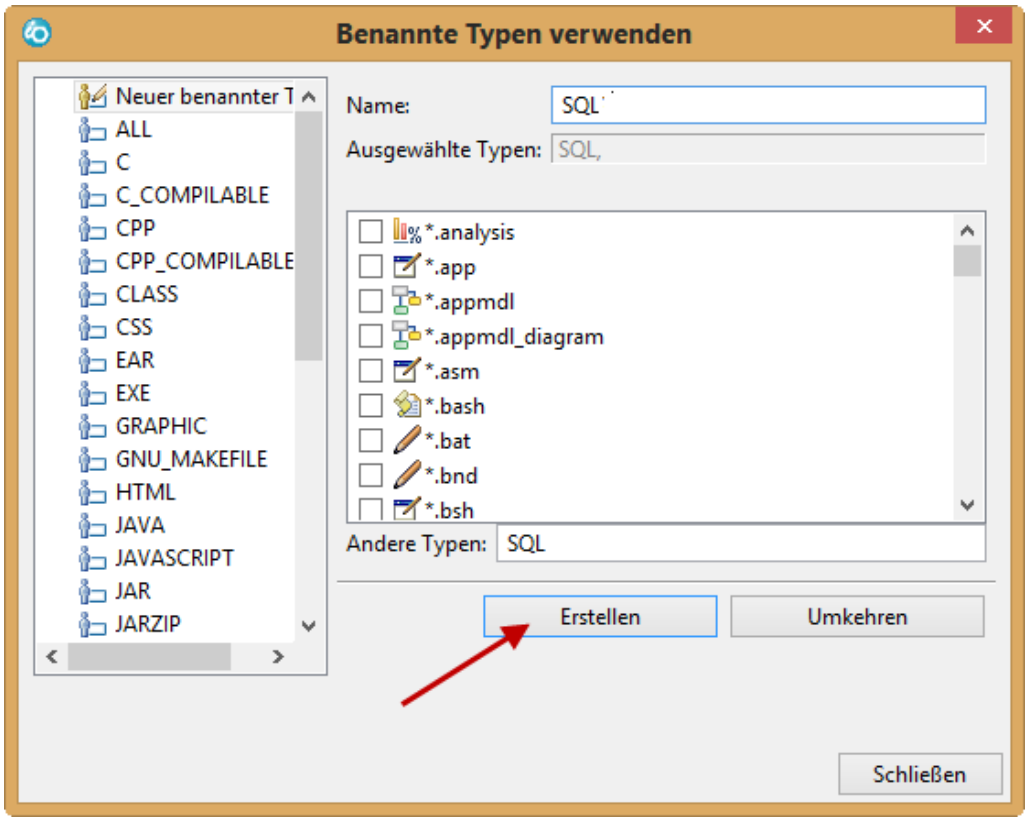
Anschließend sehen Sie den folgenden Dialog:



Benutzeraktion definieren – Teil 2

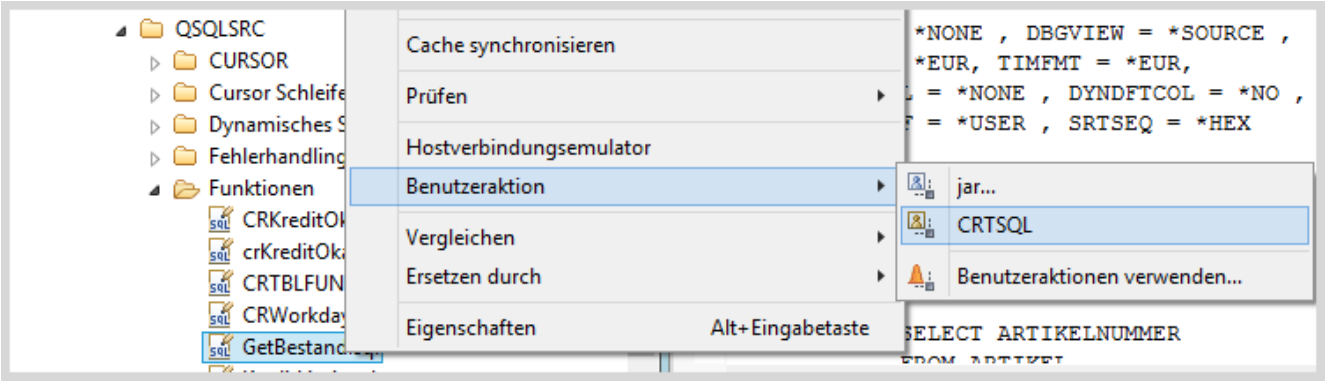
7.4.7
Seite 14

Sie vergeben für die Aktion wieder einen Namen und nutzen im CL-Befehl RUNSQLSTM den Parameter SRCSTMF. Achten Sie auf den Platzhalter &FP! Er muss zwingend in einfache Anführungszeichen gesetzt werden, damit der IFS-Pfad korrekt aufgelöst werden kann. Falls Sie die Aktion auf SQL-Ressourcen beschränken wollen – wie ich es im Beispiel getan habe – müssen Sie noch den SQL-Typ definieren:



Benannte Typen erstellen

Anschließend können Sie die selbsterstellte Benutzeraktion für die Erstellung Ihrer SQL-Objekte nutzen.



Benutzeraktion verwenden

7.5 Interfaces für die SQL-Ausführung

Die SQL-Syntax ist eine Sache, ihre anschließende Ausführung eine andere. Es gibt viele verschiedene Möglichkeiten, SQL-Statements auszuführen. In diesem Kapitel wollen wir uns die Möglichkeiten anschauen, die IBM für die Ausführung unserer SQL-Statements bereitstellt.

1. Sie können Ihre SQL-Statements in Ihre RPG-Programme integrieren. Das ist sicherlich die aufwendigste Methode, so dass wir dieser Technik ein eigenständiges Kapitel gewidmet haben.
2. Sie führen die SQL-Abfragen direkt per Betriebssystembefehl aus, wie es in Kapitel 7.5.3 gezeigt wird.
3. Es lassen sich aber auch SQL-Skripte aufbauen, die anschließend mit wenig Aufwand ausgeführt werden können. Wie das geht, zeigt Kapitel 7.5.1.
4. Eine weitere Möglichkeit ist der Query Manager. Der Query Manager ist ein SQL-basiertes Werkzeug, das Sie mittlerweile auf jedem IBM i-Server nutzen können – ohne zusätzliche Lizenzkosten. Detaillierte Informationen hierzu enthält Kapitel 7.5.4.

Ich weiß, dass jeder Leser auf seine bevorzugte Umgebung schwört. Letztendlich aber hat jede SQL-Umgebung ihre spezifischen Vor- und Nachteile, und nicht selten ist es kontextabhängig, welches Tool oder welche Oberflächen Sie einsetzen sollten, aber auch die persönlichen Vorlieben spielen hier eine entscheidende Rolle. Ich werde Ihnen im weiteren Verlauf dieses Kapitels verschiedene Umgebungen und Oberflächen vorstellen; Sie entscheiden dann, welche Oberfläche Sie nutzen wollen!

Ergänzung 28./2014

7.5 Seite 2

7.5.1 Der Befehl RUNSQLSTM

Es gibt schon seit langem diverse Möglichkeiten, SQL-Anweisungen über CL-Befehle auszuführen. Einer der am häufigsten verwendeten Befehle ist sicherlich der Betriebssystembefehl RUNSQLSTM (Run SQL Statement). Er ermöglicht die Ausführung von SQL-Skripten, die in Teildateien einer Quellendatei gespeichert werden.

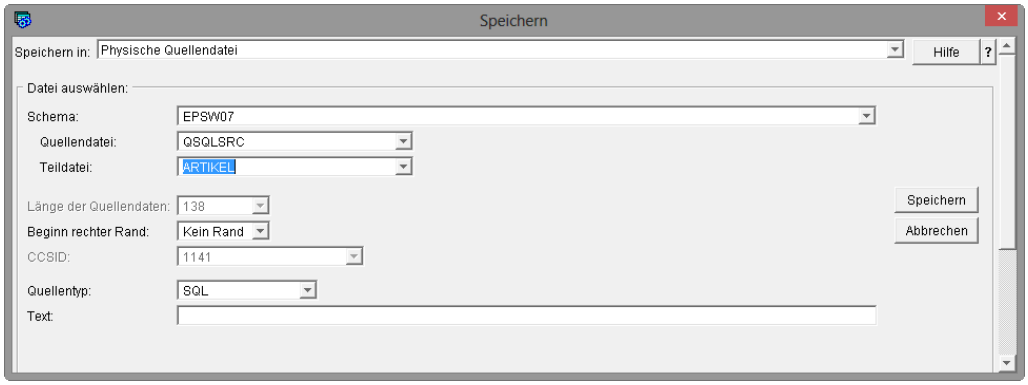
Stellen Sie sich vor, Sie hätten in der Prozedurumgebung des System i Navigators ein SQL-Skript mit folgenden DDL-Anweisungen erstellt:

7.5.1
Seite 2

```
/* Artikelstammdaten erstellen */

CL: CHGCURLIB EPSW07;

DROP TABLE ARTIKEL2 ;

CREATE TABLE ARTIKELSTAMM2 for System Name Artikel2 (
       FIRMA CHAR(3) NOT NULL DEFAULT ‚001‘ ,
       ARTIKELNUMMER FOR COLUMN ARTNR CHAR(5) NOT NULL
                                                DEFAULT ‚‘ ,
       BEZEICHNUNG FOR COLUMN ARTBEZ CHAR(20) NOT NULL
                                                DEFAULT ‚‘ ,
       WARENGRUPPE FOR COLUMN WG CHAR(4) DEFAULT NULL,
       MINDESTBESTAND FOR COLUMN MINBST DECIMAL(5, 0)
                                          NOT NULL DEFAULT 0,
       AUSLAUFKZ CHAR(1) NOT NULL DEFAULT ‘N’ ,
       BESCHAFFUNG FOR COLUMN BESCHAFF CHAR(1) NOT NULL
                                              DEFAULT ‘K’ ,
       MENGENEINHEIT FOR COLUMN MHE CHAR(5) DEFAULT NULL ,
       ARTIKELPREIS FOR COLUMN ARTPRS DECIMAL(5, 2) NOT NULL
                                                DEFAULT 0,
       ANLAGEDATUM FOR COLUMN ERFDAT DECIMAL(8, 0) NOT NULL
                                                DEFAULT 0,
       CONSTRAINT
          ARTIKEL2_PKEY PRIMARY KEY( FIRMA , ARTIKELNUMMER )
       )
       RCDFMT ARTFMT ;

LABEL ON COLUMN ARTIKEL
( ARTIKELPREIS IS ‘VK-Preis ‘ ,
 ANLAGEDATUM IS ‘Artikel erfasst am ‘ , …
         ) ;
CL: GRTOBJAUT OBJ(ARTIKEL2) OBJTYPE(*FILE) USER(*PUBLIC)
                                          AUT(*EXCLUDE) ;
```

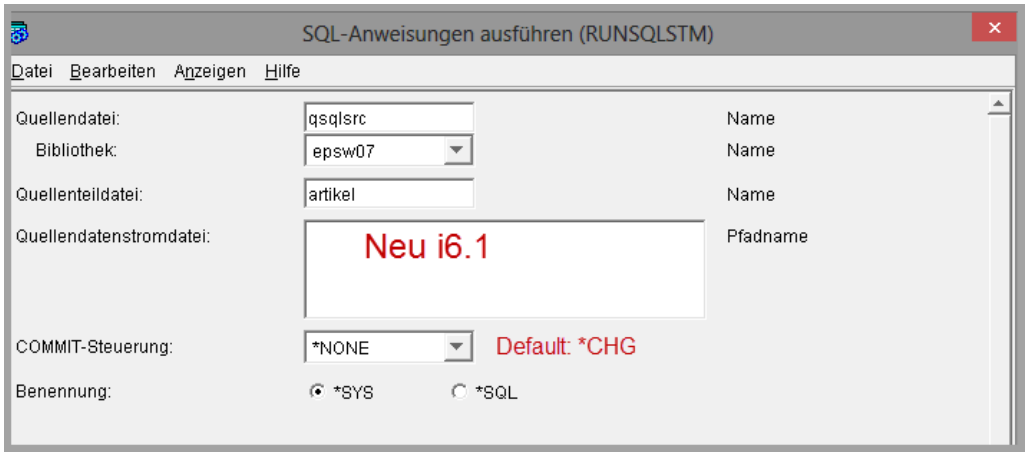
Das oben dargestellte SQL-Skript enthält sowohl SQL-Anweisungen als auch CL-Befehle.

In der Prozedurumgebung des System-i-Navigators können bereits seit längerem CL-Befehle ausgeführt werden, wenn sie durch die Zeichenfolge "CL:" gekennzeichnet und ebenso wie die SQL-Befehle mit einem Semikolon beendet werden. Seit Release V6R1 können wir diese SQL-Skripte auch mit dem Befehl RUNSQLSTM ausführen. Arbeiten Sie derzeit noch mit dem Release V5R4, müssen Sie allerdings darauf achten, dass die auszuführenden SQL-Skripte lediglich SQL-Anweisungen enthalten. Das oben gezeigte Skript habe ich zwischenzeitlich in eine Teildatei gespeichert:



SQL-Skripte in Teildateien speichern

Jetzt soll diese Teildatei auf einer 5250-Befehlszeile oder aus einem CL-Programm heraus ausgeführt werden. Dazu verwenden wir zunächst den Befehl RUNSQLSTM:



Der CL-Befehl RUNSQLSTM

Die ersten drei Parameter des Befehls sind keine Überraschung: Sie benennen die Quellendatei und die auszuführende Teildatei. Neu in Release V6R1 ist der Parameter Quellendatenstromdatei (SRCSTMF). Sie können jetzt auch im IFS gespeicherte SQL-Skripte mit dem CL-Befehl RUNSQLSTM ausführen.

Beachten Sie bei der Befehlsausführung die Namenskonvention, also den Parameter „Benennung“. In meinem SQL-Skript verwende ich die Namenskonvention *SYS, weil ich die Bibliothekssuchliste und Slash (/) für die Objektqualifizierung verwenden möchte. Auch den Parameter „Commit-Steuerung“ sollten Sie beachten. Voreingestellt ist der Wert *CHG. Alle DDL-Anweisungen – also CREATE TABLE, CREATE VIEW – aber auch einige DML-Anweisungen wie INSERT, UPDATE und DELETE müssen mit der Anweisung COMMIT oder ROLLBACK explizit bestätigt werden. Geschieht dies nicht, werden alle Änderungen und Neuerstellungen nach Beendigung des Jobs automatisch verworfen.

Für jede Ausführung – egal ob erfolgreich oder nicht – erhalten Sie ein Umwandlungsprotokoll, das Sie sich anschauen sollten.

```
MSG ID  WTK  SATZ  TEXT
CPC2198   0       1  Position 1 Aktuelle Bibliothek in EPSW07 geändert.
SQL7953   0       5  Position 1 Löschen von ARTIKEL2 in EPSW07 beendet.
SQL7950   0       7  Position 1 Tabelle ARTIKELSTAMM2 in EPSW07 erstellt.
SQL7965   0      21  Position 1 LABEL ON für ARTIKEL in EPSW07 beendet.
CPC2201   0      25  Position 1 Objektberechtigung erteilt.
                     Nachrichtenzusammenfassung
  Gesamt    Info   Warnung      Fehler  Wertigk. Beendigung
     5        5         0           0        0         0
Fehler der Wertigkeitsstufe 00 in Quelle gefunden.
```

Das Ausführungsprotokoll

Etwas ist fremd für uns: Sie sehen im folgenden Protokoll, dass der SQL-Status SQL7905 eine Fehlermeldung der Wertigkeit 20 auslöst.

```
MSG ID  WTK  SATZ  TEXT
CPC2198   0       1  Position 1 Aktuelle Bibliothek in TEST geändert.
SQL7953   0       5  Position 1 Löschen von ARTIKEL2 in TEST beendet.
SQL7905  20       7  Position 1 Tabelle ARTIKELSTAMM2 in TEST erstellt, aber
                     nicht im Journal aufgezeichnet.
SQL7965   0      21  Position 1 LABEL ON für ARTIKEL2 in TEST beendet.
CPC2201   0      25  Position 1 Objektberechtigung erteilt.
                     Nachrichtenzusammenfassung
  Gesamt    Info   Warnung      Fehler  Wertigk. Beendigung
     6        4         0           1        1         0
```

Fehlermeldungen im Ausführungsprotokoll – kein Abbruch

Normalerweise hätte die Ausführung abgebrochen werden müssen, da der Generierungslevel des RUNSQLSTM-Befehls auf 10 steht. Das Skript wurde aber erfolgreich ausgeführt, da die Nichtjournalisierung der Tabellen keinen Fehler im eigentlichen Sinn darstellt.

Anders verhält es sich allerdings, wenn Sie versuchen, ein nicht vorhandenes Objekt zu löschen.

```
MSG ID  WTK  SATZ  TEXT
CPC2198   0     1  Position 1 Aktuelle Bibliothek in TEST geändert.
SQL0204  20     5  Position 1 ARTIKEL2 der Art *FILE in *LIBL nicht
                   gefunden.
                 Nachrichtenzusammenfassung
  Gesamt    Info   Warnung      Fehler  Wertigk. Beendigung
     2        1         0          1        0         0
Fehler der Wertigkeitsstufe 20 in Quelle gefunden.
```

Fehlermeldungen im Ausführungsprotokoll – mit Abbruch

Auch jetzt erhalten wir wieder einen Fehler der Wertigkeitsstufe 20. Doch dieses Mal wird die Ausführung des SQL-Skripts abgebrochen, obwohl man auch hier nicht von einem Fehler im eigentlichen Sinn nicht sprechen kann. Wie reagieren wir nun auf die Fehlermeldungen?

Derzeit können Sie nur die Standardeinstellung des Parameters „Bewertung (ERRLVL)“ fallweise auf den Generierungslevel 21 ändern. Bleibt abzuwarten, wann IBM reagieren wird. Im System-i-Navigator können wir diese Fehlersituation bereits durch entsprechende Einstellungen der Optionen unterbinden. Vielleicht gibt es ja auch für den CL-Befehl RUNSQLSTM bald einen entsprechenden Parameter.

Auch wenn dies kein Referenzhandbuch werden soll, möchte ich Ihnen noch einige Parameter zwanglos vorstellen, von denen ich glaube, dass sie an der einen oder anderen Stelle ganz nützlich sein könnten.

Neu V6R1-Quellenbegrenzung (MARGINS)
Der Parameter bestimmt den rechten Rand der Teildatei. Bislang durften die Zeilen eines SQL-Skripts maximal 80 Zeichen enthalten – unabhängig von der vereinbarten Zeilenlänge der Quellendatei. Dies konnte dazu führen, dass SQL-Anweisungen unvollständig blieben, weil die Anweisungen einfach abgeschnitten wurden. Seit Release V6R1 ist der Default-Wert zwar immer noch auf 80 Zeichen begrenzt, er kann jedoch auf jede beliebige Breite geändert werden.

Anweisungsverarbeitung (Process)
Manchmal könnte es erforderlich sein, zunächst die SQL-Syntax lediglich zu überprüfen, bevor das SQL-Skript ausgeführt wird. Für die Syntaxprüfung setzen Sie den Parameter „Process“ auf den Wert *SYN und unterbinden dadurch eine ungewollte Ausführung.

Weiter Beispiele für die Verwendung des Befehls RUNSQLSTM finden Sie in verschiedenen Kapiteln dieses Buchs.

7.5.1 Seite 6

7.5.2 Der Befehl RUNSQL

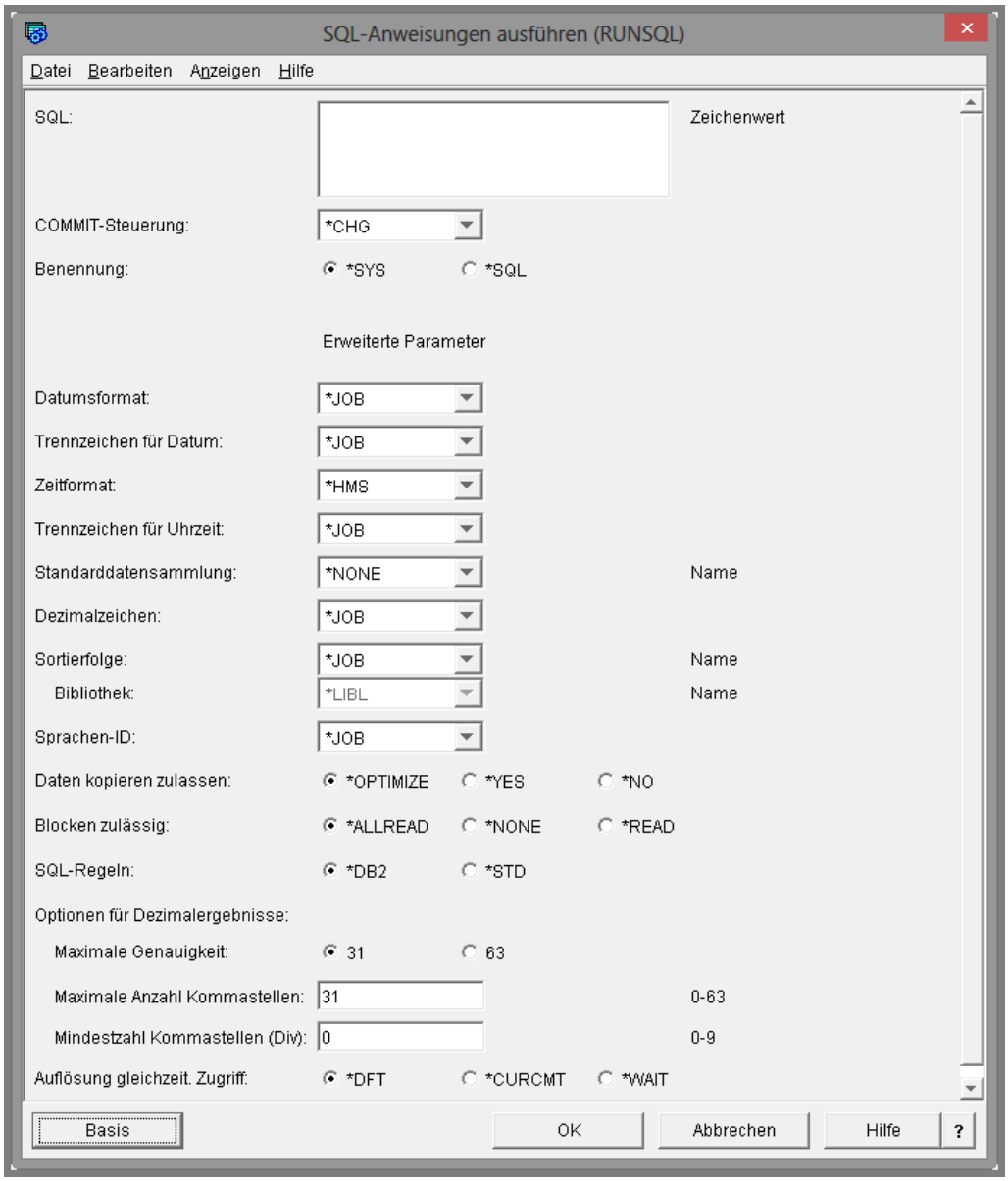
CL-Programme übernehmen heute vielfältige Aufgaben und manchmal würden wir dabei gerne Fehler oder andere Informationen in Datenbankdateien protokollieren. Doch obwohl IBM CL in letzter Zeit stark aufgewertet hat und viele neue Befehle integriert wurden, gibt es bislang keinen Schreibbefehl für Datenbankdateien. Können wir also tatsächlich keine Informationen aus einem CL-Programm heraus in Datenbankdateien ablegen? Ein Irrtum!

Ich schreibe schon seit vielen Jahren CL-Programme, die physische Datenbanklogdateien nutzen. Nur benutze ich dabei innerhalb des CL-Programms halt SQL, um diese Aufgabe zu lösen. Einziger Wehmutstropfen: Bislang musste der Umweg über den IBM DB2 Query Manager gegangen werden. Und dieser Weg war manchmal etwas umständlich.

Doch diese Zeiten sind vorbei: IBM hat den Betriebssystembefehl RUNSQL in Release 6.1 und 7.1 integriert. Dieser Befehl ermöglicht es, SQL-Statements direkt aus einem CL-Programm heraus auszuführen oder – falls erforderlich – natürlich auch über eine 5250-Befehlszeile zu nutzen. Voraussetzung für die Verfügbarkeit des Befehls ist das Datenbankgruppen-PTF SF99601 Level 25 für Release 6.1 und für Release 7.1 das DB2-Gruppen-PTF SF99701 Level 14.

Der Befehl RUNSQL hat verschiedene Parameter, um die verwendete Namenskonvention (NAMING), die Transaktionsstufe (COMMIT) oder das Datumsformat (DATFMT) festzulegen. Der wichtigste Parameter ist jedoch der Parameter SQL. Ein 5.000 Byte großes Zeichenfeld, das ein beliebiges SQL-Statement enthalten kann. Wir könnten also neue Tabellen erstellen (CREATE TABLE), logische Sichten aufbauen (CREATE VIEW) oder bestehende Tabellen manipulieren (INSERT, UPDATE, DELETE). Schauen wir uns den Befehl und seine Parameter genauer an:

7.5.2
Seite 2



Der Befehl RUNSQL

Der wichtigste Parameter ist sicherlich der **Parameter „SQL“ (SQL):** Ein 5.000 Byte großes Zeichenfeld, das ein beliebiges SQL-Statement enthalten kann. Wir könnten also neue Tabellen erstellen (CREATE TABLE), logische Sichten aufbauen (CREATE VIEW) oder bestehende Tabellen manipulieren (INSERT, UPDATE, DELETE) und natürlich könnten auch SQL-Prozeduren ausgeführt werden.

Der Parameter **„COMMIT-Steuerung“** (**COMMIT**) bestimmt die Transaktionsstufe. Sie sehen in der Abbildung oben, dass die Standardeinstellung *CHG lautet. Sie müssen daher diesen Parameter unbedingt ändern, wenn Sie in Ihren Programmen nicht mit COMMIT- und ROLLBACK-Operationen arbeiten wollen. Andernfalls würden alle Operationen rückgängig gemacht werden, wenn Sie Ihr CL-Programm beenden. Nutzen Sie die Einstellung *NONE, wenn keine Transaktionen gewünscht sind.

Bekannt dürfte auch der Parameter **„Benennung“** (**NAMING**) sein, wenn Sie bereits mit SQL im IBM i-Umfeld arbeiten. Mit Hilfe dieses Parameters bestimmen Sie, welche Namenskonvention Sie im SQL-Statement für die Adressierung Ihrer Objekte verwenden. Genauer gesagt, ob Sie Objekte mit einem Slash (/) und der Bibliothekssuchliste oder mit einem Punkt (.) und dem SQL-Standardschema adressieren wollen. Nutzen Sie am Besten immer die Namenskonvention *SYS, denn seit einiger Zeit können Sie in der Namenskonvention *SYS sowohl den Slash als auch den Punkt zur Objektqualifizierung verwenden.

Die weiteren Parameter **„Datumsformat“** (**DATFMT**), **„Trennzeichen für Datum“** (**DATSEP**), **„Zeitformat“** (**TIMFMT**) und **„Trennzeichen für Datum“** (**TIMSEP**) sind meiner Meinung nach selbsterklärend. Sie bestimmen das Format für Datum und Uhrzeit. Das Gleiche gilt auch für den Parameter **„Trennzeichen für Uhrzeit“** (**TIMSEP**).

Die **„Standarddatensammlung“** (**DFTRDBCOL**) spezifiziert das Standardschema für unqualifiziert adressierte Tabellen, Views, Indizes und SQL-Packages. Diesen Parameter benötigen Sie allerdings nur, wenn Sie mit der Namenskonvention *SQL arbeiten, ansonsten verwenden Sie – wie gewohnt – die Bibliotheksliste zur Qualifizierung Ihrer Objekte.

„Dezimaltrennzeichen“ (**DECMPT**), **„Sortierfolgen“** (**SRTSEQ**) **und „Sprachen-ID“** (**LANGID**) kennen Sie von vielen anderen i5-Befehlen.

Der Parameter **„Daten kopieren zulassen“** (**ALWCPYDTA**) ist Ihnen vielleicht in anderen Interfaces schon begegnet. Letztendlich bestimmt der Parameter, ob SQL Ihre Ergebnismenge in Form einer Datenkopie zur Verfügung stellen darf oder ob Sie stets mit „Live“-Daten arbeiten möchten. Der Eintrag *OPTIMZE überlässt die Entscheidung dem SQL-Optimizer, der abhängig von Syntax, Interface und Datenzugriff fallweise entscheidet.

Spannend könnte auch der Parameter **„SQL-Regeln“** sein – zumindest für all diejenigen, die den SQL Ansi-Standard einhalten wollen oder müssen. Der Standardwert *DB2 erlaubt alle i5-spezifischen SQL-Syntaxen, die nicht unbedingt dem Standard entsprechen müssen. Verwenden Sie hingegen den Wert *STD, muss der SQL-Standard eingehalten werden.

Ergänzung 28./2014

7.5.2
Seite 4

Der Parameter **„Auflösung gleichzeit. Zugriff“** setzt eine relativ neue SQL-Eigenschaft. *WAIT ist die Standardeinstellung. Bislang waren wir es gewohnt, im Programm auf die Freigabe einzelner Sätze zu warten, wenn diese durch andere Jobs gesperrt sind. Dieses Verhalten entspricht dem Wert *WAIT. Abweichend hiervon können Sie nun auch den Wert *CURCMT verwenden. In diesem Fall werden Sie nur die festgeschriebenen beziehungsweise freigegebenen Zeilen einer Tabelle erhalten; alle gesperrten Zeilen werden ignoriert und im Result Set nicht dargestellt. Soweit die Theorie! Beispiele für die Verwendung des Befehls RUNSQL finden Sie in verschiedenen Kapiteln dieses Buchs. Schauen Sie sich dazu beispielsweise Kapitel 7.7.1 an. In einer externen Prozedur wird der Befehl RUNSQL im CL-Programm verwendet, um unberechtigte Objektzugriffe zu protokollieren.

7.5.5 Der Query Manager

Der Query Manager ist ein SQL-basiertes Werkzeug, das Sie auf jedem IBM i-Server nutzen können. Allerdings müssen Sie darauf achten, dass das Feature 5770ST1 installiert wurde.

Und dennoch findet dieses SQL-Interface oftmals wenig Beachtung – oder ist völlig unbekannt. Nicht selten wird der Query Manager auch einfach mit Query/400 verwechselt.

Der Query Manager (QM) verfügt aber im Vergleich zu IBM Query/400 über viele erweiterte Funktionen. So können Daten nicht nur gelesen werden – auch das Einfügen, Ändern und Löschen von Datensätzen ist möglich. SQL-Abfragen können direkt ausgeführt werden, und dürfen Variablen enthalten, so dass Selektions- und Sortierkriterien erst zur Laufzeit bestimmt werden.

Neben all diesen Vorteilen darf aber nicht unerwähnt bleiben, dass die Query Manager-Abfragen eigentlich aus zwei Komponenten bestehen: der Abfrage (*QMQRY) und einem separat zu erstellendem Report (*QMFORM). In Kapitel 9.5.1 zeige ich Ihnen daher zunächst, welche Möglichkeiten Sie haben, eine QM-Abfrage zu erstellen. Und in Kapitel 9.5.2 kümmern wir uns dann um das Berichtslayout.

Und noch eine Funktion dürfen wir an dieser Stelle nicht vergessen. Wir können QUERY/400-Abfragen in QM-Abfragen konvertieren. Und sollte IBM irgendwann Query/400 nicht mehr unterstützen, dann ist der Query Manager eine geeignete Alternative.

Ergänzung 29./2014

7.5.5
Seite 2

7.5.5.1 Query Manager – Benutzerprofile

Wenn Sie den Query Manager starten (STRQM), gelangen Sie zum Startmenü:

```
                        DB2 für IBM i Query Manager
                                                        System:    ITRAINNG
Auswahlmöglichkeiten:

     1. Mit Query Manager Abfragen arbeiten
     2. Mit Query Manager Berichtsformularen arbeiten
     3. Mit Query Manager Tabellen arbeiten

    10. Mit Query Manager Profilen arbeiten

Auswahl
     _

F3=Verlassen    F12=Abbrechen    F22=QM-Anweisung
```

Das QM-Startmenü

Jeder Benutzer, der mit dem Query Manager arbeitet, hat sein eigenes Benutzerprofil. Schauen wir uns kurz die wichtigsten Profileinstellungen an. Dazu wählen Sie im Startmenü den Menüpunkt 10 „Mit Query Manager Profilen arbeiten“. Daraufhin erscheint zunächst eine Übersicht aller Benutzerprofile. Wählen Sie ein Profil mit der Option 2 aus, so gelangen Sie in den folgenden Dialog:

```
                        Query Manager Profil ändern

Benutzerprofil . . . . . . . . . . . . :   RASCHE
Beschreibung . . . . . . . . . . . . . :   Carmen Rasche


Auswahl eingeben und Eingabetaste drücken.
  Standardbibliothek für QM-Objekte  . .   EPSW07      Name, *CURLIB
  Standardberechtigung für
    Objekterstellung . . . . . . . . . .   *LIBCRTAUT  *CHANGE, *ALL, *USE
                                                      *EXCLUDE, *LIBCRTAUT
                                                      Berechtigungsliste
  Ausführungsmodus der Abfrage . . . . .   2           1=Interaktiv
                                                      2=Stapelbetrieb
  Ändern Ausführungsmodus zulässig . . .   N           J=Ja, N=Nein
  Benutzerzugriffsebene  . . . . . . . .   1           1=Alle, 2=Basis
  Anzahl der Zeilen für Beispiel . . . .   50          1-100
  Interaktive Ausführung des Beispiels
    zulässig . . . . . . . . . . . . . .   J           J=Ja, N=Nein

                                                                  Weitere ...
F3=Verlassen   F4=Bedienerführung   F5=Aktualisieren   F12=Abbrechen
F22=QM-Anweisung
Anfang des Bereichs bereits erreicht.
```

QM Profil ändern – Seite 1

Im QM-Profil wird die Standardbibliothek für QM-Objekte hinterlegt. Während Ihrer Arbeit mit QM-Abfragen können Sie diese Standardbilbliothek jederzeit temporär überschreiben.

Auch die Standardberechtigung für *PUBLIC zur Ausführung der QM-Objekte legen Sie hier grundsätzlich fest. Für meine verwendete Bibliothek EPSW07 hat *PUBLIC keine Rechte. Die QM-Objekte erben diese Bibliotheksrechte, wenn ich es beim Standardwert *LIBCRTAUT belasse. In meinem Fall ist dies so gewünscht. Laut der oberen Abbildung dürften QM-Abfragen nur im Batch ausgeführt werden; gegebenenfalls kann diese Einstellung gewünscht sein. Ich belasse es hier allerdings meistens bei dem Wert 1 „Interaktiv“ und entscheide später bei der Ausführung der Abfrage selbst, wie diese ausgeführt werden soll.

Auf der zweiten Seite des Profils können Sie interaktive Abfragen dann gegebenenfalls noch in der Ergebnismenge beschränken und legen – wie in allen SQL-Interfaces – die verwendete Namenskonvention fest.

Ergänzung 29./2014

7.5.5 Seite 4

Auf dieser Seite können Sie im Übrigen auch einen Remotezugriff auf eine ferne Datenbank konfigurieren.

```
                        Query Manager Profil ändern

Benutzerprofil . . . . . . . . . . . . :   RASCHE
Beschreibung . . . . . . . . . . . . . :   Carmen Rasche


Auswahl eingeben und Eingabetaste drücken.
  Maximal zulässige Anzahl Zeilen bei
    interaktiver Ausführung  . . . . . .   *NOMAX          1-*NOMAX
  Ausführungsauswahlmöglichk. anzeigen     J              J=Ja, N=Nein
  QM-Bestätigungsnachrichten . . . . . .   J              J=Ja, N=Nein
  Namenskonvention . . . . . . . . . . .   *SYS           *SYS, *SAA
  Verbindung zur relationalen Datenbank    *NONE
                                                          Name, *NONE
                                                          *CURRENT
                                                          F4=Liste
  RDB-Verbindungsmethode . . . . . . . .   *RUW           *RUW, *DUW
  Standardbibliothek für QM-Tabellen . .   *NONE         Name, *NONE, *CURLIB
  Ausgabe der Abfragedaten . . . . . . .   1              1=Anzeige, 2=Drucker
                                                          3=Datei
                                                                  Weitere ...
F3=Verlassen   F4=Bedienerführung   F5=Aktualisieren   F12=Abbrechen
F22=QM-Anweisung
```

QM Profil ändern – Seite 2

Die dritte Seite enthält Jobsteuerungsangaben und soll hier nicht weiter erläutert werden, da dies letztendlich bekannte Einstellungen für Systemadministratoren sind.

```
                        Query Manager Profil ändern

Benutzerprofil . . . . . . . . . . . . :   RASCHE
Beschreibung . . . . . . . . . . . . . :   Carmen Rasche


Auswahl eingeben und Eingabetaste drücken.
  Drucker für Ausgabe  . . . . . . . .   *JOB          Name, *JOB
                                                      F4=Liste
  Datei für Ausgabe  . . . . . . . . .   *NONE         Name, *NONE
                                                      F4=Liste
    Bibliothek . . . . . . . . . . . .     __________   Name, *LIBL, *CURLIB
  Jobbeschreibung bei Ausführung im
    Stapelbetrieb  . . . . . . . . . .   *USRPRF       Name, *USRPRF
                                                      F4=Liste
    Bibliothek . . . . . . . . . . . .     __________   Name, *LIBL, *CURLIB
  COMMIT-Steuerungsstufe . . . . . . .   1              1=Keine, 2=Ändern
                                                      3=Cursorstabilität
                                                      4=Alle
                                                      5=Wiederh. Lesen
                                                              Weitere ...
F3=Verlassen   F4=Bedienerführung   F5=Aktualisieren   F12=Abbrechen
F22=QM-Anweisung
```

QM Profil ändern – Seite 3

Interessanter wird es wieder auf der letzten Seite – der Seite 4. Hier legen Sie den „Standardmodus zur Abfrageerstellung“ fest. Sie entscheiden, ob standardmäßig die menügeführte Oberfläche oder das SQL-Fenster zur QM-Erstellung verwendet werden soll. Ein Wechsel des Erstellungsmodus ist zulässig, wenn Sie den zweiten Parameter „Ändern Abfrageerstellungsmodus zul.“ auf „J“ belassen.

QM Profil ändern – Seite 4

Falls Sie die Verwendung der SQL-Befehle beschränken oder erweitern wollen, dürfen Sie nicht vergessen, auf der vierten Seite den Parameter „Zulässige SQL-Anweisungen auswählen“ auf „J“ zu setzen. Erst danach erscheint der unten dargestellte Bildschirm:

7.5.5
Seite 6

Zulässige SQL-Anweisungen auswählen

Hier können Sie jetzt individuell entscheiden, welche SQL-Anweisungen der QM-Benutzer in seinen QM-Abfragen nutzen kann.

Schauen Sie sich die Benutzerprofileinstellungen einmal in Ruhe an und probieren Sie einige Einstellungen einfach mal aus. Ich denke, Sie werden schnell die für ihre Anforderungen passenden Einstellungen finden.

7.5.5.2 Querymanager – QM-Abfragen erstellen (menügeführt)

Jetzt wollen wir aber eine erste Abfrage erstellen. Doch bevor wir beginnen, möchte ich Ihnen die Aufgabenstellung zeigen: Es soll eine Bestandsauswertung erstellt werden, die auf den Datenbeständen der Dateien „Artikel“ und „Lagerbestand“ basiert.

FIRMA	ARTIKELNUMMER	BEZEICHNUNG	BESTAND	BESTANDSWERT
001	00001	Mauspad Gummi	4	8.00
001	00001	Mauspad Gummi	4	8.00
001	00002	Radiergummi (10)	50	60.00
001	00002	Radiergummi (10)	7	8.40
001	00002	Radiergummi (10)	41	49.20
001	00002	Radiergummi (10)	3	3.60
001	00004	Füllfederhalter	89	613.21
001	00005	Kugelschreiber (10)	9	135.00

Die Anforderung

Wir starten also den Query Manger mit dem Befehl STRQM und öffnen den Menüpunkt 1 „Mit Query Manager Abfragen arbeiten“. Je nachdem, welche Einstellungen Sie im QM-Profil gewählt haben, sehen Sie anschließend am oberen Bildschirm die QM-Standardbibliothek und den Standarderstellungsmodus.

Bei mir erscheinen dort die Bibliothek EPSW07 und der Modus PROMPT. Ich werde also eine menügeführte Abfrage erstellen:

```
                    Mit Query Manager Abfragen arbeiten

Bibliothek . . . . . .   EPSW07      Name, F4=Liste
Abfrageerstellungsmodus  . . :   PROMPT

Auswahl eingeben und Eingabetaste drücken.
  1=Erstellen    2=Ändern    3=Kopieren    4=Löschen    5=Anzeigen    6=Drucken
  7=Umbenennen   9=Ausführen   10=In SQL umsetzen

Ausw.  Abfrage     Art       Beschreibung
 1     Bestand
```

Eine menügesteuerte Abfrage erstellen

7.5.5
Seite 8

Wenn ich die oben gezeigten Eingaben bestätige, öffnet sich ein neuer Dialog. Der Punkt „Dateien“ ist bereits ausgewählt und kann sofort bestätigt werden, so dass wir zunächst einmal die benötigten Dateien auswählen:

```
                                Dateien angeben

Auswahl eingeben und Eingabetaste drücken.

Datei  . . . . . :   A    ARTIKEL_____        Name, F4=Liste
  Bibliothek . . . . . :    EPSW07_____        Name, *LIBL

Datei  . . . . . :   B    lagerbestand_       Name, F4=Liste
  Bibliothek . . . . . :    EPSW07_____        Name, *LIBL

+ Weitere Dateien  . . . .   _

                                                                  Ende
F4=Bedienerführung   F12=Abbrechen   F20=Gesamten Namen anzeigen
F22=QM-Anweisung
```

Die Dateiauswahl

Sie können die Dateinamen direkt eintragen, wenn sie bekannt sind, oder die Dateien mit Hilfe der Funktionstaste F4 auswählen. Anders als in Query/400 sollten Sie im Query Manager aber darauf achten, stets die physischen Dateien zu nutzen und auf die Auswahl der logischen DDS-Dateien verzichten. Warum? Lesen Sie dazu Kapitel 7.6.2.

Sobald ich die oben gemachten Eingaben bestätige, werde ich aufgefordert, die Verknüpfungen der beiden Dateien festzulegen:

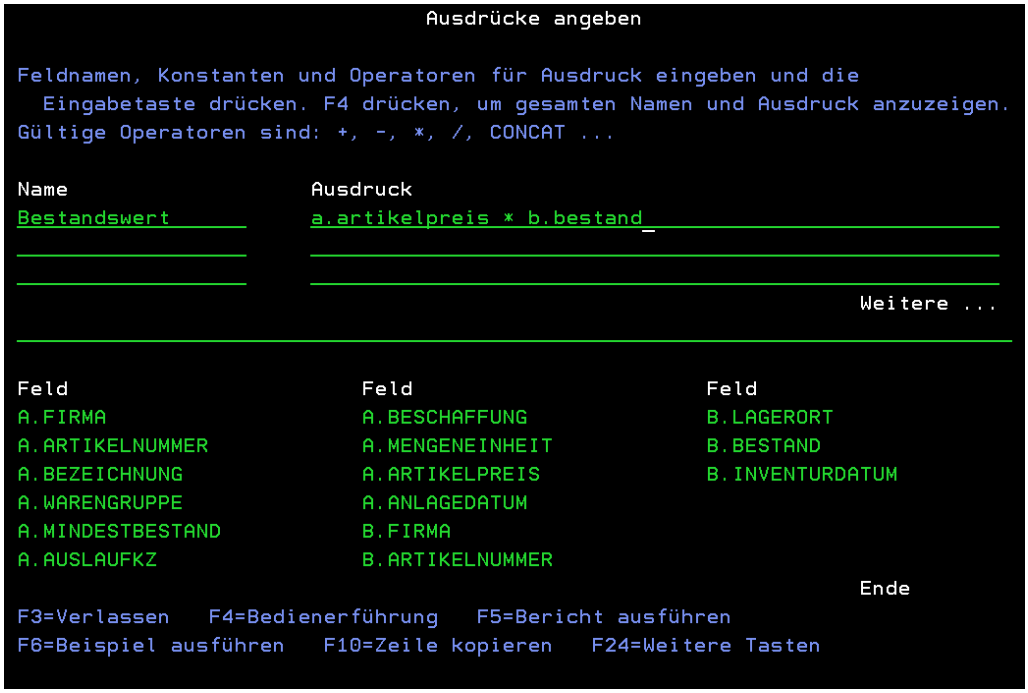
```
                        Verknüpfungsbedingungen angeben

Vergleichswert eingeben, um Beziehungen der Auswahlangaben für die Datei

Test: =, <>, <=, >=, <, >, EQ, NE, LE, GE, LT, GT.

Feld                                Test    Feld
a.firma                              =      b.firma
a.artikelnummer                      eq     b.artikelnummer
                                     __
                                                          Weitere ...

Feld                      Feld                      Feld
A.FIRMA                   A.BESCHAFFUNG             B.LAGERORT
A.ARTIKELNUMMER           A.MENGENEINHEIT           B.BESTAND
A.BEZEICHNUNG             A.ARTIKELPREIS            B.INVENTURDATUM
A.WARENGRUPPE             A.ANLAGEDATUM
A.MINDESTBESTAND          B.FIRMA
A.AUSLAUFKZ               B.ARTIKELNUMMER
                                                          Ende
F3=Verlassen   F4=Bedienerführung          F5=Bericht ausführen
F6=Beispiel ausführen   F11=Art anzeigen   F12=Abbrechen   F24=Weitere Tasten
```

Die Dateiverknüpfungen

Der QM bietet zunächst alle Felder der ausgewählten Dateien an. Wenn Sie aber eine Spalte – in meinem Beispiel die Spalte A.Firma – und den Vergleichsoperator eingetragen haben, bietet es sich an, in der letzten Spalte die Funktionstaste F4 zu verwenden. Es werden Ihnen dann nur noch die für die Verknüpfung in Frage kommenden Spalten der zweiten Datei angezeigt. So ist die Auswahl sehr viel einfacher.

7.5.5
Seite 10

Ist die Dateiverknüpfung hinterlegt, kehren Sie zum Ausgangsbildschirm zurück. Bauen Sie jetzt am besten die temporären Spalten auf. Wie in der folgenden Abbildung gezeigt auf:



Temporäre Spalten aufbauen

Ausdrücke benötigen Sie, wenn neue Spalten aufgebaut werden sollen. In meinem Beispiel berechne ich den Bestandswert, der als solcher nicht in den Tabellen vorkommt. Ausdrücke werden aber auch benötigt, wenn Typkonvertierungen nötig sind oder Feldinhalte ausgeschnitten werden müssen. Aber Vorsicht, es stehen in der menügeführten Oberfläche nicht alle SQL-Funktionen zur Verfügung. Die F1-Taste bietet hier eine gute Hilfe.

Jetzt können wir in einem weiteren Schritt die Felder auswählen, die wir im Bericht verwenden wollen. Dazu öffnen wir den Menüpunkt „Felder auswählen und anordnen“:

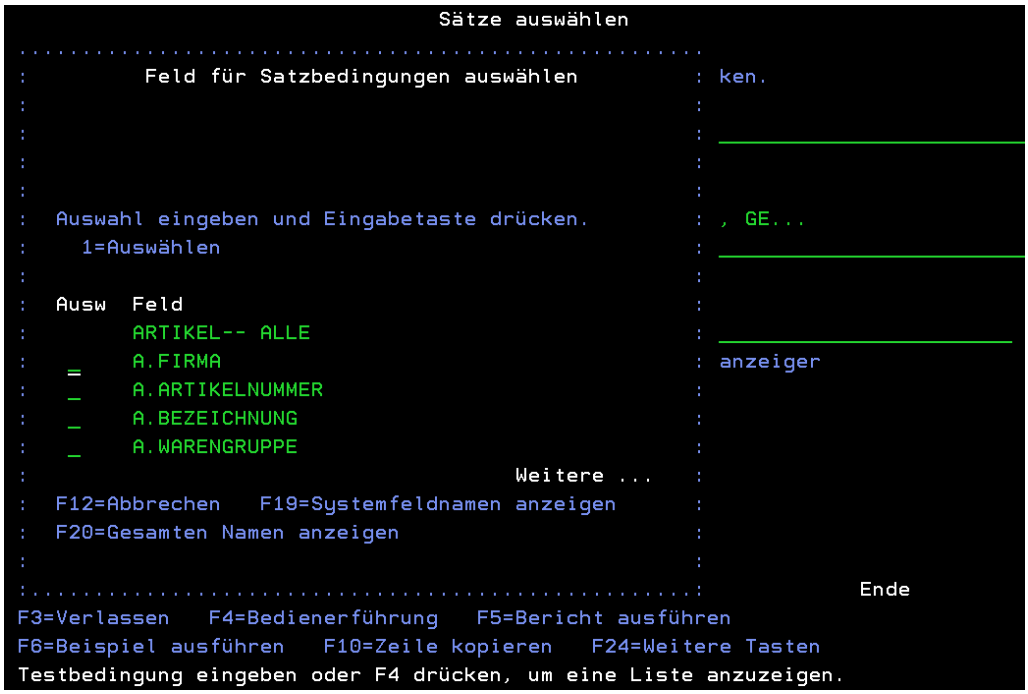
```
                          Felder auswählen und anordnen

Eine Folgenummer (0-9999) für den Namen jedes Felds im Bericht eingeben
  und die Eingabetaste drücken.


Flg.  Feld                                 Flg.  Feld
_ 10    A.FIRMA                            ____  EPSW07.LAGERBESTAND-- ALLE
  20    A.ARTIKELNUMMER                    ____    B.FIRMA
  30    A.BEZEICHNUNG                      ____    B.ARTIKELNUMMER
  40    B.BESTAND                          ____    B.LAGERORT
  50    BESTANDSWERT                       ____    B.INVENTURDATUM
____  EPSW07.ARTIKEL-- ALLE
____    A.WARENGRUPPE
____    A.MINDESTBESTAND
____    A.AUSLAUFKZ
____    A.BESCHAFFUNG
____    A.MENGENEINHEIT
____    A.ARTIKELPREIS
____    A.ANLAGEDATUM
                                                                      Ende
F3=Verlassen          F5=Bericht ausführen   F6=Beispiel ausführen
F10=Spalte kopieren   F11=Art anzeigen   F12=Abbrechen   F24=Weitere Tasten
Eingabetaste zur Bestätigung drücken.
```

Die Feldauswahl

Ich denke, dieser Dialog muss nicht weiter erläutert werden.

7.5.5
Seite 12

Wenn es erforderlich ist, Sätze von der Verarbeitung auszuschließen, müssen Sie jetzt den Menüpunkt „Sätze auswählen“ aufrufen. Falls Sie ungeübt sind, empfiehlt es sich während des gesamten Dialogs die Taste F4 zu nutzen. Mit Hilfe der Bedienerführung werden Sie Step by Step durch den Dialog geführt, wie die folgenden Abbildungen zeigen:



Satzbedingung – Felder

Satzbedingung – Bedingung formulieren

```
                              Sätze auswählen
.........................................................................
:                    Wert für Vergleichstest angeben                    :
:                                                                       :
:  Wert, Ausdruck oder Feld für Vergleich eingeben, Eingabetaste drücken.  :
:  Wenn BESTANDSWERT ungleich                                           :
:    0_____________________________________________________________   :
:  OR                                                                   :
:    ____________________________________________________________   :
:  OR                                                                   :
:    ____________________________________________________________   :
:                                                          Weitere ...  :
:  F12=Abbrechen                                                        :
:  Funktionstaste nicht zulässig.                                       :
:.......................................................................:
Vorherige Vergleichstests

  (Keine Vergleichstests definiert.)


                                                                Ende
F3=Verlassen   F4=Bedienerführung   F5=Bericht ausführen
F6=Beispiel ausführen   F10=Zeile kopieren   F24=Weitere Tasten
Testbedingung eingeben oder F4 drücken, um eine Liste anzuzeigen.
```

Satzbedingung – Vergleichswert definieren

Wissen sollten Sie, dass die Prüfung auf NULL nicht in Form eines Vergleichswerts eingegeben wird, sondern als „Test“-Operator NL. Schauen Sie sich die nächste Abbildung an, dort können Sie sehen, was gemeint ist.

```
                              Sätze auswählen

Neuen Vergleichstest eingeben und die Eingabetaste drücken.

AND/OR . . . . . . . . . . . .   AND     AND, OR
Feld oder Ausdruck . . . . . .   BESTANDSWERT
_____________________
IS/ISNOT . . . . . . . . . . .   ISNOT   IS, ISNOT
Test . . . . . . . . . . . . .   NL      EQ, LT, LE, GT, GE...
Wert . . . . . . . . . . . . .   ____________________________________
_____________________
_______________________________________________________________________
Zum Ändern des vorherigen Vergleichstests den Positionsanzeiger
auf den Test stellen und die Eingabetaste drücken.

Vorherige Vergleichstests
  Wenn BESTANDSWERT ungleich 0
> Und BESTANDSWERT nicht Null ist, dann

                                                                Ende
F3=Verlassen   F4=Bedienerführung   F5=Bericht ausführen
F6=Beispiel ausführen   F10=Zeile kopieren   F24=Weitere Tasten
```

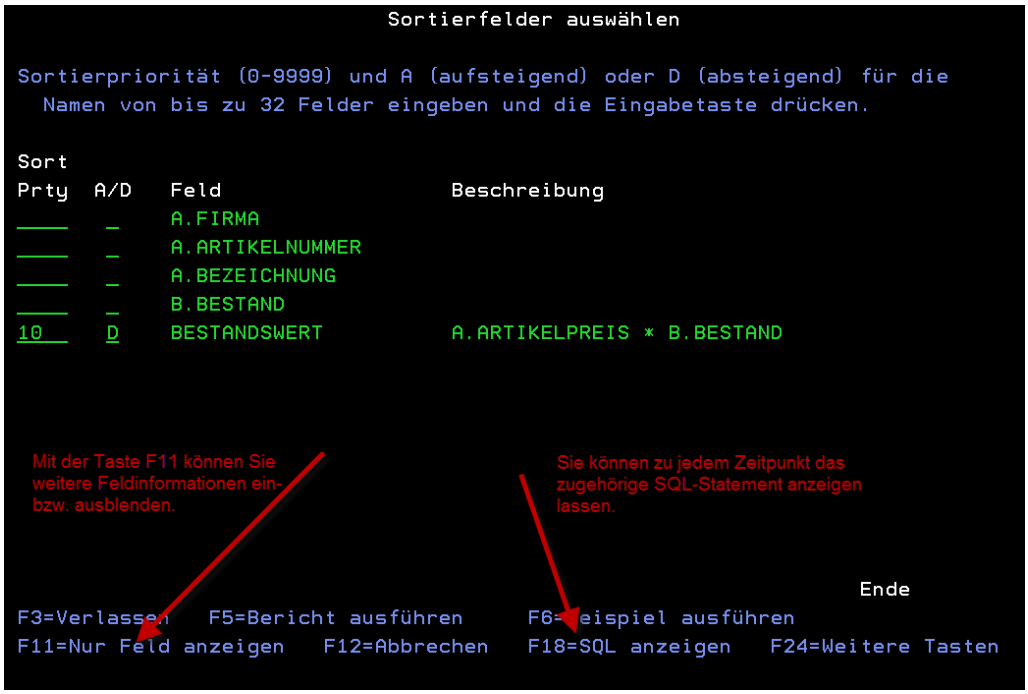
Zum Bearbeiten den Cursor auf die Bedingung positionieren und die Eingabetaste drücken.

Satzbedingung – Vergleichswert ist NULL

7.5.5
Seite 14

Falls Sie eine Bedingung bearbeiten wollen, müssen sie den Cursor auf die entsprechende Bedingung positionieren und anschließend die Funktionstaste F4 drücken. Daraufhin wird die Bedingung wieder im oberen Bildschirmteil angezeigt und kann anschließend problemlos bearbeitet werden.

Falls Ihr Bericht sortiert werden soll, müssen Sie den Menüpunkt „Sortierfelder auswählen“ aufrufen. Ich sortiere meine Daten absteigend nach dem Bestandswert.



Sortierfelder festlegen

Praktisch finde ich im QM, dass Sie sich an vielen Stellen zusätzliche Feldinformationen einblenden können, wie beispielsweise Datentyp, Feldlänge etc. Und auch das zu Grunde liegende SQL-Statement können Sie sich anzeigen lassen. Vielleicht wollen Sie aber auch nur mal sehen, wie ihr Report ausschaut? Dann steht Ihnen über die Funktionstaste F6 immer eine Vorschau zur Verfügung. Achten Sie darauf, dass Sie während der Erstellung immer nur Beispiele ausführen. Die Datenmenge ist dabei durch die Profileinstellungen begrenzt. Ganz anders, wenn Sie die Taste F5 „Bericht ausführen“ verwenden. Mit F5 werden alle Daten abgerufen – das belastet in dieser Phase lediglich Ihren Server und hilft Ihnen nicht weiter!

Auch ich rufe einfach mal einen Beispielbericht auf und kontrolliere so meine bisherige Arbeit:

```
                          Bericht anzeigen
Abfrage . . . .:   EPSW07/BESTAND           Breite. . .:        71
Formular. . . .:   *SYSDFT                  Spalte  . .:         1
Steuerung. . . .   ________
Zeile  ....+....1....+....2....+....3....+....4....+....5....+....6....+....7.
         FIRMA  ARTIKELNUMMER  BEZEICHNUNG           BESTAND    BESTANDSWERT
         -----  ------------  --------------------  -------  --------------
000001   001    00004         Füllfederhalter            89          613,21
000002   001    00005         Kugelschreiber (10)         9          135,00
000003   001    00002         Radiergummi (10)           50           60,00
000004   001    00002         Radiergummi (10)           41           49,20
000005   001    00002         Radiergummi (10)            7            8,40
000006   001    00001         Mauspad Gummi               4            8,00
000007   001    00001         Mauspad Gummi               4            8,00
000008   001    00002         Radiergummi (10)            3            3,60
****** * * * * *  E N D E  D E R  D A T E N  * * * * *
```

Beispielbericht

7.5.5.3 Querymanager – SQL-Abfragen erstellen und ausführen

Dieses Mal wollen wir etwas anders vorgehen. Ich nutze eine SQL-Abfrage im Query Manager und verzichte somit auf die menügeführte Oberfläche. Schauen wir uns auch jetzt die Aufgabenstellung an: Es soll eine wöchentliche Bestandsabfrage angefordert werden.

Die SQL-Abfrage ist schnell geschrieben:

```
SELECT
Firma, Artikelnummer,
Bezeichnung,
SUM(Bestand) As Bestand ,
SUM(Bestand * ArtikelPreis) AS Bestandswert
FROM artikel JOIN lagerbestand USING(Firma, Artikelnummer)
GROUP BY Firma, Artikelnummer, Bezeichnung
ORDER BY Bestandswert DESC ;
```

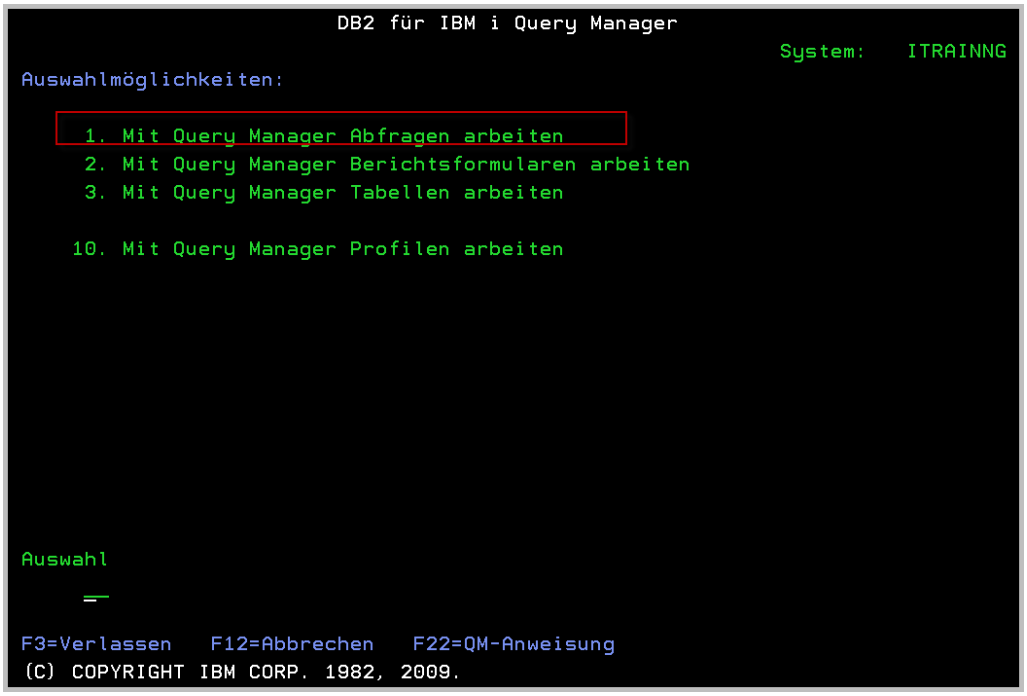
Als Result Set erhalte ich die geforderte Bestandsauswertung mit folgendem Aufbau:

FIRMA	ARTIKELNUMMER	BEZEICHNUNG	BESTAND	BESTANDSWERT
001	00004	Füllfederhalter	89	613.21
001	00005	Kugelschreiber (10)	9	135.00
001	00002	Radiergummi (10)	101	121.20
001	00001	Mauspad Gummi	8	16.00

Die Bestandsauswertung

Da die Abfrage einmal monatlich als Scheduler Job ausgeführt werden soll, entscheide ich mich dazu, die Anforderung mit Hilfe einer Query Manager-Abfrage zu lösen.

Als Erstes rufe ich dafür den Query Manager auf. Dazu geben wir den Befehl STRQM auf einer 5250-Befehlszeile ein. Es erscheint das Startmenü:

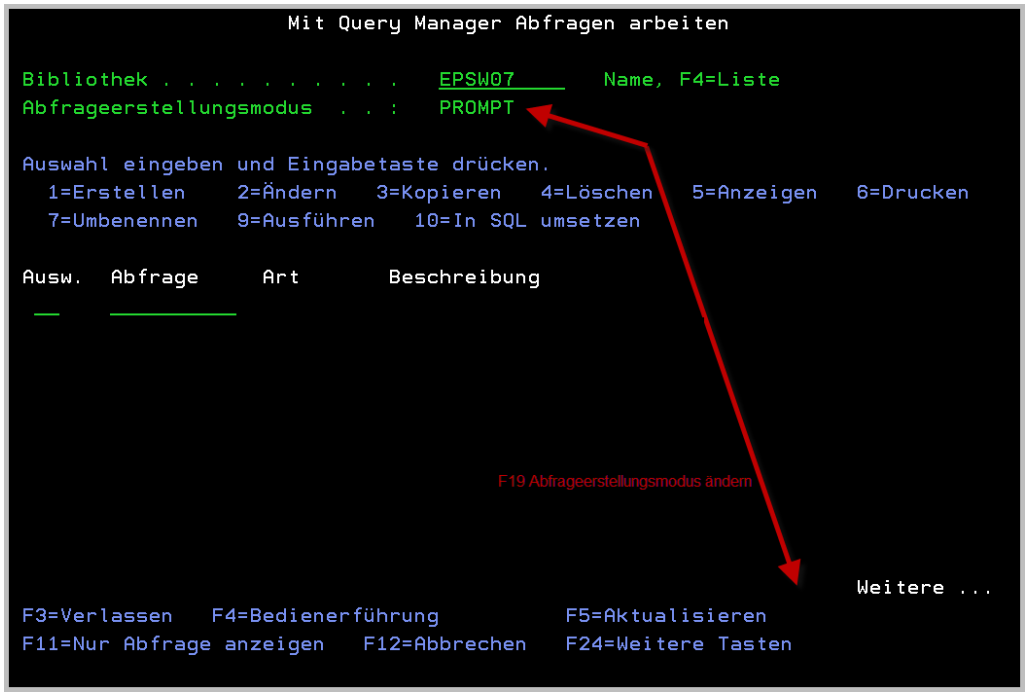


Query Manager-Menü

Menüpunkt 1 „Mit Query Manager Abfragen arbeiten“ dient dazu, Datenbankabfragen zu erstellen und auszuführen. Menüpunkt 2 wird für das Berichtslayout verwendet, und Menüpunkt 3 erlaubt uns die Manipulation von Datenbankdateien. Beachten sollten Sie den Menüpunkt 10. Query Manager erlaubt individuelle Benutzerprofileinstellungen!

7.5.5
Seite 18

Ich wähle den Menüpunkt 1, um die SQL-Abfrage zu hinterlegen. Daraufhin erscheint der folgende Dialog:



Mit Query Manager Abfragen arbeiten

Am oberen Bildschirmrand steht die aktuelle Bibliothek. Den Eintrag können Sie jederzeit problemlos ändern. Direkt darunter finden Sie den Abfrageerstellungsmodus. Wenn Sie die Taste F19 drücken, können Sie zwischen PROMPT- und SQL-Modus wechseln. Da meine SQL-Abfrage bereits fertig ist, wechsle ich in den SQL-Modus. Kapitel 9.5 zeigt Ihnen die menügeführte Erstellung einer QM-Abfrage.

Query Manager Abfragen erstellen – SQL-Modus

Wenn ich meine Eingaben im oberen Bildschirm bestätige, erhalte ich anschließend das folgende Skriptfenster:

```
                              Abfrage editieren
Spalten . . . :  1 70                              Abfrage  . . : BSTWERT
QM . . ____________________________________________________________________

SQL-Anweisung eingeben
        ************************** Anfang der Daten **************************
0001.00  SELECT
0002.00  Firma, Artikelnummer,
0003.00  Bezeichnung,
0004.00  SUM(Bestand) As Bestand ,
0005.00  SUM(Bestand * ArtikelPreis) AS Bestandswert
0006.00  FROM artikel JOIN lagerbestand USING(Firma, Artikelnummer)
0007.00  GROUP BY Firma, Artikelnummer, Bezeichnung
0008.00  ORDER BY Bestandswert DESC
        ***************************** Ende der Daten *************************
```

Query Manager Abfragen erstellen – SQL-Skriptfenster

Im Skriptfenster steht ihnen wie fast überall im IBM i-Umfeld natürlich auch die Bedienerführung (F4-Taste) zur Verfügung; hier für den Fall, dass Sie das SQL-Statement nicht – wie ich – kopieren, sondern direkt aufbauen wollen.

```
                          Anweisung SELECT angeben

Daten für SELECT-Anweisung eingeben. F4=Liste

FROM Dateien . . . . . .     artikel JOIN lagerbestand USING(Firma,
                             Artikelnummer)
SELECT Felder  . . . . . .   Firma, Artikelnummer, Bezeichnung,
                             SUM(Bestand) As Bestand , SUM(Bestand *
                             ArtikelPreis) AS Bestandswert
WHERE Bedingungen. . . . . .
GROUP BY Felder  . . . . . .  Firma, Artikelnummer, Bezeichnung
HAVING Bedingungen . . . . .
                                                              Weitere ...
Auswahl eingeben und Eingabetaste drücken.

  DISTINCT Sätze in Ergebnisdatei  . . . . . .   N    J=Ja, N=Nein
  UNION mit anderer Anweisung SELECT . . . . . .  N    J=Ja, N=Nein
  Weitere Auswahlmöglichkeiten angeben . . . . .  N    J=Ja, N=Nein



F3=Verlassen   F4=Bedienerführung   F5=Aktualisieren   F6=Zeile einfügen
F9=Unterabfrage angeben   F10=Zeile kopieren          F24=Weitere Tasten
```

Die SQL-Bedienerführung

Wer bislang immer noch mit der STRSQL-Umgebung arbeitet, wird sich schnell zu Recht finden, denn die Bedienerführungen sind in beiden Interfaces identisch.

7.5.5
Seite 20

Auch die Zeilensteuerung – Zeile einfügen, Zeile löschen etc. – wird vielen Anwendern vertraut vorkommen. Schließlich gelten hier die gleichen Zeilenbefehle wie im PDM.

Eigentlich kann in dieser Umgebung nichts mehr schief gehen!

Ist das SQL-Statement fertig, verlassen Sie die Skriptumgebung mit der Funktionstaste F3 und speichern Ihre Abfrage:

```
                                  Verlassen

Auswahl eingeben und Eingabetaste drücken.

  Auswahl  . . . .   1              1=Sichern und verlassen
                                    2=Verlassen ohne Sichern
                                    3=Wiederaufnehmen ohne Sichern
                                    4=Sichern und wiederaufnehmen

  Abfrage  . . . .   WK_BSTWERT     Name
    Bibliothek . .     EPSW07       Name, *CURLIB
  Beschreibung . .   wöchentliche Bestandsauswertung
```

Query Manager-Abfrage speichern

Erstellt wurde auf diesem Weg ein Objekt mit Namen WK_BSTWERT der Art *QMQRY.

```
                          Mit Objekten arbeiten

Auswahl eingeben und Eingabetaste drücken.
  2=Berechtigung editieren   3=Kopieren   4=Löschen   5=Berechtigung anzeigen
  7=Umbenennen   8=Beschreibung anzeigen   13=Beschreibung ändern

Opt  Objekt      Art       Bibliothek  Attribut    Text
__   WK_BSTWERT  *QMQRY    EPSW07      SQL         wöchentliche Bestandsauswer
```

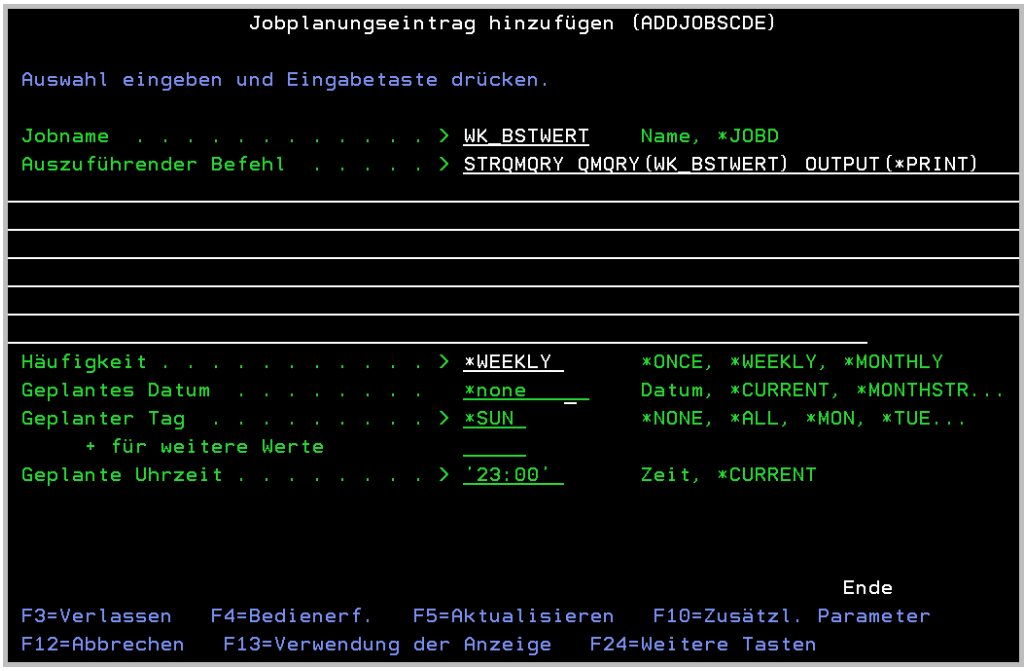
Das fertige Ausführungsobjekt

Die Query-Abfrage können Sie anschließend mit der Option 9 im Query Manager oder mit dem Betriebssystembefehl STRQMQRY ausführen.

```
                  Abfrageverwalt.abfrage starten (STRQMQRY)

Auswahl eingeben und Eingabetaste drücken.

Abfrageverwaltungsabfrage  . . . > WK_BSTWERT    Name
  Bibliothek . . . . . . . . . .   EPSW07       Name, *LIBL, *CURLIB
Ausgabe  . . . . . . . . . . . . > *PRINT       *, *PRINT, *OUTFILE
Abfrageverw.-Berichtsformular  .   *SYSDFT      Name, *SYSDFT, *QMQRY
  Bibliothek . . . . . . . . . .                Name, *LIBL, *CURLIB
Datum und Zeit . . . . . . . . .   *YES          *YES, *NO
Seitenzahlen . . . . . . . . . .   *YES          *YES, *NO
```

Der Befehl STRQMQRY

Natürlich lässt sich der Aufruf jetzt auch genauso gut als Scheduler Job hinterlegen:



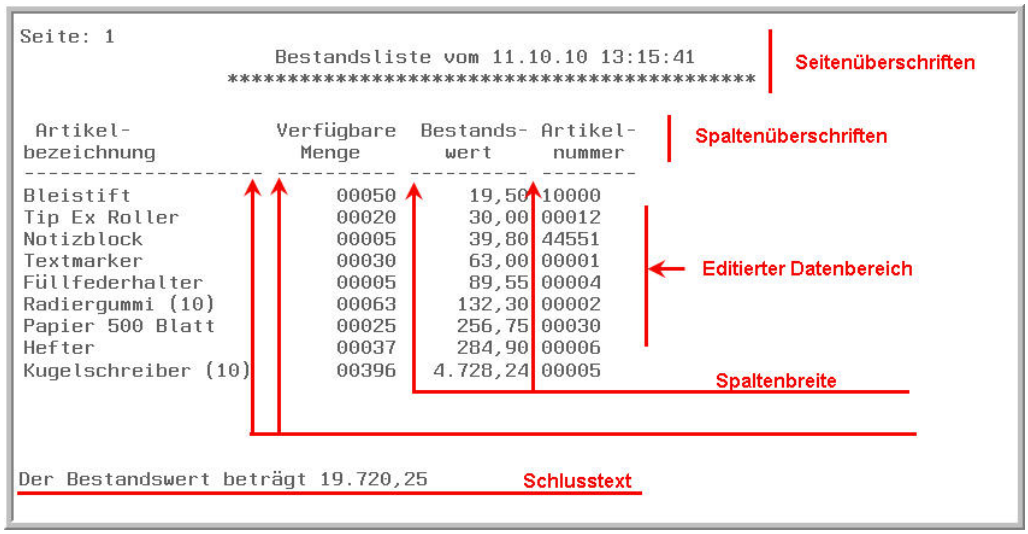
Der Scheduler-Eintrag

Schneller und einfacher lassen sich derartige Reports nicht integrieren. Und wer an dem Berichtslayout etwas zu bemängeln hat, dem empfehle ich an dieser Stelle das folgende Kapitel. Dort finden Sie eine Beschreibung der Reporterstellung. Sie werden feststellen, dass sich auch mit dem Query Manager die Reports individuell aufbauen lassen.

7.5.5
Seite 22

7.5.5.4 Querymanager Formulare (*QMFORM) – einfache Berichte

Um die vielfältigen Möglichkeiten der Query-Formulare zu verstehen, schauen wir uns zunächst einen anderen Report an:

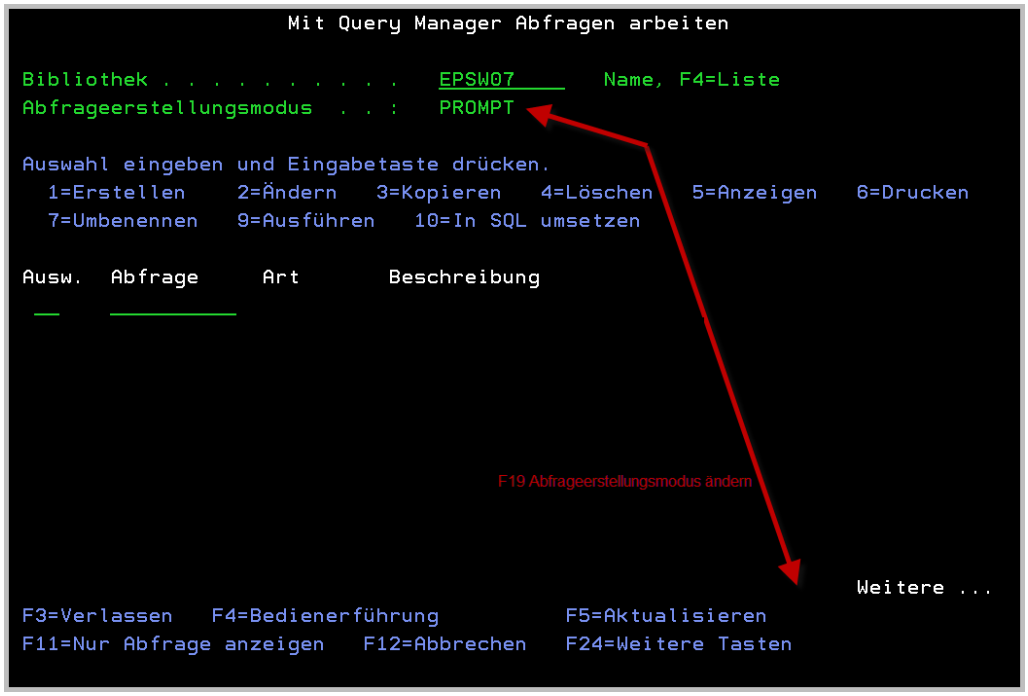


Unser Ziel – der fertige Report

Der Bericht enthält verschiedene Datenbankfelder, einige Überschriften und am Ende stehen zusammenfassende Informationen, in unserem Beispiel die Bestandsgesamtsumme.

Um dieses Berichtsformular zu generieren, rufen wir zunächst wieder den Query Manager auf. Wir könnten den Menüpunkt 2 „Mit Query Manager Berichtsformularen arbeiten“ wählen, um ein Formular zu erstellen. Ich muss allerdings zunächst noch die Abfrage definieren und entscheide mich daher für den Menüpunkt 1 „Mit Query Manager Abfragen arbeiten“.

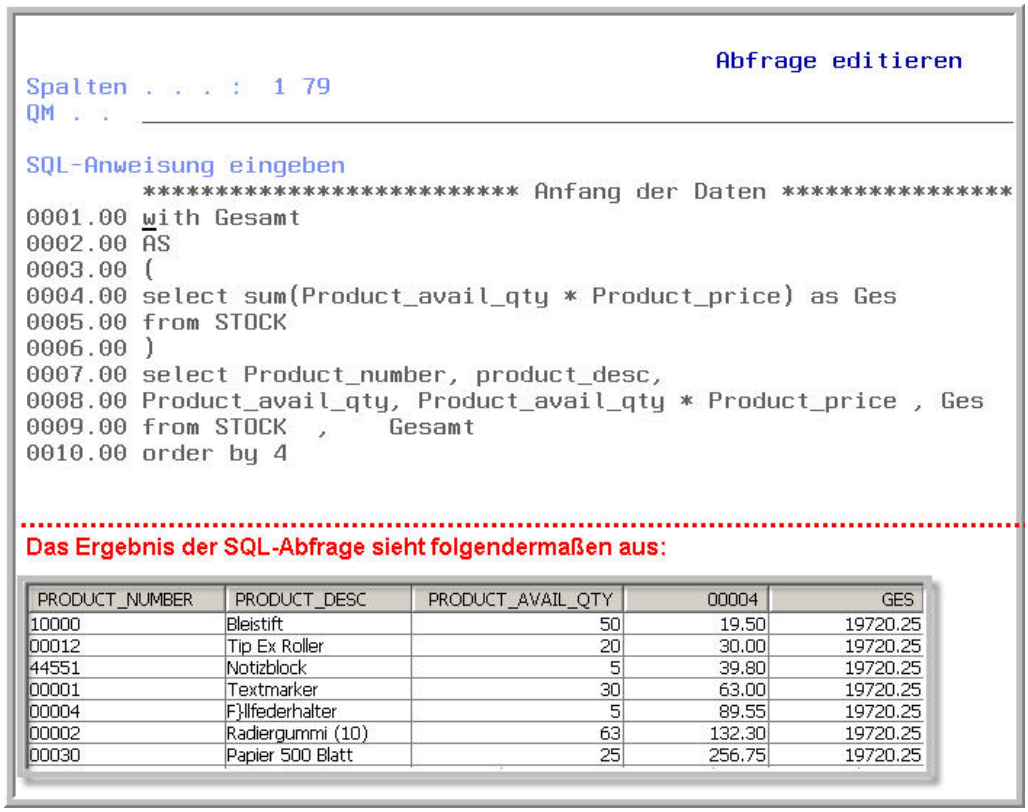
Auch dieses Mal verwende ich ein SQL-Statement und nicht die menügeführte Oberfläche. Also wechsle ich zunächst wieder in den SQL-Modus:



Mit Query Manager Abfragen arbeiten

Wenn ich jetzt eine Abfrage mittels Option 1 erstelle, öffnet sich das SQL-Skriptfenster, in das ich die fertige SQL-Abfrage hineinkopieren kann.

7.5.5
Seite 24



PRODUCT_NUMBER	PRODUCT_DESC	PRODUCT_AVAIL_QTY	00004	GES
10000	Bleistift	50	19.50	19720.25
00012	Tip Ex Roller	20	30.00	19720.25
44551	Notizblock	5	39.80	19720.25
00001	Textmarker	30	63.00	19720.25
00004	F}llfederhalter	5	89.55	19720.25
00002	Radiergummi (10)	63	132.30	19720.25
00030	Papier 500 Blatt	25	256.75	19720.25

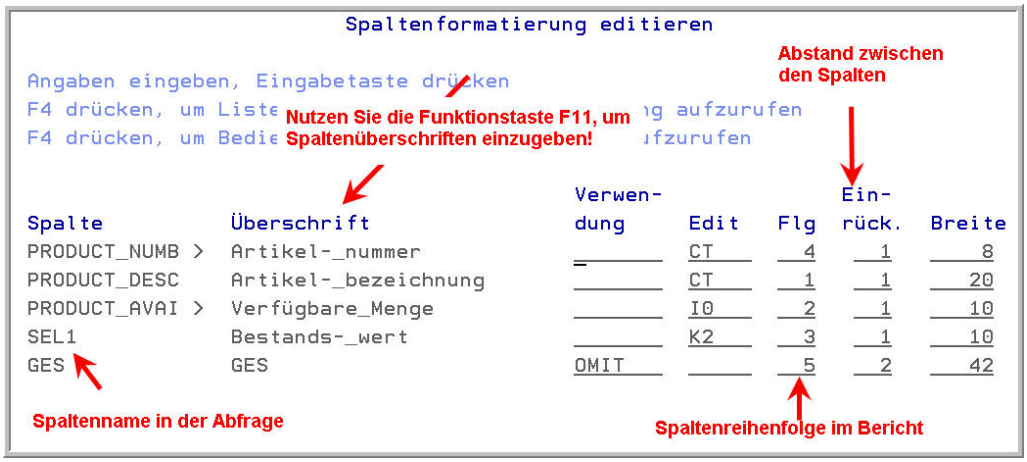
Die auszuführende SQL-Abfrage

Meine SQL-Abfrage generiert ein Result Set, das drei Spalten der Tabelle STOCK enthält, zusätzlich sind zwei berechnete Spalten vorhanden, die einerseits den Bestandswert des Artikels und andererseits den Bestandswert aller Artikel enthält.

Um jetzt für diese Abfrage einen Report zu erstellen, wechseln wir mit Hilfe der Funktionstaste F13 in den Dialog „Berichtsformat auswählen“ und sollten daraufhin folgenden Bildschirm angezeigt bekommen:

Berichtsformat auswählen

Sie können anhand der oberen Abbildung erkennen, dass das Berichtsformular den gleichen Namen erhält wie die Abfrage selbst. Nun müssen wir die Spalten definieren, die im Bericht enthalten sein sollen:



Spaltenformatierung editieren

Als Erstes sollten Sie die Spaltenüberschriften bestimmen. Wenn Sie keine eigenen Überschriften im Report definieren, bestimmt die Abfrage die Überschriften der Berichtsspalten. Da aber die Spaltennamen in den Tabellen häufig nicht sprechend sind und die Abfragen zudem oftmals temporäre Spalten in Form von Berechnungen enthalten, sollten Sie die Überschriften im Report selbst bestimmen. Drücken Sie hierfür die Funktionstaste F11. Überschriften können Groß- und Kleinbuchstaben, Ziffern, Leerzeichen und Sonderzeichen enthalten. Sie können auch Überschriften definieren, die sich über mehr als eine Zeile erstrecken. Dafür verwenden Sie innerhalb der Überschrift den Unterstrich (_). An Stelle des Unterstrichs erfolgt zur Laufzeit der Zeilenumbruch. Maximal können so 9 Zeilen für die Überschrift verwendet werden. Vielleicht wollen Sie aber auch auf eine Überschrift verzichten? Dann wird *NONE eingegeben. Nach Eingabe der Spaltenüberschriften drücken Sie einfach wieder die Funktionstaste F11, um zur ursprünglichen Anzeige „Spaltenformatierung editieren“ zurückzukehren. Hier bestimmen Sie anschließend die Breite der Spalte im Formular, den Abstand zwischen den einzelnen Berichtsspalten und die Reihenfolge der Spalten im Bericht. Standardmäßig wird die Reihenfolge verwendet, in der die Spalten durch die Anweisung SELECT oder die menügesteuerte Abfrage abgerufen wurden. Abschließend müssen wir die Spalten noch editieren. Der Editiercode bestimmt, ob es sich um Zeichendaten, numerische Daten, Grafikdaten oder temporale Ausgabedaten handelt. In Abhängigkeit vom jeweiligen Datentyp stehen verschiedene Editiercodes zur Verfügung, die die Ausgabe der Daten im Bericht bestimmen.

7.5.5
Seite 26

Editiercodes für Zeichendaten	
Editiercode	**Bedeutung**
C	Der Text wird abgeschnitten, wenn die Berichtsspalte kürzer ist als der darzustellende Spalteninhalt.
CW	Der Text wird in die nächste Zeile umgebrochen, wenn die Berichtsspalte kürzer ist als der darzustellende Spalteninhalt.
CT	Der Text wird in die nächste Zeile umgebrochen, wenn die Berichtsspalte kürzer ist als der darzustellende Spalteninhalt. Wortgrenzen werden dabei beachtet.

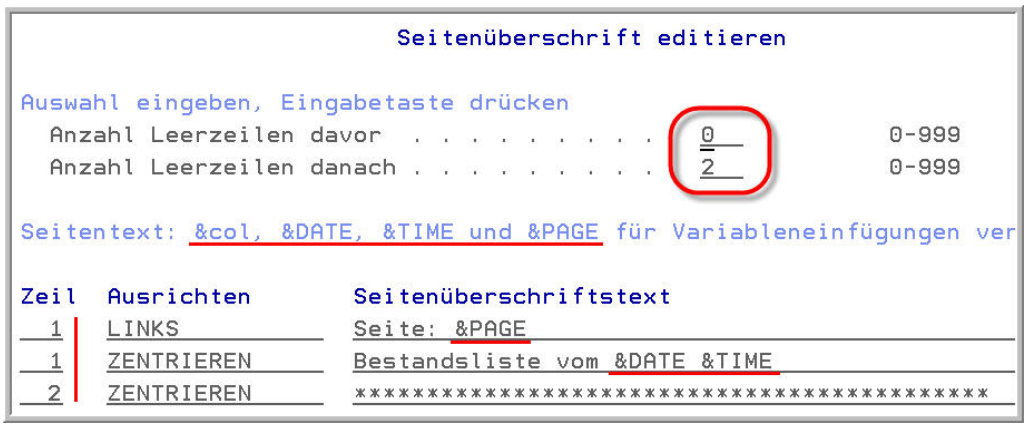
Die Editierung der numerischen Spalten erlaubt einerseits die Angabe der Nachkommastellen, andererseits können auch für numerische Spalten verschiedene Editiercodes verwendet werden:

Editiercodes für numerische Daten	
Editiercode	**Bedeutung**
D	Währungszeichen ($) und Tausenderpunkt (Bsp. -$12.345,67)
I	Führende Nullen werden dargestellt (Bsp. -0012345,67)
J	Führende Nullen, kein negatives Vorzeichen (Bsp. 0012345,67)
K	Tausenderpunkt (Bsp. -12.345,67)
L	Dezimalschreibweise (Bsp. -12345,67)
E	Exponentialschreibweise (Bsp. -1,234567E+04)
P	Prozentzeichen und Tausenderpunkt (Bsp. -1,234,457.89%)

Auch für Datums- und Zeitfelder stehen verschiedene Editiercodes zur Verfügung. Hier empfiehlt es sich jedoch, auf die Reportcodes vollständig zu verzichten, solange Sie mit europäischem Datumsformat arbeiten.

Der Datentyp der Tabellenspalten kann im Report nicht implizit geändert werden, das heißt, eine numerische Spalte muss im Report als solche gekennzeichnet werden und umgekehrt. Eine Datentypkonvertierung ist somit nicht über das Berichtsformular möglich. Außerdem müssen Sie für jede Spalte Ihrer Abfrage auch eine Berichtsspalte definieren. Wenn Sie im Report mehr Spalten definieren als in der Abfrage erstellt werden, würden Sie Laufzeitfehler erhalten. Das gilt natürlich auch für den umgekehrten Fall. Aus diesem Grund hinterlege ich auch für meine Spalte GES eine Spaltenformatierung, verwende aber das Schlüsselwort OMIT, um kenntlich zu machen, dass die Spalte nicht im Datenteil verwendet werden soll.

Als Nächstes legen wir die Seitenüberschriften fest:



Seitenüberschriften editieren

Zunächst legen wir fest, wie viele Zeilen vor der Überschrift und wie viele Leerzeilen nach der Überschrift eingefügt werden sollen. Dann hinterlegen wir den Text, der auf jeder gedruckten Berichtsseite erscheinen soll. Standardmäßig werden hierfür fünf Zeilen zur Verfügung gestellt. Sie können allerdings weitere Überschriften hinzufügen, wenn Sie die Funktionstaste F6 (Zeilen einfügen) drücken. Durch Angabe von „LINKS“, „RECHTS“ bzw. „ZENTRIEREN“ bestimmen Sie die Ausrichtung des Texts, den Sie in der Spalte „Seitenüberschriftstext“ hinterlegen. Für die Eingabe des Texts stehen 55 Zeichen zur Verfügung. Allerdings dürfen Überschriften bis zu 132 Zeichen Text enthalten. Um diese Diskrepanz zu umgehen, müssen Sie Ihre Überschrift in der nächsten Zeile fortsetzen, aber die gleiche Zeilennummer verwenden. So werden die Texte in einer Zeile geschrieben. Durch verschiedene Ausrichtungen müssen Sie anschließend das Problem der Textüberlagerungen umgehen.

Schauen Sie sich in der oberen Abbildung die Definition der Seitenzahl an. Sie wird in der ersten Überschriftenzeile gedruckt. In dieselbe Zeile soll aber auch der Text „Bestandsliste vom ….“ gedruckt werden. Beide Definitionen erhalten im Beispiel die Zeilennummer 1; die Seitenzahl wird allerdings linksbündig ausgerichtet, während die eigentliche Überschrift zentriert wird. Außerdem können Sie im Beispiel erkennen, dass ich die Textvariablen &DATE, &TIME und &PAGE verwende, um Datum, Uhrzeit und Seitenzahlen in den Bericht zu integrieren. Achten Sie bei Verwendung der Textvariablen auf die Großschreibung.

7.5.5
Seite 28

Sobald alle Überschriften definiert wurden, kehre ich wieder zum Ausgangsbildschirm zurück, um für meinen Beispielreport den Schlusstext zu hinterlegen:

```
                          Berichtsformat auswählen

Formular . . . . . :   ART_QTY01

Auswahl eingeben und Eingabetaste drücken. F21 drücken,
  1=Auswählen


  Ausw.   Berichtsformat
  _       Spaltenformatierung editieren
  _       Seitenüberschrift editieren
  _       Seitenfuß editieren
  1       Schlusstext editieren
  _       Gruppenwechseltext editieren
  _       Auswahlmöglichkeiten zum Formatieren angeben
```

Berichtsformat – Schlusstext

Auf „Seitenfußtexte“ verzichte ich im Beispielreport, da ich die Seitenzahlen ja bereits in die Überschriften integriert habe. Mir fehlt lediglich der Schlussbestand, der einmalig am Ende des Berichts ausgegeben werden soll.

```
                          Schlusstext editieren

Auswahl eingeben, Eingabetaste drücken
  Neue Seite für Schlußtext . . . . . .   N          J=Ja, N=Nein
  Anzahl Leerzeilen davor . . . . . .     2          0-999, BOTTOM
  Zeile für Endergebnis . . . . . .       1          1-999, NONE

Schlußtext: &col für Variableneinfügung verwenden.

Zeil  Ausrichten       Schlußtext
  1   LINKS            Der Bestandswert beträgt &5
```

Schlusstext editieren

Als Erstes können Sie wieder die Position des Schlusstexts bestimmen, wobei der Eintrag „Zeile für Endergebnis“ nur wirksam ist, wenn Sie sich entscheiden, die Schlussbemerkungen auf einer eigenen Seite zu drucken. Ich verzichte darauf und verwende lediglich eine Zeile, die linksbündig ausgerichtet werden soll. Im Schlusstext nehme ich allerdings Bezug auf die fünfte Spalte meiner Abfrage; sie enthält den Gesamtbestandswert, wird im Bericht nicht mit ausgegeben, sondern lediglich als Abschlussangabe am Ende des Berichts benötigt.

Wenn alle Angaben soweit hinterlegt sind, kann ich den Query Manager verlassen. Automatisch werden Sie dabei zum Speichern des Reports und der Abfrage aufgefordert.

Bevor wir nun den Bericht ausführen, wollen wir uns ansehen, welche Objekte generiert wurden:

```
                          Mit Objekten arbeiten

 Auswahl eingeben und Eingabetaste drücken.
   2=Berechtigung editieren   3=Kopieren   4=Löschen   5=Berechtigung anzeigen
   7=Umbenennen   8=Beschreibung anzeigen   13=Beschreibung ändern

 Opt  Objekt      Art       Bibliothek  Attribut    Text
 __   ART_QTY01   *QMQRY    EPS0042CR   SQL         Artikelbestand  - FORM ART
 __   ART_QTY01   *QMFORM   EPS0042CR   QUERY MGR   Artikelbestand
```

Die generierten QM-Objekte

Die Abfrage wurde als Objekt der Art *QMQRY gespeichert, während der Bericht als eigenständiges Objekt der Art *QMFORM abgelegt wurde. So ist es grundsätzlich möglich, für eine Abfrage verschiedene Berichtsformulare zu verwenden, solange die Spaltenanzahl und die Datentypen der Abfrage beachtet werden.

Abschließend soll die Abfrage ausgeführt werden: Dazu verwenden wir wiederum den Betriebssystembefehl STRQMQRY:

```
                  Abfrageverwalt.abfrage starten (STRQMQRY)

 Auswahl eingeben und Eingabetaste drücken.        Name der Abfrage (*QMQRY)

 Abfrageverwaltungsabfrage  . . . > ART_QTY01      Name
   Bibliothek . . . . . .             *LIBL        Name, *LIBL, *CURLIB
 Ausgabe  . . . . . . . . . . . . > *PRINT
 Abfrageverw.-Berichtsformular  . > *QMQRY         verwendeter Report
   Bibliothek . . . . . .                          Name, *LIBL, *CURLIB
 Datum und Zeit . . . . . . . . . > *NO            *YES, *NO
 Seitenzahlen . . . . . . . . . . > *NO            *YES, *NO
```

Die Berichtsausführung

Sie müssen lediglich den Namen der Abfrage eingeben und ein Berichtsformular zuordnen. Da mein Formular wie die Abfrage benannt wurde, kann ich den Sonderwert *QMQRY verwenden. Alternativ könnte ich auch explizit den Namen eines Formulars eintragen. Nur den Sonderwert *SYSDFT sollten Sie nicht beibehalten, weil sonst ein Standardbericht erstellt würde und unser erstelltes Berichtsformular ungenutzt bliebe. Außerdem sollen Datum, Uhrzeit und Seitenzahlen dem Bericht nicht automatisch hinzugefügt werden. Schließlich habe ich diese Informationen bereits im Berichtsformular verwendet.

Bei meinem Beispiel handelt es sich um einen sehr einfachen Bericht. Doch man kann auf diesem Weg auch Formulare für Gruppenwechsel realisieren. Und das schauen wir uns im nächsten Kapitel an.

7.5.5
Seite 30

7.5.5.5 Query Manager–Formulare (*QMFORM) – Gruppenwechsel

Im vorangegangenen Kapitel ging es darum, die Grundfunktionen der Query Manager-(QM-)Formulare kennenzulernen. Doch können mit Hilfe der Formulare auch Reports realisiert werden, die einerseits Detaildaten, andererseits Summenwerte beinhalten. Wir wollen jetzt einen Summenbericht erstellen.

Um die Summenfunktionen der QM-Formulare zu verstehen, schauen wir uns zunächst die SQL-Abfrage an:

```
WITH BESTAND
AS
(SELECT FIRMA, ARTIKELNUMMER, LAGERORT, SUM(BESTAND) AS BE-
STANDSMENGE , SUM(BESTAND * ARTIKELPREIS) AS BESTANDSWERT
FROM ARTIKEL JOIN LAGERBESTAND USING(FIRMA, ARTIKELNUMMER)
GROUP BY FIRMA, ARTIKELNUMMER, LAGERORT
)

SELECT FIRMA, ARTIKELNUMMER, BEZEICHNUNG, LAGERORT, BESTANDS-
MENGE, BESTANDSWERT
FROM ARTIKEL JOIN BESTAND USING(FIRMA, ARTIKELNUMMER)
ORDER BY ARTIKELNUMMER ;
```

Ich greife zwar die SQL-Abfrage aus den vorangegangenen Kapiteln wieder auf, doch ein wenig wurde die Abfrage modifiziert, um die Bestände je Lagerort auszuwerten. Die SQL-Abfrage liefert das folgende Result Set:

FIRMA	ARTIKELNUMMER	BEZEICHNUNG	LAGERORT	BESTANDSMENGE	BESTANDSWERT
001	00001	Mauspad Gummi	25	4	8.00
001	00001	Mauspad Gummi	10	4	8.00
001	00002	Radiergummi (10)	33	7	8.40
001	00002	Radiergummi (10)	20	41	49.20
001	00002	Radiergummi (10)	13	3	3.60
001	00002	Radiergummi (10)	27	50	60.00
001	00004	Füllfederhalter	21	89	613.21
001	00005	Kugelschreiber (10)	22	9	135.00

Das Result Set der Abfrage

Der fertige Bericht soll später folgendes Layout haben:

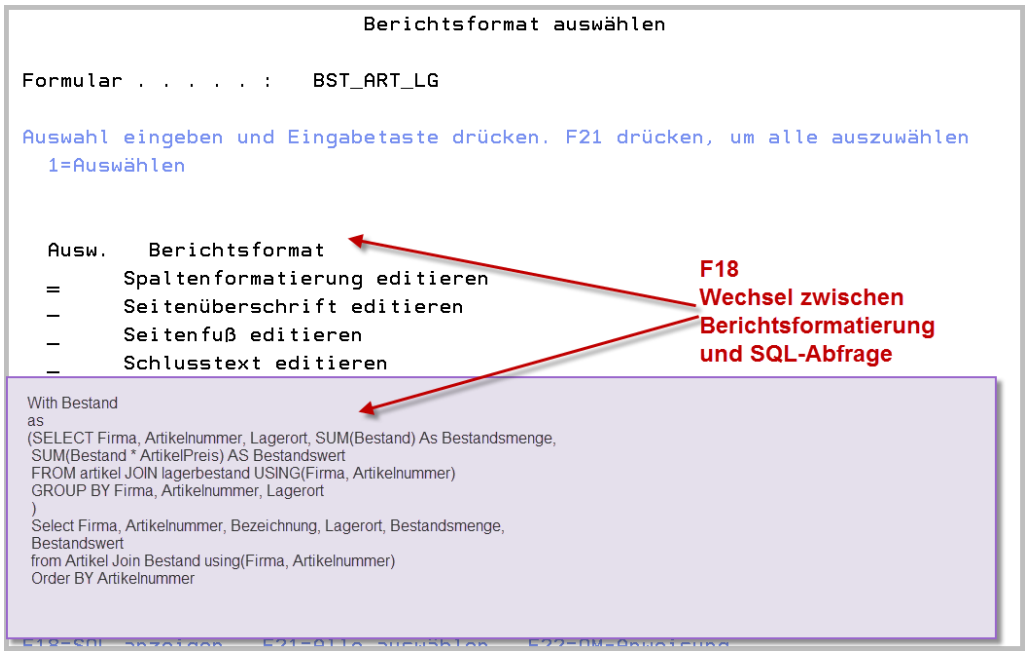
```
                                          Bestands    Bestands
                                Lager      menge        Wert


  001    Kugelschreiber (10)    22              9      135,00
                                        ----------  ----------
Bestand von Artikel Kugelschreiber              9      135,00


Der Bestandswert beträgt insgesamt:           207      885,41
```

Der fertige Summenbericht

Um nun hierfür den gezeigten Report zu erstellen, rufen wir wieder den QM auf. Ich gehe davon aus, dass die Abfrage bereits hinterlegt wurde, so dass wir direkt mit dem Report beginnen können und den Menüpunkt 2 „Mit Query Manager Berichtsformularen arbeiten“ ausführen.

Da der neue Bericht meinem ersten Report ähnelt, kopiere ich einfach das im letzten Kapitel erstellte Formular und nehme anschließend lediglich Anpassungen vor. Das neue Formular nenne ich BST_ART_LG. Ist die Kopie erstellt, kann ich das Formular entsprechend anpassen (Option 2 = Ändern).

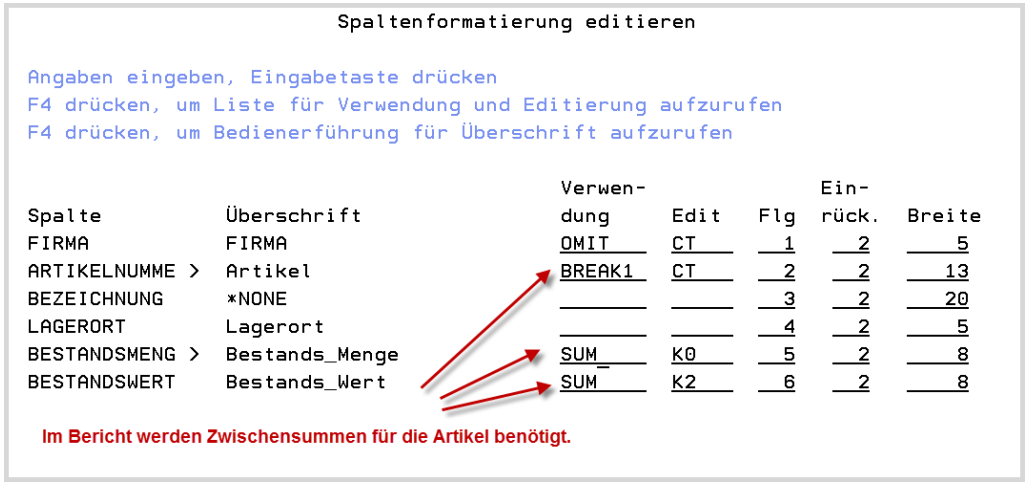
Falls Sie die SQL-Abfrage bisher noch nicht geändert haben, können Sie das auch jetzt noch machen. Denn obwohl wir uns derzeit in der Berichtsformatierung befinden, können wir jederzeit zwischen der aktiven SQL-Abfrage und dem Formular wechseln. Dazu nutzt man einfach die Funktionstaste F13. Auf diesem Weg ist es sehr einfach, die entsprechende SQL-Abfrage anzupassen.



Wechsel zwischen Berichtsformular und SQL-Abfrage

7.5.5
Seite 32

Meine SQL-Abfrage muss ich nicht mehr modifizieren. Sie ist bereits getestet und läuft einwandfrei. Ich kehre daher wieder zu meinem Berichtsformular zurück und öffne den Menüpunkt „Spaltenformatierung editieren“, so dass folgender Dialog erscheint:



Spaltenformatierung editieren

Erneut müssen wir die Spalten definieren, die im Bericht enthalten sein sollen: Als Erstes bestimmen wir wieder die Spaltenüberschriften. Dieses Mal verzichte ich allerdings auf Spaltenüberschriften für die Artikelbezeichnung. Außerdem benötige ich die Firmennummer nicht für die Detaildaten, sondern später lediglich für die Überschriften, so dass ich die Spalte mit dem Schlüsselwort OMIT für den Report ausschließe. Das geschieht allerdings nur aus optischen Gründen und ist nicht zwingend notwendig.

Die Spalte „Verwendung“ kennen Sie bereits aus dem ersten Beispiel. Sie bestimmt dieses Mal den Gruppenwechsel. Man spricht von einem Gruppenwechsel, wenn sich der Wert in der Spalte „Artikelnummer“ ändert, der Spaltenwert also wechselt. Sobald Sie derartige Gruppenwechselstufen im Formular verwenden, müssen die Daten anhand dieser Spalte sortiert werden!. Nun müssen Sie noch die Summenbildung aktivieren, indem Sie für die Spalten „Bestandswert“ und „Bestandsmenge“ das Schlüsselwort SUM verwenden. Alternativ stehen folgende Schlüsselwörter zur Verfügung:

Schlüsselwort	Bedeutung
SUM	Summe der entsprechenden Spaltenwerte
AVERAGE	Durchschnitt der entsprechenden Spaltenwerte
COUNT	Anzahl der entsprechenden Spaltenwerte
FIRST	Erster Wert in der entsprechenden Spalte
LAST	Letzter Wert in der entsprechenden Spalte
MAXIMUM	Größter Wert in der entsprechenden Spalte
MINIMUM	Kleinster Wert in der entsprechenden Spalte

Viel mehr ist für die Einrichtung von Gruppensummen eigentlich nicht erforderlich:

```
                                                  Bestands    Bestands
Überschrift für Gruppensumme                Lager    menge        Wert
   ----------  --------------------  -----  ----------  ----------
   00001       Mauspad Gummi         25             4        8,00
   00001       Mauspad Gummi         10             4        8,00
                                             ----------  ----------
Schlusstext für Gruppensumme                  *        8       16,00
```

Fehlende Gruppentexte

Wir benötigen jetzt nur noch die Seitenüberschrift und die Anfangs- und Schlusstexte für die Gruppensummen. Für die Seitenüberschrift rufen Sie wieder den Menüpunkt „Seitenüberschrift editieren“ auf. Hier finden wir aber kaum etwas Neues. Neu sind lediglich die Gruppenwechseltexte:

```
                         Gruppenwechseltext editieren

 Auswahl eingeben und Eingabetaste drücken.
   1=Überschriftstext   2=Fußtext

            Gruppen-
 Ausw      wechselebene      Spalten mit dieser Gruppenwechselstufe
  1            1              ARTIKELNUMMER
```

Gruppenwechseltext editieren

Für jede hinterlegte Gruppenwechselstufe können Sie eine Überschrift und einen separaten Schlusstext hinterlegen. Beginnen wir mit der Überschrift:

```
                    Gruppenwechselüberschrift editieren

Gruppenwechselstufe  . . . . . :   1

Auswahl eingeben, Eingabetaste drücken
  Auf neue Seite stellen . . . . . . . . . .   J          J=Ja, N=Nein
  Spaltenüberschrift wiederholen . . . . . .   J          J=Ja, N=Nein
  Anzahl Leerzeilen davor  . . . . . . . . .   1__        0-999
  Anzahl Leerzeilen danach . . . . . . . . .   1__        0-999

Gruppenwechseltext: &col für Variableneinfügung verwenden.
                                                Spaltennamen könnten im Text integriert werden, indem die
                                                Spaltennummer mit vorangestelltem & verwendet wird.
Zeil  Ausrichten        Gruppenwechselüberschriftstext
  1   LINKS             Bestandswert:
```

Gruppenwechselüberschrift editieren

Ergänzung 29/2014

7.5.5
Seite 34

Vieles ist in dem Dialog selbsterklärend: Sie legen fest, ob jede Gruppe auf einer separaten Seite gedruckt werden soll und ob die hinterlegten Spaltenüberschriften für jede Gruppe wiederholt werden sollen. Es folgt die Angabe der Leerzeilen vor und nach Ausgabe der Überschrift und letztendlich die Überschrift selbst. Im Überschriftentext können Sie jederzeit Spaltennamen des Result Sets integrieren. Bei den Gruppenwechselschlusstexten können Sie sehen, wie das geht.

```
                        Gruppenwechselfuß editieren

Gruppenwechselstufe  . . . . . :   1

Auswahl eingeben, Eingabetaste drücken
  Auf neue Seite stellen . . . . . .  N          J=Ja, N=Nein
  Anzahl Leerzeilen davor  . . . . .  0          0-999, BOTTOM
  Anzahl Leerzeilen danach . . . . .  0          0-999
  Gruppenwechselergebnis in Zeile stellen  1     1-999, NONE

Gruppenwechseltext: &col für Variableneinfügung verwenden.

Zeil  Ausrichten       Gruppenwechselfußtext
  1   LINKS            Bestandswert (&3):
```

Gruppenwechselfuß editieren

Im Gruppenwechselfuß soll die Artikelbezeichnung integriert werden. Dafür verwende ich den Platzhalter &3.

Zum Schluss müssen wir wieder die Schlusstexte editieren, um den Schlussbestand einmalig am Ende des Berichts auszugeben:

```
                          Schlusstext editieren

Auswahl eingeben, Eingabetaste drücken
  Neue Seite für Schlußtext  . . . . . .  N          J=Ja, N=Nein
  Anzahl Leerzeilen davor  . . . . . .  2          0-999, BOTTOM
  Zeile für Endergebnis  . . . . . .  1            1-999, NONE

Schlußtext: &col für Variableneinfügung ver
Gesamtwerte könnten unterdrückt werden,
wenn sie nicht benötigt werden.

Zeil  Ausrichten       Schlußtext
  1   RECHTS           Der Bestandswert(Gesamt):
```

Schlusstext editieren

Jetzt sind alle Angaben hinterlegt und unser Bericht kann erneut getestet werden:

```
                                          Bestands    Bestands
                                 Lager     menge        Wert
  -----  ------------------  -----  ----------  ----------

Bestandswert:
                                 Auswahlmöglichkeiten zum Formatieren angeben
                                 _
  001    Mauspad Gummi         25              4        8,00
  001    Mauspad Gummi         10              4        8,00
                                        ----------  ----------
Bestandswert (Mauspad Gummi):                   8       16,00
```

Der Beispielbericht

Mich stören zurzeit nur noch die Trennzeichen unterhalb der Spaltenüberschriften. Leider können wir keine individuellen Trennzeichen vereinbaren, aber wir können auf die Trennzeichen verzichten, wenn wir die entsprechenden Formatoptionen („Auswahlmöglichkeiten zum Formatieren angeben“) auswählen. Außerdem empfinde ich die Wiederholungen der Artikelbezeichnungen störend und korrigiere auch das über „Auswahlmöglichkeiten zum Formatieren angeben“:

```
                Auswahlmöglichkeiten zum Formatieren des Berichts

Auswahl eingeben, Eingabetaste drücken

  Einzelzeilenabstand  . . . . . . . . .   1      1-4

  Entwurf für Gruppenwechselspalten  . .   J      J=Ja, N=Nein

  Standardgruppenwechseltext . . . . . .   N      J=Ja, N=Nein

  Zeilen mit Spaltenüberlauf auf Seite
    zusammenhalten . . . . . . . . . . .   J      J=Ja, N=Nein

  Trennzeichen für Spaltenüberschrift      N      J=Ja, N=Nein

  Trennzeichen für
    Gruppenwechselergebnis . . . . . . .   J      J=Ja, N=Nein

  Trennzeichen für Endergebnis . . . . .   N      J=Ja, N=Nein


F2=Alternativt.    F3=Verl.   F5=Bericht anz.   F12=Abbr.   F13=Abfrage edi
F18=SQL anzeigen   F22=QM-Anweisung
```

Der Menüpunkt „Auswahlmöglichkeiten zum Formatieren angeben“

Sind alle Angaben hinterlegt, verlassen Sie den QM und führen den Report aus. Denken Sie daran, dass es unnötig ist, dem Bericht Datum, Uhrzeit und Seitenzahlen hinzuzufügen. Schließlich haben wir diese Informationen bereits im Berichtsformular hinterlegt.

```
                  Abfrageverwalt.abfrage starten (STRQMQRY)

Auswahl eingeben und Eingabetaste drücken.

Abfrageverwaltungsabfrage  . . . > BST_ART_LG    Name
  Bibliothek . . . . . . . . . .   EPSW07        Name, *LIBL, *CURLIB
Ausgabe  . . . . . . . . . . . . > *PRINT        *, *PRINT, *OUTFILE
Abfrageverw.-Berichtsformular  .   *SYSDFT       Name, *SYSDFT, *QMQRY
  Bibliothek . . . . . . . . . .                 Name, *LIBL, *CURLIB
Datum und Zeit . . . . . . . . . > *NO           *YES, *NO
Seitenzahlen . . . . . . . . . . > *NO           *YES, *NO
```

Berichtsabfrage ausführen (STRQMQRY)

7.5.5.6 Query Manager-Abfragen parametrisieren

Der Bericht aus dem vorangegangenen Kapitel läuft problemlos. Doch es wird nicht lange dauern bis der Wunsch entsteht, nur einzelne Läger oder einzelne Artikel abzufragen. Was sollen wir tun? Für jede individuelle Anfrage eine eigene Abfrage erstellen? Bitte nicht! Sehr schnell herrscht eine unüberschaubare Flut von Query-Abfragen und der Überblick geht verloren. Dieses Szenario kennen Sie vermutlich von Ihren Query/400-Abfragen.

QM-Abfragen können wir sehr einfach parametrisieren. Dazu bauen wir zunächst einmal die SQL-Abfrage ein wenig um:

```
*************************** Anfang der Daten ***************************
 With Bestand
 as
(SELECT Firma, Artikelnummer, Lagerort, SUM(Bestand) As Bestandsmenge,
 SUM(Bestand * ArtikelPreis) AS Bestandswert
 FROM artikel JOIN lagerbestand USING(Firma, Artikelnummer)
 GROUP BY Firma, Artikelnummer, Lagerort
 )
 Select Firma, Artikelnummer, Bezeichnung, Lagerort, Bestandsmenge,
 Bestandswert
 from Artikel Join Bestand using(Firma, Artikelnummer)
 where Firma between  &VFIRMA1 and &VFIRMA2
 and Artikelnummer between  &VARTNR1 and &VARTNR2
**************************** Ende der Daten ****************************
```

Die modifizierte Abfrage

Sie sehen, dass ich die Abfrage um eine WHERE-Klausel erweitert habe. Gleichzeitig habe ich an Stelle der Konstanten Platzhalter verwendet – im Beispiel die Variablen &VFIRMA und &VARTNR. Sie sollten darauf achten, die Variablennamen in Großbuchstaben zu schreiben, da der Variablenname später case sensitive ist, ersparen Sie sich auf diesem Weg unnötigen Ärger. Des Weiteren muss jedem Platzhalter zwingend ein &-Zeichen vorangestellt werden.

Nun testen wir unsere Abfrage:

```
Einen Wert für Variable VFIRMA1 eingeben und die Eingabetaste drücken.

Antwort eingeben und Eingabetaste drücken.
  Antwort . .   _____________________________________________________________
```

Werte für Variablen angeben

Anders als zuvor, erhalten wir jetzt einen Abfragebildschirm, um die eben hinterlegten Variablen zu setzen. Als Erstes wird ein Wert für die Variable VFIRMA1 angefordert. Anschließend werden die weiteren Variablen angefordert. Natürlich wollen wir diese Variablen nicht interaktiv setzen.

Zur Lösung dieser Aufgabe ist es am einfachsten, einen Bildschirm in einem CL-Programm zu verwenden. Der Anwender gibt dort die entsprechenden Werte ein, bevor die Abfrage ausgeführt wird:

```
ITRAINNG                                                    3.03.14
                                                            RASCHE

 Bestandsliste drucken:

                    Firma              __   CL-Variable: &C_FIRMA
                    Artikelnummer      _____ CL-Variable: &C_ARTNR

 Soll eine Gesamtbestandsliste gedruckt werden,
 lassen Sie die Felder leer!

 F3 Programm          F19 Befeile
```

Die Oberfläche

7.5.5
Seite 38

Das CL-Programm ist schnell geschrieben, wenn man ein paar einfache Regeln beachtet:

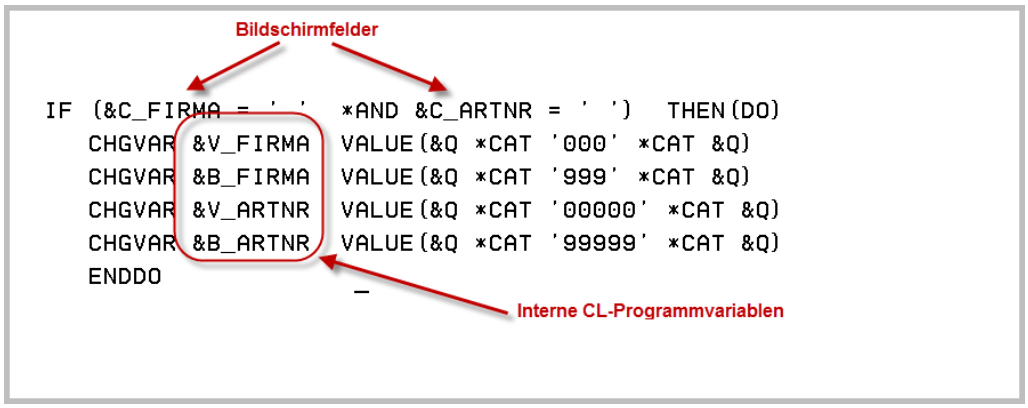
Alle alphanumerischen Werte müssen explizit in Hochkommata eingeschlossen sein, wenn der Wert an das Query übergeben wird. Aus diesem Grund arbeite ich in allen Programmen, die direkt oder indirekt SQL verwenden, stets mit einer konstanten Variable, die ein Hochkomma enthält. Im Beispiel-CL-Programm wurde die Variable &Q folgendermaßen deklariert:

```
DCL &Q *CHAR 1 VALUE(''''')
```

Da die alphanumerischen Übergabewerte Hochkommata enthalten, müssen Sie die Feldlängen der CL-Variablen entsprechend anpassen:

```
DCL &V_ARTNR *CHAR 7  -- CHAR(5) in der Datenbank
DCL &B_ARTNR *CHAR 7  -- CHAR(5) in der Datenbank
DCL &V_FIRMA *CHAR 5  -- CHAR(3) in der Datenbank
DCL &B_FIRMA *CHAR 5  -- CHAR(3) in der Datenbank
```

Jetzt gilt es nur noch entsprechende Bedingungen abzuprüfen. Hat der Anwender die Bildschirmfelder leer gelassen und möchte eine Gesamtbestandsauswertung?



Bedingungen auswerten I

Oder hat der Anwender im Bildschirm eine selektive Bestandsauswertung angefordert?

```
IF (&C_FIRMA *NE ' '  *AND &C_ARTNR *NE ' ')  THEN(DO)
   CHGVAR &V_FIRMA  VALUE(&Q *CAT &C_FIRMA *CAT &Q)
   CHGVAR &B_FIRMA  VALUE(&Q *CAT &C_FIRMA *CAT &Q)
   CHGVAR &V_ARTNR  VALUE(&Q *CAT &C_ARTNR *CAT &Q)
   CHGVAR &B_ARTNR  VALUE(&Q *CAT &C_ARTNR *CAT &Q)
   ENDDO
```

Bedingungen auswerten II

Der Aufruf des Querys ist im Beispielprogramm in beiden Fällen identisch:

```
STRQMQRY    QMQRY(BST_ART_L2) OUTPUT(*)   +
            SETVAR((VFIRMA1 &V_FIRMA)     +
                   (VFIRMA2 &B_FIRMA)     +
                   (VARTNR1 &V_ARTNR)     +
                   (VARTNR2 &B_ARTNR))
```

SQL-Variable

CL Programmvariable

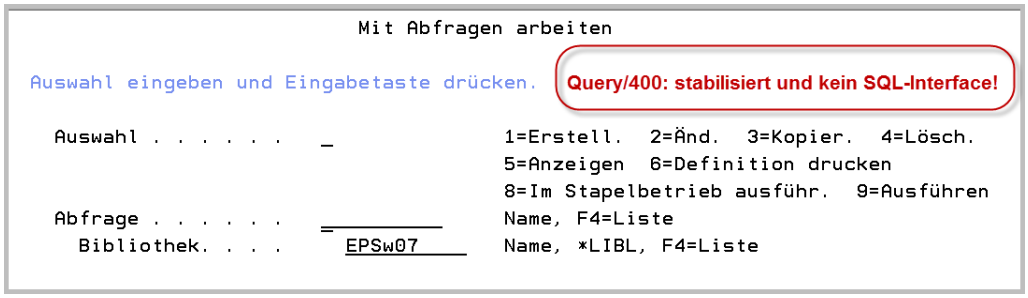
Aufruf der Query Manager-Abfrage

Beachten müssen Sie hierbei lediglich die Konventionen für die Namen der Query-Variablen:

- Der Query-Variablenname wird im CL-Programm ohne vorangestelltes &-Zeichen verwendet.
- Der Query-Variablenname ist case sensitive. Sie müssen also auf die Groß- und Kleinschreibung achten. Daher die Empfehlung, für die Variablennamen im Query grundsätzlich Großbuchstaben zu verwenden.

7.5.5.7 Query/400 (*QMQRY) konvertieren

Einige Leser werden sich vielleicht fragen, warum ich dem QM in diesem Fachbuch so viel Raum gebe? Die Antwort ist ganz einfach: Viele Auswertungen existieren in den Firmen häufig als Query/400-Auswertungen. Query/400 findet man fast auf jedem System i-Server. Aufgerufen wird das Produkt mit dem Befehl GO QUERY oder WORK QUERY (WRKQRY):



Der Befehl WRKQRY

Was ist verkehrt an den Query/400-Abfragen?

- Zunächst einmal: Query/400 wird seitens IBM nicht weiterentwickelt. Das Tool ist stabilisiert und verschwindet eines Tages vielleicht ganz. IBM garantiert zur Zeit lediglich, dass alte Query/400-Abfragen weiterhin ausgeführt werden können. Eine Garantie für weitere Modifikationen gibt es nicht!
- Query/400 ist kein SQL-Interface und unterstützt somit auch keine SQL-Abfragen. Query/400-Abfragen werden über eine alte Datenbankschnittstelle ausgeführt. Daher sind die Abfragen auch vergleichsweise langsam und belasten in nicht unerheblichem Maß die Systemperformance. Und eines ist gewiss: Das wird sich im Laufe der Zeit auch nicht ändern.
- Hinzu kommt, dass die QM-Abfragen sehr viel flexibler eingesetzt werden können, wenn Sie Variablen in Ihren Abfragedefinitionen verwenden – wie im vorangegangenen Kapitel gezeigt.

Viele Firmen kennen das Problem und sind vielleicht sogar bereit, zum QM zu wechseln. Doch es bleibt die Frage zu klären: „Wer hat die notwendige Zeit, die vielen existierenden Abfragen neu zu erstellen?“ Und so scheitert der Wechsel oftmals und alles bleibt beim Alten. Dabei ist es einfach, Query/400-Abfragen zu konvertieren. Ich möchte Ihnen daher in diesem Kapitel zeigen, wie Sie mit überschaubarem Aufwand ihre Query/400-Abfragen konvertieren können.

Schauen wir uns zunächst die Query/400-Abfrage an:

```
AUFTRAGSNUMMER  RABATT   ARTIKELNUMMER   MENGE              PosWert
    00043        3,20       00005         100             1.500,00
                 3,20       00003         100               150,00
                 3,20       44551         100                50,00
                 3,20       00002          30                36,00
                                         Endsumme 00043
                                         SUMME             1.736,00

    00143        3,20       66111         100               399,00
                 3,20       00001          10                20,00
                                         Endsumme 00143
                                         SUMME               419,00
```

Der Query/400-Report

Wenn wir eine bestehende Query/400-Abfrage konvertieren wollen, benötigen wir zunächst einmal eine Quellendatei zum Abrufen der Query/400-Informationen. Ich nenne meine Quellendatei QRY400SRC und erstelle sie in der Bibliothek EPSW07:

```
CRTSRCPF FILE(EPSW07/QRY400SRC) RCDLEN(92) TEXT('Query/400-
Abfragen')
```

Anschließend importiere ich in einem ersten Schritt die Query/400-Abfragedefinition:

```
                  Abfrageverw.-Abfrage auffind.  (RTVQMQRY)

Auswahl eingeben und Eingabetaste drücken.

Abfrageverwaltungsabfrage  . . . > ART_BST      Name
  Bibliothek . . . . . . . . . . >   EPSW07      Name, *LIBL, *CURLIB
Quellendatei . . . . . . . . . . > QRY400SRC    Name
  Bibliothek . . . . . . . . . .     *LIBL      Name, *LIBL, *CURLIB
Quellenteildatei . . . . . . . . > ART_BST_D    Name der Teildatei
Informat. von QRYDFN zulassen      *YES         *NO, *YES, *ONLY
```

Query/400-Name

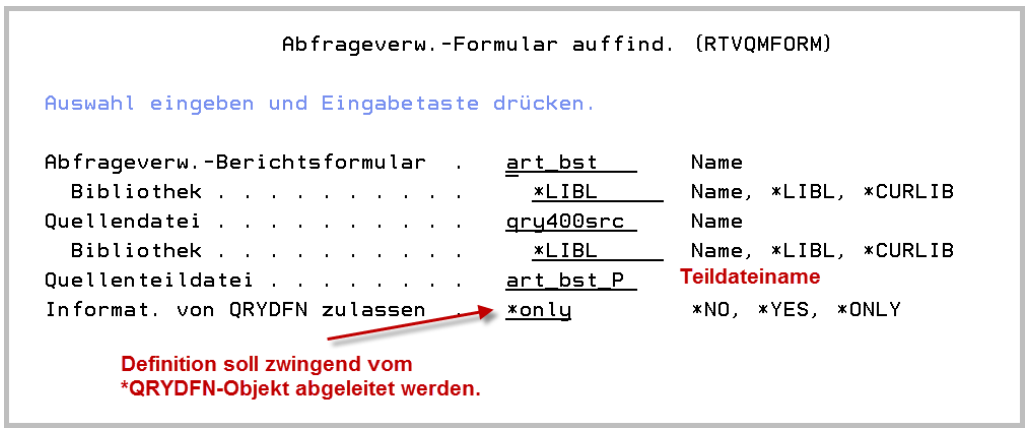
Abruf einer Query/400-Definition

Der CL-Befehl RTVQMQRY

Mit dem Befehl RTVQMQRY („Abfrageverwaltungsabfrage auffinden“) wird eigentlich eine SQL-Quelle aus einem QM-Objekt abgerufen. Doch wir können den CL-Befehl auch verwenden, um die Query/400-Definition zu generieren. Dafür muss der letzte Parameter „Informat. von QRYDFN zulassen“ auf *YES gesetzt sein.

7.5.5
Seite 42

Nun haben wir es aber im Beispiel mit einem Query/400-Objekt zu tun und nicht mit QM-Objekten. Daher müssen wir daran denken, dass wir auch noch die Formulardefinition benötigen. Dafür verwenden wir allerdings den CL-Befehl RTVQMFORM (Retrieve Query Manager Formular):



Der CL-Befehl RTVQMFORM

Der Befehl ist identisch zum vorherigen Befehl aufgebaut, erstellt allerdings das Berichtsformular. Beachten müssen Sie hierbei lediglich, dass Sie einen neuen Teildateinamen verwenden, wenn die Definitionen in der gleichen Quellendatei abgespeichert werden. Außerdem empfiehlt es sich, den letzten Parameter „Informat. von QRYDFN zulassen“ auf den Wert *ONLY zu setzen, damit ausschließlich QRYDFN-Objekte beim Informationsabruf verwendet werden.

Wenn alles funktioniert hat, müssten Sie jetzt zwei Teildateien in der Quellendatei QRY400SRC finden. Die erste Teildatei enthält die SQL-Abfrage.

```
****************** Datenanfang ********************************
H QM4 05 Q 01 E V W E R 01 03 14/03/03 16:51
V 1001 050 Objektkonvertierung                              LBW
V 5001 004 *HEX
SELECT
  ALL       T01.AUFNR, T01.RABATT, T02.ARTNR, T02.MENGE,
            ((T02.MENGE*T02.ARTPREIS))
  FROM      EPSW07/AUFTRAG T01 INNER JOIN
            EPSW07/AUFTR00001 T02
  ON        T01.AUFNR = T02.AUFNR INNER JOIN
            EPSW07/ARTIKEL T03
  ON        T02.ARTNR = T03.ARTNR
  WHERE     T01.AUFDAT = '2010-07-12'
  ORDER BY  001 ASC
```

Inhalt der Teildatei ART_BST_D

Und die zweite Teildatei beinhaltet die Berichtsdefinition.

```
H QM4 05 F 01 E V W E R 01 03 14/03/03 17:02
V 1001 050 Objektkonvertierung                              LBW
T 1110 005 006 1114 007 1115 006 1116 005 1117 005 1118 003 1113 062
R BREAK1  0                   1
R         2                   2
R         2                   3
R         2                   4   MENGE
R SUM     2      21    K02    5   SUMME_PosWert
V 1201 001 0
V 1202 001 2
V 1301 001 2
V 1302 001 0
V 1401 002 NO
V 1402 004 2
V 1403 006 0
T 1410 001 003 1412 004 1413 006 1414 055
```

Die Berichtsdefinition – Teildatei ART_BST_P

Sie können die Teildateien ruhig öffnen, wie ich es getan habe. Aber keine Sorge, es ist nicht notwendig, den Quellcode genauer zu untersuchen oder gar zu modifizieren.

Wir erstellen jetzt mit Hilfe dieser Teildateien zwei neue QM-Objekte. Als Erstes kommt die Abfrage dran. Dafür nutzt man den Befehl CRTQMQRY (Create Query Manager Query):

```
                  Abfrageverwalt.-Abfrage erst.  (CRTQMQRY)

Auswahl eingeben und Eingabetaste drücken.

Abfrageverwaltungsabfrage  . . . > ART_BST      Name
  Bibliothek . . . . . . . . . . >   EPSW07      Name, *CURLIB
Quellendatei . . . . . . . . . . > QRY400SRC    Name
  Bibliothek . . . . . . . . . .     *LIBL      Name, *LIBL, *CURLIB
Quellenteildatei . . . . . . . . > ART_BST_D    Name, *QMQRY
Text 'Beschreibung'  . . . . . . > 'Konvertiert: Auftragsstatus (Query/400)'

Sortierfolge . . . . . . . . . .   *SRC         Name, *SRC, *JOBRUN, *JOB...
  Bibliothek . . . . . . . . . .     *LIBL      Name, *LIBL, *CURLIB
Sprachen-ID  . . . . . . . . . .   *SRC         *JOB, *JOBRUN...

                Zusätzliche Parameter

Berechtigung . . . . . . . . . .   *LIBCRTAUT   Name, *USE, *CHANGE, *ALL...
Objekt ersetzen  . . . . . . . .   *YES         *YES, *NO

                                                                      Ende
F9=Alle Parameter   F11=Schlüsselwörter   F14=Befehlszeichenfolge
F24=Weitere Tasten
```

Name der Query Manager-Abfrage

Query Manager-Abfrage erstellen

7.5.5
Seite 44

Mit dem Befehl CRTQMQRY wird eine QM-Abfrage mit einem Standardformular erstellt. In einem zweiten Schritt muss anschließend das zugehörige Formular generiert werden:

```
                    Abfrageverwalt.formular erst.  (CRTQMFORM)

Auswahl eingeben und Eingabetaste drücken.

Abfrageverw.-Berichtsformular  . > BST_ART_P      Name
  Bibliothek . . . . . .           *CURLIB       Name, *CURLIB
Quellendatei . . . . . . . . . > QRY400SRC      Name
  Bibliothek . . . . . .           *LIBL         Name, *LIBL, *CURLIB
Quellenteildatei . . . . . .     *QMFORM       Name, *QMFORM
Text 'Beschreibung'  . . . . . > 'Konvertiert Auftragsstatus'
______

                 Zusätzliche Parameter

Berechtigung . . . . . .     *LIBCRTAUT    Name, *USE, *CHANGE, *ALL...
Objekt ersetzen  . . . . . .     *YES          *YES, *NO
```

Query Manager-Berichtsformular erstellen

Mit dem Befehl CRTQMFORM kann ein Abfrageverwaltungsformular erstellt werden. Wir wissen, dass das Formular das Format eines Berichts definiert, der nach Ausführung der Abfrage angezeigt oder gedruckt wird. Eventuell erhalten Sie Warnungen bei der Formularerstellung. Das kann beispielsweise daran liegen, dass im Query/400 Editiercodes verwendet wurden, die dem QM unbekannt sind. Solange es sich um Warnungen handelt, können Sie diese fürs Erste ignorieren.

Sie müssten jetzt drei Objekte in Ihrer Zielbibliothek haben:

```
                              Mit Objekten arbeiten

  Auswahl eingeben und Eingabetaste drücken.
    2=Berechtigung editieren   3=Kopieren   4=Löschen   5=Berechtigung anzeigen
    7=Umbenennen   8=Beschreibung anzeigen   13=Beschreibung ändern

  Opt  Objekt      Art       Bibliothek  Attribut    Text
  __   ART_BST     *QRYDFN   EPSW07      QRY         Objektkonvertierung
  __   ART_BST     *QMQRY    EPSW07      SQL         Konvertiert: Auftragsstatus
  __   ART_BST     *QMFORM   EPSW07      QUERY MGR   Konvertiert Auftragsstatus
```

Das Ergebnis der Konvertierung – Objektstatus

Achten Sie darauf, dass die *QMQRY-Abfrage denselben Namen hat wie das Berichtsformular (*QMFORM). Ist das nicht der Fall, müssen Sie den Erstellungsvorgang wiederholen.

Jetzt starten wir wie gehabt den QM. Sie sollten auch hier sowohl Ihre eben erstellte Abfrage als auch das zugehörige Formular in der Übersicht finden:

```
                Mit Query Manager Berichtsformularen arbeiten

Bibliothek . . . . . .   EPSW07     Name, F4=Liste

Auswahl eingeben und Eingabetaste drücken.
  1=Erstellen    2=Ändern    3=Kopieren    4=Löschen    5=Anzeigen    6=Drucken
  7=Umbenennen   9=Bericht ausführen

Ausw.  Formular    Beschreibung
  =    __________
  _    ART_BST     Konvertiert Auftragsstatus
```

```
                    Mit Query Manager Abfragen arbeiten

Bibliothek . . . . . .   EPSW07     Name, F4=Liste
Abfrageerstellungsmodus  . . :   PROMPT

Auswahl eingeben und Eingabetaste drücken.
  1=Erstellen    2=Ändern    3=Kopieren    4=Löschen    5=Anzeigen    6=Drucken
  7=Umbenennen   9=Ausführen   10=In SQL umsetzen

Ausw.  Abfrage     Art       Beschreibung
 =_    __________
 __    ART_BST     SQL       Konvertiert: Auftragsstatus (Query/400)
```

Das Ergebnis der Konvertierung – Query Manager

Beides – Abfrage und Formular – können Sie jetzt wie gewohnt im QM bearbeiten. Sie müssen es aber nicht! In vielen Fällen ist eine Nachbearbeitung nicht notwendig, und die Abfragen können sofort mit dem Befehl STRQMQRY ausgeführt werden. Sie müssen lediglich darauf achten, dass das richtige Berichtsformular verwendet wird.

```
                Abfrageverwalt.abfrage starten (STRQMQRY)

  Auswahl eingeben und Eingabetaste drücken.

  Abfrageverwaltungsabfrage  . . . > ART_BST     Name
    Bibliothek . . . . . .     *LIBL     Name, *LIBL, *CURLIB
  Ausgabe . . . . . .          *        *, *PRINT, *OUTFILE
  Abfrageverw.-Berichtsformular  .   *QMQRY     Name, *SYSDFT, *QMQRY
    Bibliothek . . . . . .           __________  Name, *LIBL, *CURLIB
```

Ausführung der konvertierten Query/400-Abfrage

So einfach kann die Konvertierung einer Query/400-Abfrage sein!

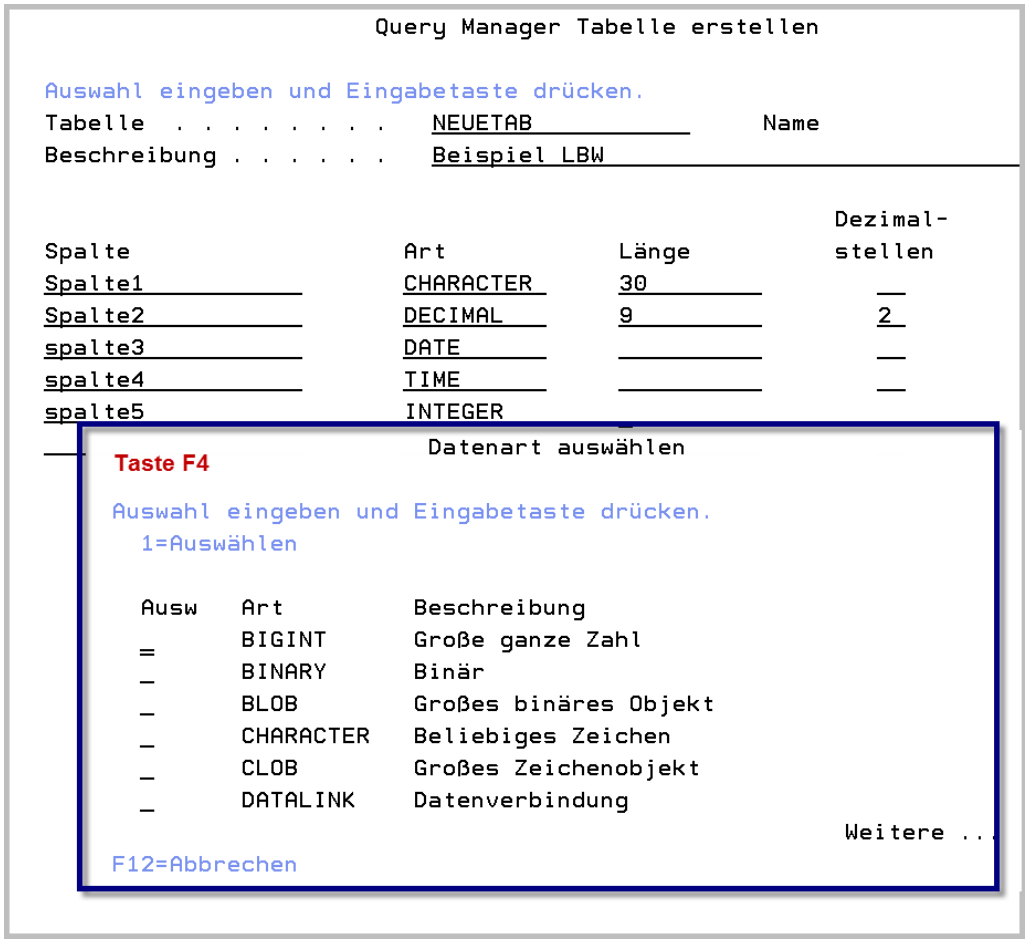
7.5.5.8 Query Manager – Mit Tabellen arbeiten

Der Vollständigkeit halber wollen wir uns abschließend auch noch den Menüpunkt 3 „Mit Query Manager Tabellen arbeiten“ anschauen. Jede Datenbankdatei kann über den QM auch gepflegt, also modifiziert werden, vorausgesetzt, Sie haben die entsprechenden Zugriffsberechtigungen auf das Objekt. Über Menüpunkt 3 und nachdem Sie die entsprechende Arbeitsumgebung angegeben haben, gelangen Sie in dieselbe.

```
                    Mit Query Manager Tabellen arbeiten

Bibliothek . . . . . .   EPSW07      Name, F4=Liste

Auswahl eingeben und Eingabetaste drücken.
  1=Tabelle erstellen   3=Tabelle kopieren   4=Tabelle löschen
  5=Tabelle anz.        6=Tabelle drucken    7=Tabelle umbenennen
  8=Definition anz.     9=Daten hinzufügen   10=Daten ändern   11=Daten anz.

Ausw.  Tabelle               Beschreibung
 _     ____________________
 _     ACT
 _     ACTION
 _     ARTIKEL
```

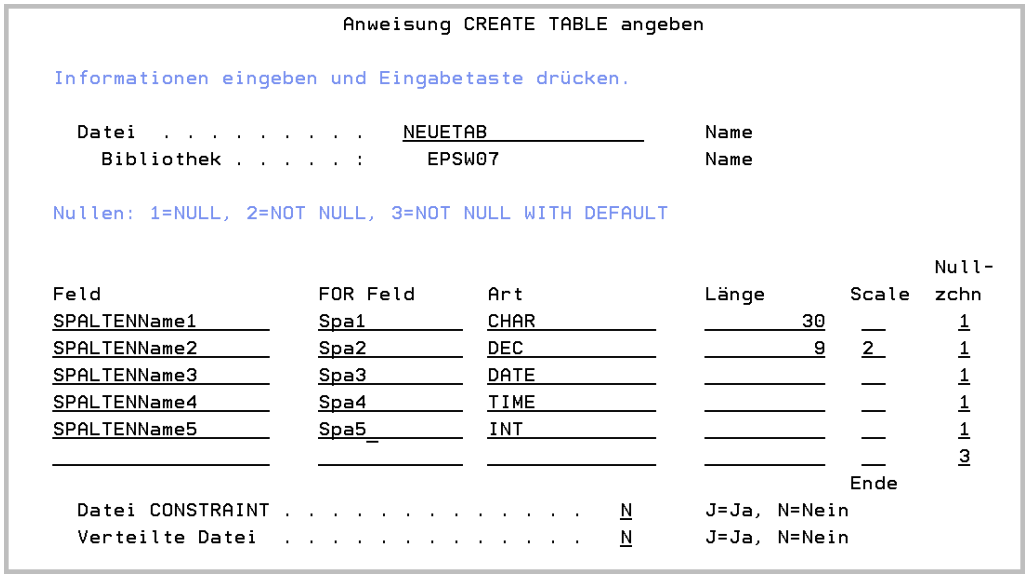
Mit Query Manager Tabellen arbeiten

Mit der Auswahl 1 „Tabelle erstellen“ ist es sehr einfach, neue Tabellen zu definieren. Sie müssen in dem Eingabebildschirm lediglich darauf achten, die Freigabetaste erst dann zu nutzen, wenn Sie alle Spaltendefinitionen eingegeben haben. Am einfachsten bewegen Sie sich in der Maske mit der Tabulatortaste von Feld zu Feld. Angenehm ist auch hier wieder die Taste F4, mit der Sie sich alle Datentypen auflisten können. Leider ist es in diesem Interface nicht möglich, versteckte Zeitstempel, die sich automatisch aktualisieren, oder Autoinkrement-Felder zu definieren.



Query Manager – Tabellen erstellen

Sie können aber mit der Funktionstaste F21 „Anweisung CREATE TABLE angeben“ das Eingabe-Interface ändern:

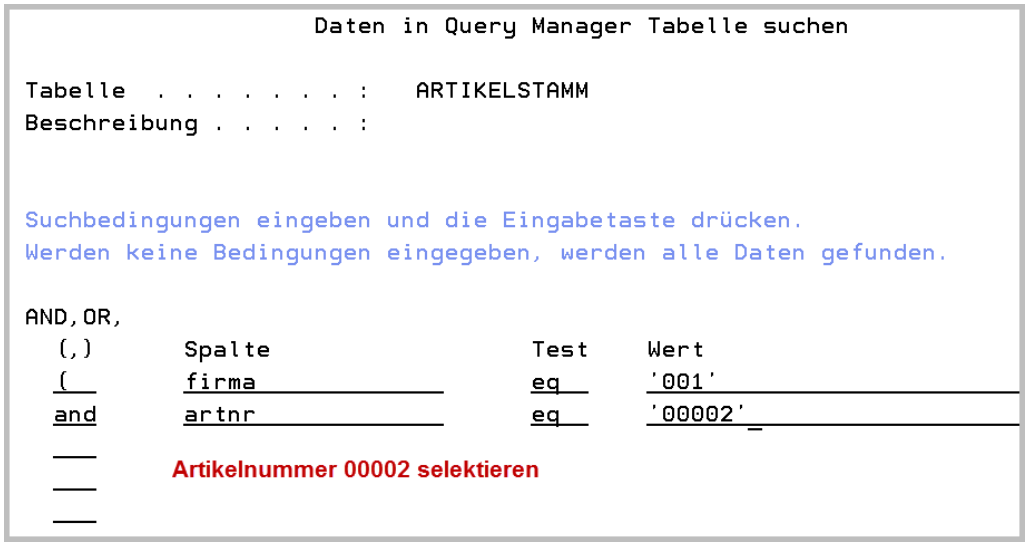


F21 „Anweisung CREATE TABLE angeben“

7.5.5
Seite 48

Hier können Sie zumindest den internen Spalten-Systemnamen angeben und festlegen, ob die Spalte NULL-Werte enthalten darf. Weitere Möglichkeiten für die Tabellendefinition haben Sie, wenn Sie in diesem Bildschirm die Funktionstaste F11 „Weitere Attribute anzeigen“ verwenden. Auch das SQL-Statement können Sie anzeigen. Dafür nutzen Sie die Funktionstaste F21 „Anweisung anzeigen“. Leider kann das SQL-Statement aber nicht in einer Quellendatei gespeichert werden. Abschließend bestätigen Sie Ihre Eingaben, um die Datei zu erstellen.

Sie können auch bestehende Dateien manipulieren, das heißt Datensätze ändern oder löschen. Dazu verwenden Sie Option 10. Es erscheint ein Dialog, der es Ihnen ermöglicht, Bedingungen einzugeben, um bestimmte Zeilen der Datei zu selektieren.



Daten ändern – die Selektion

Erst danach erhalten Sie die ausgewählten Zeilen zur Änderung. Wenn Sie keine Suchkriterien eingeben, wird der vollständige Datenbestand dargestellt.

```
                    Daten in Query Manager Tabelle ändern

Tabelle  . . :   ARTIKEL                 Aktuel. Zeile:   1
Beschreibung :

Angaben bis (]) eingeben und die Eingabetaste drücken.

Spalte                    Wert
FIRMA                [  001]
ARTIKELNUMMER        [  00002]
BEZEICHNUNG          [  Radiergummi(10)    ]
WARENGRUPPE          [  0003]
MINDESTBESTAND       [  40      ]
AUSLAUFKZ            [  N]
BESCHAFFUNG          [  L]
MENGENEINHEIT        [  Pckg ]
ARTIKELPREIS         [  1,20    ]
ANLAGEDATUM          [  19990229  ]


                                                              Ende
F3=Verlassen   F5=Aktualisieren   F11=Art/Länge anzeigen   F12=Abbrechen
F14=Vorherige Datenzeile          F15=Nächste Datenzeile   F24=Weitere Tasten
```

Datensätze ändern

In diesem Arbeitsbildschirm überschreiben Sie die Datenbankinhalte und bestätigen die Eingabe durch die Datenfreigabetaste. Den nächsten Datensatz können Sie bearbeiten, wenn Sie die Taste F15 drücken. Wünschen Sie in den vorherigen Satz zurückzukehren, betätigen Sie die Taste F14.

7.5.5
Seite 50

Erwähnenswert ist vielleicht noch die Löschfunktion. Um eine Zeile in der Tabelle zu löschen, müssen Sie die Funktionstaste F23 im Änderungsdialog benutzen:

```
                    Daten in Query Manager Tabelle ändern

Tabelle  . . :   ARTIKEL                 Aktuel. Zeile:   1
Beschreibung :                           Funktionstaste F23 - Datensatz löschen

Angaben bis (]) eingeben und die Eingabetaste drücken.

Spalte                    Wert
FIRMA                [  001]
ARTIKELNUMMER        [  00002]
BEZEICHNUNG          [  Radiergummi(10)    ]
WARENGRUPPE          [  0003]
MINDESTBESTAND       [  40     ]
AUSLAUFKZ            [  N]
BESCHAFFUNG          [  L]
MENGENEINHEIT        [  Pckg ]
ARTIKELPREIS         [  1,20   ]
ANLAGEDATUM          [  19990229  ]


                                                                  Ende
F3=Verlassen   F5=Aktualisieren   F11=Art/Länge anzeigen   F12=Abbrechen
F14=Vorherige Datenzeile          F15=Nächste Datenzeile   F24=Weitere Tasten
F23 drücken, um die Löschanforderung für die angezeigte Zeile zu bestätigen.
```

Datensatz löschen

Jede Löschanforderung muss mit der Funktionstaste F23 bestätigt werden, bevor der Satz endgültig aus der Datei entfernt wird.

Ich denke, dies sollte als Einführung und Überblick zum QM-Menüpunkt 3 „Mit Query Manager Tabellen arbeiten“ ausreichen. Alles andere spricht für sich und muss meiner Meinung nach nicht weiter erläutert werden. Und wer weiß, vielleicht müssen Sie ja auch gar nicht mit dem Menüpunkt 3 arbeiten.

7.6 SQL-Data Manipulation Language (DML)

Anders als die Vorgängermaschinen stellen die aktuellen System-i-Server nicht nur eine der ausfallsichersten Serverfamilien auf dem Markt dar sondern ermöglichen zudem flexible Anwendungen, die sowohl in klassischen, als auch in modernen Sprachen entwickelt werden können. Heute gibt es eine Vielzahl von Methoden, um auf die Daten eines System-i-Servers zuzugreifen. Allerdings lässt sich seit einiger Zeit eine Tendenz hin zum SQL erkennen. Eine Sprache, die universell verwendbar ist und so auch ihren Teil zur Langlebigkeit einer Applikation beiträgt. Sie haben in den vorangegangenen Kapiteln gesehen, dass SQL eine Sprache ist mit der Datenbankobjekte erstellt werden können. Aber SQL ist auch eine Abfragesprache, mit der wir Daten selektieren und manipulieren können. So kommt es, dass wir im SQL-Umfeld zwischen DDL-Anweisungen – wie CREATE TABLE, CREATE VIEW, CREATE INDEX – einerseits und DML-Anweisungen – wie SELECT, INSERT, UPDATE und DELETE – andererseits unterscheiden. So komplex eine SQL-Abfrage auch aufgebaut werden kann, so einfach ist zunächst die Unterteilung in die vier DML-Grundanweisungen:

DML-Anweisung	Funktion
INSERT	Zeilen in einer Tabelle einfügen
UPDATE	Zeilen einer Tabelle ändern
DELETE	Zeilen einer Tabelle löschen
SELECT	Zeilen einer Tabelle lesen

Mit diesen vier SQL-Anweisungen und den verschiedenen Möglichkeiten des Einsatzes wollen wir uns im folgenden Kapitel beschäftigen.

Ergänzung 15/2010

7.6 Seite 2

7.6.1 Abfragen mit SELECT

Wie bereits zuvor beschrieben, gliedert sich die Datenmanipulationssprache DML von SQL in vier Hauptbereiche. Der wohl bekannteste Bereich dient dem Zugriff auf den Inhalt einer Tabelle in einer Datenbank. Damit ist zwar keine wirkliche Datenmanipulation verbunden – jedoch steht die Auswahl der Datensätze in einem direkten Zusammenhang mit den später durchzuführenden Manipulationen. Die Rede ist von der Anweisung SELECT. Bei SELECT handelt es sich um einen SQL-Grundbegriff, mit dem Daten aus einer oder mehreren Tabellen zur weiteren Verarbeitung ausgewählt werden können.

Es sind stets die folgenden Fragen, die mit den SELECT-Anweisungen beantwortet werden:

- Von wo will ich Daten abrufen bzw. ermitteln?
  In welcher Datenbank und in welcher Tabelle befinden sich die zu verarbeitenden Daten und Informationen?
- Welche Informationen werden benötigt?
  Hier geht es im Wesentlichen um die einzelnen Spalten, deren Inhalte ermittelt bzw. zurückgegeben werden sollen.
- Wie werden die Informationen benötigt?
  Die Reihenfolge der Ausgabe der ermittelten Daten ist besonders bei größeren Datenmengen nicht unwichtig.

Anmerkung

Es ist der Besonderheit von SQL und den damit verbundenen Technologien zu verdanken, dass man sich als Entwickler über den optimalen Zugriffsweg auf den Inhalt von Tabellen keine zu großen Gedanken machen muss. SQL ist in gewisser Weise „intelligent“ und ermittelt in vielen Fällen den idealen Zugriff auf die gewünschten Daten. SQE (SQL Query Engine) ist eine Funktion innerhalb der Datenbank des IBM i, mit der Abfragen aus SQL heraus an die Datenbank übertragen und dort optimiert ausgeführt werden. Details dazu finden Sie in einem separaten Kapitel.

Neben diesen Hauptfragestellungen bietet die SQL SELECT-Anweisung noch weitere Teilbereiche, mit denen sich die Ergebnisse beeinflussen und optimieren lassen. Ohne darauf in unseren ersten Beispielen im Detail eingehen zu wollen, sei an dieser Stelle bereits ein Überblick über die Möglichkeiten gestattet. Damit gewinnen Sie dann auch einen ersten Eindruck, was alles mit SELECT-Anweisungen möglich sein kann. Die SELECT-Syntax gestaltet sich folgendermaßen:

Ergänzung 15/2010

7.6.1
Seite 2

- SELECT *Ausdrucksliste*
  Über die Ausdrucksliste wird definiert, welche Spalten aus der Tabelle angezeigt werden sollen. Damit definieren wir die Spalten und die Ergebnisse für die Selektion.
- FROM *Tabelle Joinliste*
  In diesem Bereich wird die Tabelle (bzw. auch mehrere Tabellen) angegeben, deren Inhalt verarbeitet werden soll. Die einfachste SELECT-Anweisung verarbeitet den Inhalt einer einzelnen Tabelle. Wenn mehrere Tabellen verarbeitet werden sollen, dann müssen diese zueinander in Verbindung gebracht werden. Wir sprechen dann von einem Join.
- WHERE *Bedingungsliste* (Optional in der Verwendung)
  Bei der WHERE-Klausel handelt es sich um eine optionale Angabe. Diese kann, muss aber nicht, in der SELECT-Anweisung verwendet werden. Mittels der WHERE-Klausel definieren wir die aus der Tabelle zu selektierenden Datensätze. Damit lassen sich die Ergebnisse der SELECT-Anweisung auf den gewünschten Teilbereich beschränken.
- GROUP BY *Ausdrucksliste* (Optional in der Verwendung)
  Auch die GROUP BY-Klausel ist in der Verwendung optional. Wenn die Ergebnisdatensätze in Gruppen zusammengefasst werden sollen, verwenden wir diese Klausel.
- HAVING *Bedingungsliste* (Optional in der Verwendung)
  Eng mit der GROUP BY-Klausel ist die HAVING-Klausel verbunden. Mit der Verwendung von HAVING definieren wir die Bedingungen für die zu gruppierenden Datensätze.
- ORDER BY *Ausdrucksliste* (aufsteigend /absteigend)
  Die Sortierung des Ergebnisses lässt sich über die Anweisung ORDER BY steuern. Damit erreichen wir eine nach unseren Vorgaben strukturierte Sortierung der Ergebnisliste.

Anmerkung

Die Reihenfolge der Klauseln der SELECT-Anweisung ist unbedingt einzuhalten. So darf zum Beispiel die WHERE-Klausel nicht nach der GROUP-Klausel angegeben werden!

Mit diesem Wissen können wir unsere erste einfache SELECT-Anweisung erstellen.

Der grundsätzliche Aufbau mit den Mindestangaben der SELECT-Anweisung schaut wie folgt aus:

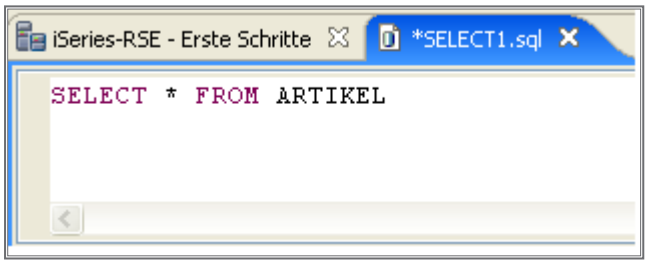
```
SELECT * FROM Tabellenname
```

Bei dieser Form der SELECT-Anweisung handelt es sich um die kürzest mögliche Variante.

Mit der Anweisung „SELECT“ teilen wir dem System mit, dass wir auf den Inhalt einer Datenbanktabelle zugreifen wollen.

Angewendet auf eine einfache Tabelle „ARTIKEL“ mit Artikelstammdaten sieht das Beispiel dann so aus.

```
SELECT * FROM ARTIKEL
```



SELECT

Das Ergebnis dieser Abfrage finden Sie in der folgenden Abbildung.

SELECT1.sql

Nachrichten | Parameter | Ergebnisse | Profilerstellungsdaten

FIRMA	ARTIKELNR	BEZEICHUNG	SORTIMENT	AUSLAUFKZ	SPERRKZ	BESCHAFF...	ANLAGEDA...
001	10001	Schnellhefter A4	Standard	N	N	K	08.08.09
001	10002	Büroklammern 1000 Stück ...	Standard	N	N	K	08.08.09
001	10003	Bleistift	Standard	N	N	K	08.08.09
001	10004	Ringbinder	Standard	N	N	K	08.08.09
001	10005	Radiergummi	Standard	N	N	K	01.07.09
001	10006	Locher	Standard	N	N	K	10.10.08
001	10008	Drucker Toner	Drucker	N	N	K	08.08.09
001	10009	Mauspad	Computer	N	N	K	01.08.09
001	10010	Post It Klein	Standard	N	N	K	01.08.09
001	10011	Post It Mittel	Standard	N	N	K	01.08.09
001	10012	Post It groß	Standard	N	N	K	01.08.09
100	10010	DIN A4 Druckerpapier weiß ...	Papier	N	N	E	01.08.09
001	10015	DIN A4 Druckerpapier weiß ...	Standard	N	N	E	01.08.09
001	20001	Schere	Standard	N	N	K	01.08.09
010	10011	Ladegerät Universal	Elektro	N	N	K	01.08.09
010	10020	Heftklammern 1000 Stück ...	Standard	N	N	E	01.08.09
010	10030	USB Anschlusskabel	IT	N	N	J	01.08.09
001	10030	Fachbuch SQL	Buecher	N	N	K	01.01.09
001	10013	Atlas 2008	Standard	J	J	X	01.01.08
001	20014	Atlas 2009	Standard	J	N	X	01.01.08
001	20015	Atlas 2010	Standard	N	N	K	01.01.09

Ergebnis der SELECT-Abfrage

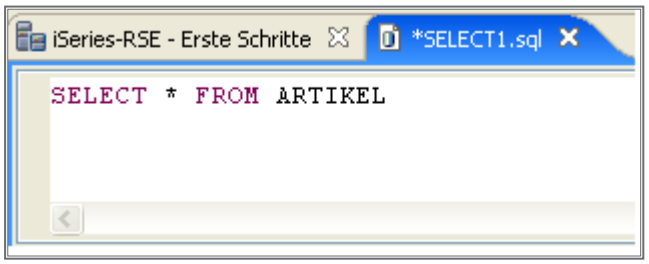
Als Ergebnis der SELECT-Abfrage erhalten wir eine Ausgabe des Inhalts der angegebenen Tabelle. In diesem Fall erhalten wir einen Überblick über die in dem Artikelstamm befindlichen Datensätze.

7.6.1
Seite 4

An dieser Stelle sei noch eine Besonderheit beim Einsatz von SQL mit IBM i erwähnt. Relationale Datenbanken und SQL kommen auf den unterschiedlichsten Systemen vor. Dort verfügt SQL dann je nach Betriebssystem und Version über unterschiedliche Ausprägungen, die nicht nur die Ergebnisse von SQL leicht variieren lassen können, sondern die auch bei der Erstellung von SQL-Anweisungen zu zum Teil unterschiedlichen Ergebnissen führen. SQL ist zwar ein Standard – leider wird dieser hier und da ein wenig großzügig ausgelegt.

Schauen wir uns nochmals die SELECT-Anweisung an, die zuvor eingegeben wurde:

```
SELECT * FROM ARTIKEL
```



SELECT

Ist Ihnen hier etwas aufgefallen?

Die Eingabe der SQL-Anweisung SELECT und der dazugehörigen Komponenten erfolgte in dem Beispiel in Großbuchstaben. Was passiert nun, wenn SELECT oder andere Bereiche der SQL-Anweisung in Kleinbuchstaben eingegeben werden?

```
Select *
  FROM SQLDEMO.Artikel
```

Die Antwort: IBM i unterscheidet an dieser Stelle keine Groß-/Kleinschreibung. Egal wie Sie Ihren SQL-Code eingeben, mit Ausnahme von konstanten Werten spielt die Eingabeform keine Rolle. Intern werden die Eingaben so umgesetzt, dass mit Großbuchstaben gearbeitet wird. Einige SQL-Eingabewerkzeuge setzen daher meist auch die Kleinbuchstaben in Großbuchstaben um.

Tipp

Beachten Sie unbedingt, dass die Klein-/Großschreibung bei Konstanten korrekt angegeben werden muss. Würden wir zum Beispiel nach einem Artikel „Radiergummi“ suchen oder diesen Wert in einer Spalte einfügen wollen, dann sind die folgenden konstanten Werte als unterschiedliche Werte zu betrachten, die genau in der angegebenen Schreibweise in die Spalten eingefügt werden:

„Radiergummi“

„RADIERGUMMI“

„radiergummi“

„RaDIERGUMMI“

Die Regeln für die Verwendung der Groß-/Kleinschreibung gelten nicht nur für die Verwendung der SELECT-Anweisung, sondern richten sich an alle Teilbereiche von SQL.

Eine SELECT-Anweisung dient also der Ermittlung bestimmter Datensätze. Diese werden in Form einer Ergebnistabelle zurückgegeben. Diese Ergebnistabelle bezeichnen wir auch als Resultset. Das Resultset kann dann für weitere Verarbeitungen genutzt werden.

Die SELECT-Anweisung ist heute auch in der Webwelt kaum mehr wegzudenken. Es ist schier unvorstellbar, wie viele SELECT-Anweisungen jeden Tag als Notwendigkeit der Datenermittlung im Internet ausgeführt werden. SQL ist nun einmal nicht nur auf IBM i, sondern auch auf vielen anderen Plattformen im Einsatz. So wundert es auch nicht, dass viele Programmiersprachen sich den Stärken und Standards von SQL bedienen, um schnell und effizient die notwendigen Informationen aus einer Datenbank zu beschaffen.

Was haben wir nun mit der Verwendung der einfachsten der SELECT-Varianten erreicht?

Der Platzhalter „*“ bedeutet, dass wir alle Spalten (bzw. Felder) der angegebenen Tabelle sehen bzw. für die weitere Verarbeitung nutzen wollen.

„FROM Tabellenname“ ist die Angabe der Tabelle, deren Inhalt mit der SELECT-Anweisung verarbeitet werden soll. Dabei kann der Tabellenname qualifiziert angegeben oder in Kurzform verwendet werden. Letzteres wird durch die Arbeitsumgebung bestimmt, in der die SQL-Anweisungen ausgeführt werden sollen. Greift man aus einer Umgebung auf die Tabelle zu, in der die gewünschte Tabelle sich bereits in der Bibliotheksliste befindet, dann ist kein qualifizierter Name erforderlich.

7.6.1
Seite 6

Der Tabellenname wurde in einfacher Form genutzt. Das bedeutet, dass sich die Tabelle bereits in der Umgebung des SQL-Jobs befinden muss, damit ein Ergebnis ermittelt werden kann. So kann zum Beispiel durch das korrekte Setzen der Bibliotheksliste die Voraussetzung für die Verwendung der einfachen Tabellennamen gegeben sein. Es kommt letztlich darauf an, ob die Umgebung für den jeweiligen Job so definiert ist, dass einfache Tabellenname verwendet werden können, oder ob diese durch fixe Namensvorgaben an ein spezielles Schema bzw. eine Bibliothek gebunden werden sollen.

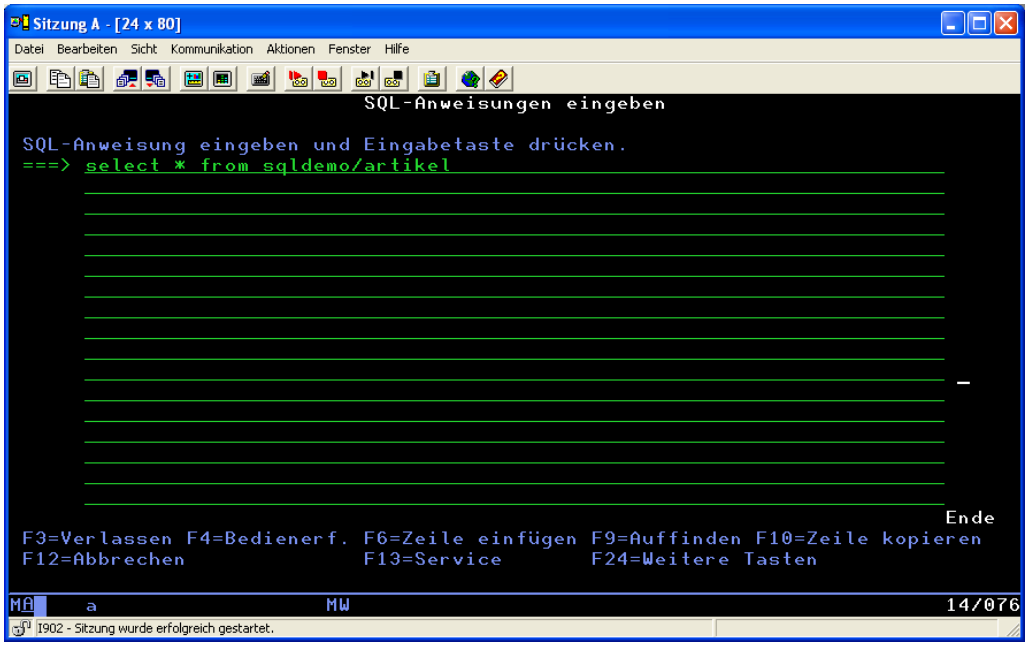
Schauen wir uns deshalb das vorherige Beispiel in einer leicht abgewandelten Form nochmals an. Dabei ergänzen wir den einfachen Tabellennamen durch die Angabe des Schemas, in dem sich die Tabelle befindet.

Damit sieht die SELECT-Anweisung wie folgt aus:

```
SELECT * FROM SQLDEMO.ARTIKEL
```

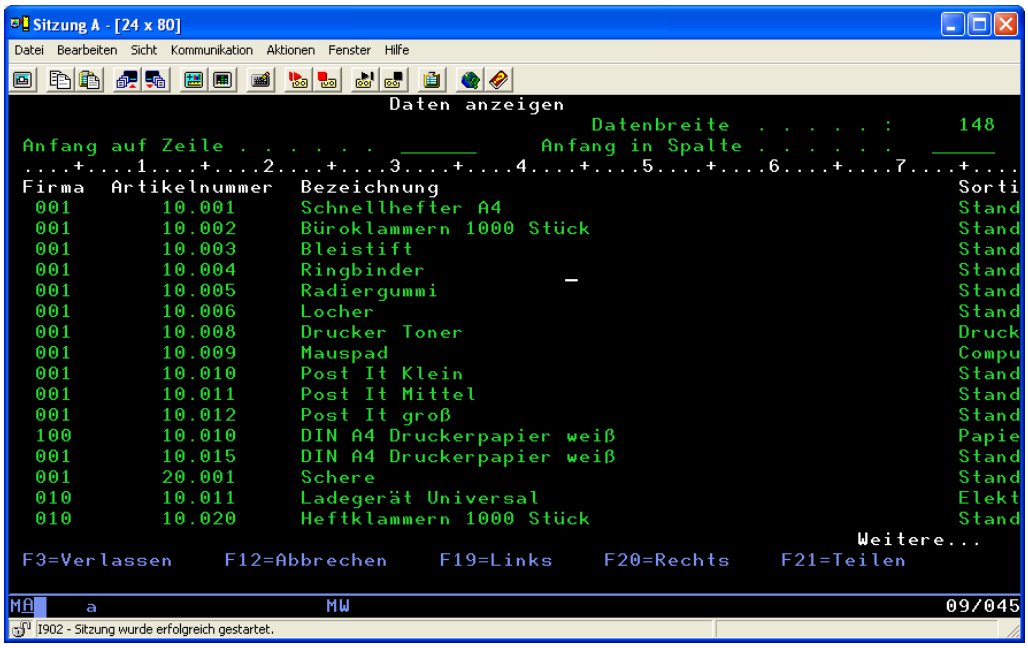
Während in dem ersten SELECT-Beispiel ein Datenbankeditor aus dem Rational-Umfeld genutzt wurde, möchte ich Ihnen nun bewusst die Verwendung des SQL-Editors nativ auf IBM i in der 5250-Umgebung demonstrieren. Der Editor wird über das Lizenzprogramm 57xx-ST1 auf dem System installiert und lässt sich mit dem Befehl STRSQL starten.

Damit gelangen wir in den Editierbereich, in dem wir nun die SELECT-Anweisung eingeben.



SELECT-Anweisung im 5250-SQL-Editor

Das Ergebnis wird dann wie folgt ausgegeben:



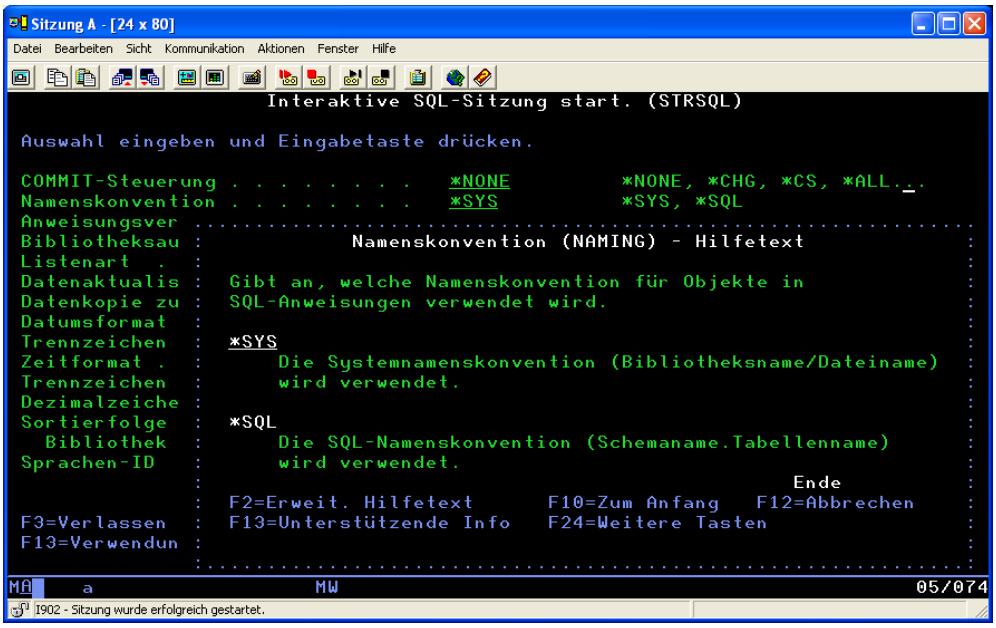
SELECT-Ergebnis in 5250-Anzeige

Tipp

Beachten Sie, dass unter Umständen die Angabe des qualifizierten Namens für die Tabelle abweichend zu dem gezeigten Beispiel erfolgen kann. Der Grund dafür liegt in den verschiedenen „Standards“ im SQL-Umfeld und in IBM i. Während im Beispiel zuvor der Schrägstrich „/“ als Trennzeichen zwischen Schema und Tabelle genutzt wurde, kann bei gewissen SQL-Voreinstellungen auf IBM i auch der Punkt (.) als Trennzeichen genutzt werden.

Der Hintergrund dafür liegt in den IBM i eigenen Namenskonventionen begründet. Diese weichen zum Teil von den ansonsten üblichen SQL-Standards ab. Damit ein Anwender sich aber nicht zu sehr umstellen muss, sind wir in der Lage, die SQL-Voreinstellungen so anzupassen, dass sie dem allgemeinen Standard weitestgehend entsprechen. Für die Verwendung der Trennzeichen bedeutet dies, dass wir den Parameter „Namenskonvention“ für SQL umstellen. Diesen Parameter können wir, wie einige andere SQL-Parameter auch, durch Eingabe des Befehls STRSQL direkt gefolgt von der F4-Taste verändern.

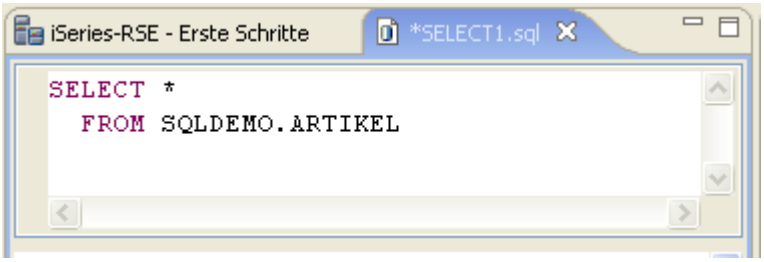
Die folgende Abbildung zeigt dazu ein Beispiel.



SQL-Vorgaben anpassen

In Abhängigkeit von der Verwendung von *SYS oder *SQL kann also der qualifizierte Tabellenname mit einem Punkt oder einem Schrägstrich als Trennzeichen angegeben werden. Diese SQL-Eigenschaft kann nicht nur in den 5250-SQL-Eigenschaften ausgewählt werden, sondern steht auch bei anderen SQL-Werkzeugen zur Auswahl bereit.

Das Gegenstück der qualifizierten SELECT-Eingabe mit dem Punkt als Trennzeichen kann dann beispielsweise in RDi so aussehen.



SELECT mit Punkt als Trennzeichen

Damit haben wir die beiden einfachsten SELECT-Derivate kennengelernt.

Wie Sie gesehen haben, lassen sich damit schnell und unkompliziert die Inhalte von Tabellen anzeigen.

Der Vorteil: Einfache Codierung und schnelle Ausführung. Schnelle Ausführung? Hier wird vielleicht der eine oder andere Leser ein Veto einlegen wollen – denn so schnell ist die Ausführung dieser einfachsten der SQL-Anweisungen leider nicht immer. In unserem einfachen Beispiel – der Beispieltabelle mit Artikelstammdaten – sind nur einige wenige Datensätze enthalten. Diese machen der Maschine wirklich keine Mühe, sie mit einem SELECT-Statement auszugeben. Was ist aber, wenn tausende oder gar Millionen von Datensätzen mit einer SELECT-Anweisung verarbeitet werden? Das kann natürlich spürbar Zeit in Anspruch nehmen. Unter Umständen sitzt der Anwender bei wirklich großen Tabellen dann doch mehrere Sekunden, im schlechten Fall vielleicht gar Minuten vor dem Bildschirm, in der spannenden Erwartung des Ergebnisses. Deshalb gibt es über den Einsatz von einfachen SELECT-Anweisungen hinaus auch noch Möglichkeiten der Einschränkung der SELECT-Ergebnisse, die richtig angewendet, auch bei großen Tabellen nach einer recht kurzen Verarbeitungszeit zur Verfügung stehen.

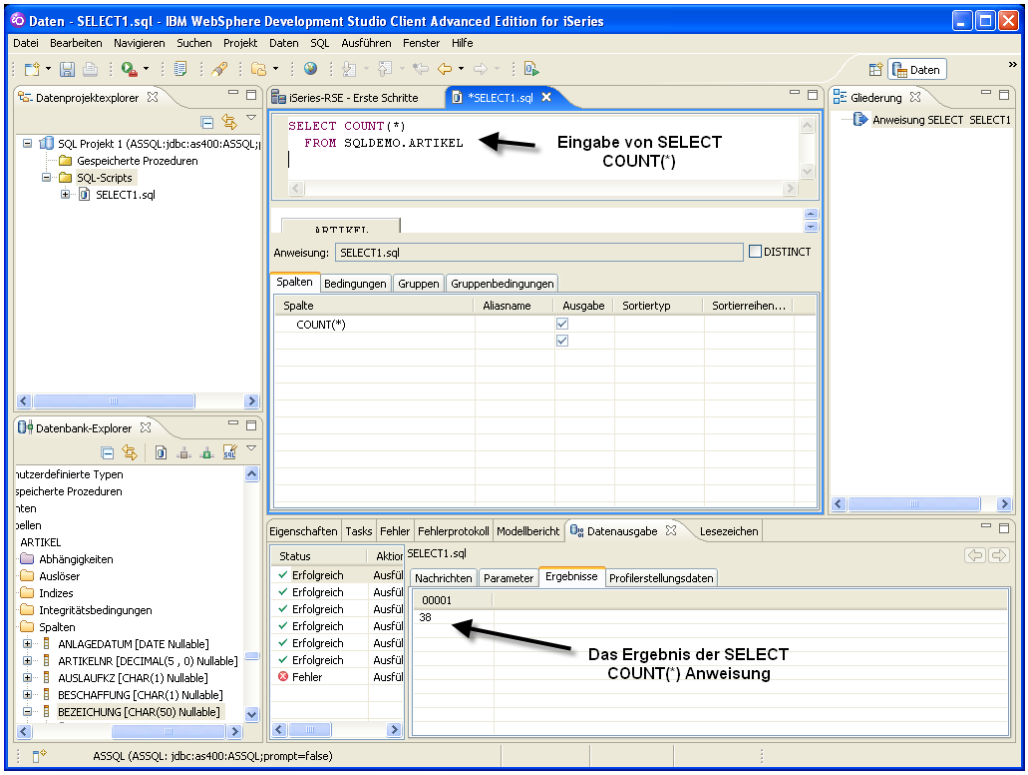
Bevor wir uns mit den Einschränkungen bzw. Optimierungen beschäftigen, sollten wir uns noch eine Variante der SELECT-Anwendung ansehen, die der Ermittlung der Anzahl der Datensätze dient. Dies können wahlweise alle Datensätze innerhalb einer Tabelle oder aber die Anzahl der über Selektionen zu ermittelten Datensätze sein.

Um beispielsweise die Gesamtanzahl der Datensätze in einer Tabelle zu ermitteln, verwenden wir die SELECT-Anweisung mit Angabe COUNT(*). Diese Erweiterung veranlasst SQL dazu, keine Detailliste der Datensätze als Ergebnis zurückzugeben, sondern die Anzahl der ermittelten Datensätze als Ergebnis zu liefern.

Nachfolgend ein Beispiel für den Einsatz des Zusatzes COUNT, mit dem wir die Anzahl der ermittelten Datensätze aus einer Tabelle zurückgeben lassen können.

```
SELECT COUNT(*) FROM ARTIKEL
```

7.6.1
Seite 10



SELECT COUNT

Am Beispiel der Verwendung der Datenperspektive in WDSC haben wir im oberen Fensterbereich den Befehl eingegeben. Das Ergebnis der Auswertung sehen wir im unteren Bereich der Anzeige. Dort erhalten wir als Rückgabewert die Anzahl der nach den Selektionskriterien ermittelten Datensätze. In unserem Beispiel sind dies 38 Datensätze.

Damit haben wir die ersten einfachen SELECT-Ausprägungen kennen gelernt.

7.6.1.1 Spaltenauswahl

Im Kapitel 7.6.1 konnten Sie sehen, dass Sie durch die Angabe SELECT * alle Spalten der jeweiligen Tabelle selektieren. Doch nicht immer werden alle Spalten im Result Set benötigt. Schauen Sie sich die folgende Abfrage an:

```
SELECT Artikelnummer, Bezeichnung, Preis_Netto, Warengruppe
FROM Artikel;
```

In der SELECT-Klausel ist jetzt jeder Spaltenname einzeln aufgeführt. Die Reihenfolge, in der die Spalten genannt sind, entspricht der Reihenfolge, in der sie im Result Set erscheinen. Die Spaltennamen sind durch Komma zu trennen, wobei aber zwischen dem letzten Spaltennamen und der darauffolgenden Klausel – der Klausel FROM – kein Komma stehen darf. Vielleicht ist Ihnen aufgefallen, dass ich die SQL-Anweisung im Gegensatz zu den Beispielen im Kapitel 7.6.1 jetzt mit einem Semikolon abschließe. In manchen Implementierungen von SQL zeigt das Semikolon am Ende der Anweisung dem Interpreter an, dass die Eingabe der Anweisung abgeschlossen ist. Beispielsweise führt die Prozedu-

ARTIKELNUMMER	BEZEICHNUNG	PREIS_NETTO	WARENGRUPPE
00001	Textmarker	4.35	0001
00002	Radiergummi (10)	5.65	0003
00003	Textmarker Rot (10)	9.19	0001
00004	Füllfederhalter	15.46	0001
00005	Kugelschreiber (10)	10.30	0001
00006	Hefter	7.00	0003
00008	Kabelbinder	4.00	0003

Die Spaltenauswahl

Nur die in der Abfrage namentlich genannten Spalten der Tabelle werden angezeigt. Sie können auf diesem Weg die Reihenfolge der Spalten im Result Set individuell bestimmen.

Wenn Sie sich die Spalte WARENGRUPPE ansehen, werden Sie feststellen, dass es Wiederholungen gibt. Wie können wir erreichen, dass gleiche Werte einer Spalte nur einmal in der Ausgabe erscheinen? Probieren Sie folgende Anweisung aus:

```
SELECT DISTINCT Warengruppe
FROM Artikel;
```

Die Abfrage wählt dieses Mal nur Zeilen mit unterschiedlichen Warengruppen aus. Das Schlüsselwort DISTINCT bewirkt, dass doppelt vorhandene Daten nur einmal erscheinen.

Ergänzung 15/2010

7.6.1
Seite 12

In der grundlegenden SELECT-Anweisung ist das Schlüsselwort ALL für alle Datensätze implizit enthalten. Allerdings wird man dieses Schlüsselwort kaum in einer Abfrage finden, da SELECT und SELECT ALL immer dasselbe Ergebnis liefern. Probieren Sie das folgende Beispiel – zum ersten und einzigen Mal in Ihrer SQL-Karriere – aus:

```
SELECT ALL Warengruppe
FROM Artikel;
```

Diese Anweisung entspricht der Anweisung:

```
SELECT Warengruppe
FROM Artikel;
```

Auf die überflüssigen Tastenanschläge für das Wort ALL kann man daher gut und gerne verzichten.

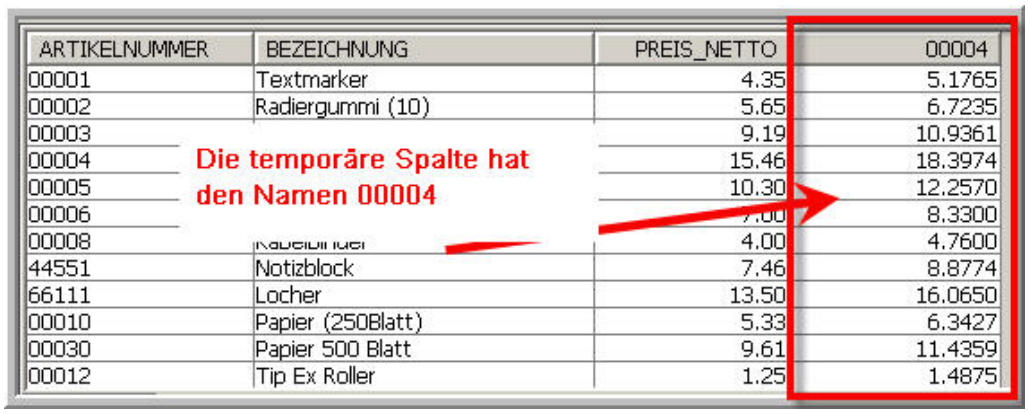
Das Schöne an den SQL-Abfragen ist, dass die Spaltenliste nicht zwingend echte Datenbankfelder enthalten muss, sondern auch Ausdrücke oder Spaltenfunktionen enthalten kann. Die Spaltenfunktion COUNT(*) haben Sie bereits im vorherigen Kapitel kennengelernt, um die Zeilen einer Tabelle zu zählen. Die Funktion COUNT gibt die Zeilenanzahl in der Tabelle zurück.

```
SELECT COUNT(*)
FROM Artikel;
```

Aber wir können auch temporäre Spalten im Result Set aufbauen, indem wir Ausdrücke verwenden. Stellen Sie sich vor, dass der gespeicherte Artikelpreis ein Nettopreis ist. Im Result Set benötigen wir aber sowohl den Netto- als auch den Bruttopreis.

```
SELECT Artikelnummer, Bezeichnung, Preis_Netto,
Preis_Netto * 1.19
FROM Artikel;
```

Das Ergebnis der Abfrage sieht folgendermaßen aus:



ARTIKELNUMMER	BEZEICHNUNG	PREIS_NETTO	00004
00001	Textmarker	4.35	5.1765
00002	Radiergummi (10)	5.65	6.7235
00003		9.19	10.9361
00004		15.46	18.3974
00005		10.30	12.2570
00006		7.00	8.3300
00008		4.00	4.7600
44551	Notizblock	7.46	8.8774
66111	Locher	13.50	16.0650
00010	Papier (250Blatt)	5.33	6.3427
00030	Papier 500 Blatt	9.61	11.4359
00012	Tip Ex Roller	1.25	1.4875

Temporäre Spalten – Ausdrücke

Mit Hilfe der arithmetischen Operatoren Plus (+), Minus (-), Division (/) und Multiplikation (*) können Sie jeden arithmetischen Ausdruck in der Abfrage formulieren. Selbst komplexe Ausdrücke mit Klammerungen sind problemlos möglich:

```
SELECT Bezeichnung,
Warengruppe,
Preis_Netto,
Preis_netto * 19 / 100.00,
Preis_Netto + (Preis_netto * 19 / 100.00)
FROM Artikel;
```

Das Result Set enthält jetzt neben dem Nettopreis den Mehrwertsteuerbetrag und den Bruttopreis:

BEZEICHNUNG	WARENGRUPPE	PREIS_NETTO	00004	00005
Textmarker	0001	4,35	0,8265000000000	5,1765000000000
Radiergummi (10)	0003	5,65	1,0735000000000	6,7235000000000
Textmarker Rot (10)	0001	9,19	1,7461000000000	10,9361000000000
Füllfederhalter	0001	15,46	2,9374000000000	18,3974000000000
Kugelschreiber (10)	0001	10,30	1,9570000000000	12,2570000000000

Komplexe arithmetische Ausdrücke

Sicherlich ist Ihnen aufgefallen, dass ich in der SQL-Abfrage den Punkt als Dezimalzeichen verwende. In der Ausgabe erscheint aber ein Komma als Dezimaltrennzeichen. Welches Dezimaltrennzeichen im Result Set verwendet wird, ist von der Konfiguration Ihrer SQL-Umgebung abhängig. Im System i Navigator bestimme ich im Menü „Verbindung ➜ JDBC-Einstellung ➜ Format“ das Dezimaltrennzeichen für die Ausgabe im Result Set. Kapitel 7.4.1.2.2 enthält detaillierte Informationen zur Konfiguration der Prozedurumgebung des System i Navigators. Um Problemen vorzubeugen, verwende ich in der Abfrage

7.6.1
Seite 14

selbst stets einen Punkt als Dezimaltrennzeichen, obwohl auch das Komma möglich wäre. Jedoch könnte das Komma als Trennzeichen zwischen zwei Spalten interpretiert werden und dann sähe das gerade gezeigte Beispiel vielleicht folgendermaßen aus:

BEZEICHNUNG	WARENGRUPPE	PREIS_NETTO	00004	00005	00006
Textmarker	0001	4,35	0,00	19	5,1765000000000
Radiergummi (10)	0003	5,65	0,00	19	6,7235000000000
Textmarker Rot (10)	0001	9,19	0,00	19	10,9361000000000

Dezimalkomma kann als Spaltentrennzeichen interpretiert werden

Derartige Probleme kann es nicht geben, wenn Sie in der SQL-Abfrage stets einen Punkt als Dezimaltrennzeichen verwenden.

Wir können aber nicht nur arithmetische Operatoren in der Spaltenliste verwenden, sondern uns stehen auch Zeichenkettenoperatoren zur Verfügung. Mit Zeichenoperatoren kann man die Art und Weise der String-Darstellung beeinflussen. Zwei Zeichenketten lassen sich mit dem doppelten Pipe-Symbol (||) oder dem Operator CONCAT verbinden. Schauen wir uns hierfür wieder ein Beispiel an. Die Tabelle KUNDEN enthält eine Spalte mit dem Länderkennzeichen, die Postleitzahl und die Ortsbezeichnung. Im Result Set möchte ich, dass die Spalten folgendermaßen ausgegeben werden:

```
D-28195 Bremen
```

Wir benötigen zunächst die entsprechenden Spalten der Tabelle KUNDEN:

```
SELECT     kdlkz, ,-‘, kdplz, kdort
FROM       kunden;
```

In der Abfrage füge ich bereits einen Bindestrich als temporäre Spalte im Result Set ein. Auch das ist problemlos möglich, wie Sie im Result Set sehen können:

KDLKZ	00002	KDPLZ	KDORT
D	-	28193	BREMEN
D	-	28650	WESTERLAND/SYLT
D	-	28228	BREMEN
D	-	40012	DÜSSELDORF

Mit Konstanten können Sie temporäre Spalten aufbauen

Jetzt ist alles weitere sehr einfach! In einem nächsten Schritt verketten wir die Spalten miteinander, indem wir einfach die Kommata durch den Verkettungsoperator ersetzen:

```
SELECT     kdlkz CONCAT ,-‘ CONCAT kdplz CONCAT kdort
FROM       kunden;
```

Alternativ können Sie den Operator || verwenden.

```
SELECT     kdlkz || ,-‘ || kdplz || kdort
FROM       kunden;
```

Das Ergebnis der Abfrage sehen Sie in der folgenden Abbildung:

00001
D -28193BREMEN
D -28650WESTERLAND/SYLT
D -28228BREMEN
D -40012DÜSSELDORF
D -28231BREMEN
D -28195BREMEN

Das Ergebnis der Verkettung

Noch können wir nicht zufrieden sein! Es fehlt ein Leerzeichen zwischen der Postleitzahl und der Ortsbezeichnung und die nachfolgenden Leerzeichen des Länderkennzeichens müssten auch entfernt werden.

Das fehlende Leerzeichen kann als Konstante schnell hinzugefügt werden:

```
SELECT     kdlkz || ,-‘ || kdplz || , , || kdort
FROM       kunden;
```

Um die Leerzeichen der Spalte KDLKZ zu entfernen, benötigen wir hingegen eine Spaltenfunktion. Spaltenfunktionen sind im SQL zahlreich vorhanden und sollen zu einem späteren Zeitpunkt noch ausführlich behandelt werden, so dass zum jetzigen Zeitpunkt eine kurze Erläuterung ausreicht:

```
Funktionsname(argument1, argument2, …, …)
```

Eine Funktion hat immer einen eindeutigen Namen. Wir benötigen für unser Problem die Funktion TRIM. Jeder Funktion müssen dann ein oder mehrere Argumente übergeben werden, die in Klammern eingeschlossen und durch Komma getrennt sind. Die Funktion TRIM erwartet beispielsweise als Argument lediglich den Namen der Spalte, deren Leerzeichen entfernt werden sollen. Wir müssen also die Spalte KDLKZ als Argument an die Funktion TRIM übergeben:

```
SELECT     Kdnam1, Kdstr, TRIM(kdlkz) || ,-‘ || kdplz || , ,
|| kdort
FROM       kunden;
```

Ergänzung 15/2010

7.6.1
Seite 16

Ich habe meiner Abfrage noch die Spalten KDNAM1 und KDSTR hinzugefügt, so dass das Ergebnis die vollständige Kundenadresse enthält:

KDNAM1	KDSTR	00003
HANDWERKER AN DER WEIDE	AN DER WEIDE 87/88	D-28193 BREMEN
KUNSTHANDLUNG PETERSEN	FRIEDRICHSTRASSE 50	D-28650 WESTERLAND/SYLT
SCHMIDT KOCH	STRESMANNSTR. 1-7	D-28228 BREMEN

Die störenden Leerzeichen wurden entfernt

Normalerweise wird der Spaltenname als Überschrift für das Result Set verwendet. Doch für temporäre Spalten existieren keine Spaltennamen und daher erhält die temporäre Spalte einfach einen virtuellen Spaltennamen. Wie wollen Sie aber auf die Spalte zugreifen, wenn Sie die Abfrage in einem Programm verwenden? Die Spaltennummer können Sie nicht nutzen. Hier ist die Antwort: Das Schlüsselwort AS erlaubt Ihnen die Zuweisung eines Spaltennamens für eine Spalte des Result Sets.

```
SELECT    Kdnam1 as Kundenname,
          Kdstr AS Strasse,
          TRIM(kdlkz) || ,-‘ || kdplz || , , || kdort as
          Adresse
FROM      kunden;
```

Im Result Set erhält die temporäre Spalte mit der Kundenadresse den Namen ADRESSE. Sie müssen hierbei die Namenskonventionen für Spalten berücksichtigen. Sie dürfen also nur maximal 128 Zeichen und die erlaubten Sonderzeichen verwenden. Außerdem werden die Spaltennamen im Result Set stets in Großbuchstaben umgesetzt wie folgende Abbildung zeigt:

KUNDENNAME	STRASSE	ADRESSE
HANDWERKER AN DER WEIDE	AN DER WEIDE 87/88	D-28193 BREMEN
KUNSTHANDLUNG PETERSEN	FRIEDRICHSTRASSE 50	D-28650 WESTERLAND/SYLT
SCHMIDT KOCH	STRESMANNSTR. 1-7	D-28228 BREMEN
WEGNER WOHNSTUDIOS	KÖNIGSALLEE 175/178	D-40012 DÜSSELDORF

Spaltennamen werden in Großbuchstaben umgesetzt

Anders verhält es sich, wenn Sie Spaltenüberschriften für das Result Set vereinbaren wollen:

```
SELECT Artikelnummer, Bezeichnung, Preis_Netto, Preis_Netto *
1.19 AS “Preis (Brutto)“

FROM Artikel;
```

Wenn der Spaltenname in doppelte Anführungszeichen eingeschlossen wird, interpretiert SQL den Wert als Spaltenüberschrift. In diesen Fällen können Sie auch Sonderzeichen verwenden und auch Klein- und Großbuchstaben werden berücksichtigt:

ARTIKELNUMMER	BEZEICHNUNG	PREIS_NETTO	Preis (Brutto)
00001	Textmarker	4.35	5.1765
00002	Radiergummi (10)	5.65	6.7235
00003	Textmarker Rot (10)	9.19	10.9361

Spaltenüberschriften dürfen beliebige Sonderzeichen enthalten

Im letzten Beispiel hat die vierte Spalte des Result Sets keinen Namen, sondern lediglich eine Überschrift erhalten. Derzeit ist der Unterschied noch nebensächlich. Doch sobald wir anfangen mit temporären Datenmengen zu arbeiten, spielt es sehr wohl eine Rolle, ob Sie Spaltennamen oder Spaltenüberschriften verwenden.

7.6.1.2 Daten selektieren

Nicht immer benötigt man alle Zeilen einer Tabelle, sondern sucht nur nach einer Zeile oder einer Gruppe von Zeilen. In diesen Fällen braucht man eine oder mehrere Bedingungen. Diese Bedingungen stehen in der WHERE-Klausel der SELECT-Anweisung. Um alle Artikel zu finden, die mehr als 100,00 Euro kosten, schreibt man zum Beispiel folgende Anweisung:

```
SELECT    Artikelnummer, Bezeichnung, Preis_Netto
FROM      artikel
WHERE     Preis_netto >= 100;
```

ARTIKELNUMMER	BEZEICHNUNG	PREIS_NETTO
00100	Schreibtisch	125,00
00200	Schreibtisch Eck	145,00
10000	SCHREIBTISCH	134,90

Bedingungen formulieren

Mit Bedingungen lassen sich selektive Abfragen realisieren. Die einfachste Form einer Bedingung besteht aus einer Spalte, einer Konstanten und einem Vergleichsoperator. Im ersten Beispiel ist PREIS_NETTO die Spalte, der Wert 100 ist eine Konstante und >= ist der Vergleichsoperator. Bevor man allerdings bedingte Abfragen erstellen kann, muss man die WHERE-Klausel und die möglichen Vergleichsoperatoren kennen. Sie können alle bekannten arithmetischen Vergleichsoperatoren wie Gleich (=), Ungleich (!= oder <>), Größer (>), Größer gleich (>=) und Kleiner gleich (<=) verwenden.

7.6.1
Seite 18

Suchen wir doch einmal nach dem Artikel SCHREIBTISCH:

```
SELECT      *
FROM        artikel
WHERE       Bezeichnung = 'Schreibtisch';
```

ARTIKELNUMMER	BEZEICHNUNG	PREIS_NETTO
00100	Schreibtisch	125,00

Ein Artikel mit der Bezeichnung „Schreibtisch“ wurde gefunden

Und jetzt wollen wir alle Artikel selektieren, die ungleich der Bezeichnung ‘Schreibtisch‘ sind:

```
SELECT      Artikelnummer, Bezeichnung, Preis_Netto
FROM        artikel
WHERE       Bezeichnung <> 'Schreibtisch';
```

Alternativ können Sie die Symbole != verwenden.

ARTIKELNUMMER	BEZEICHNUNG	
00012	Tip Ex Roller	1,25
00003	Textmarker Rot (10)	9,19
00001	Textmarker	4,35
10000	SCHREIBTISCH	134,90
10100	Schuhschrank	89,90
00200	Schreibtisch Eck	145,00
00002	Radiergummi (10)	5,65

Der Operator UNGLEICH

Um die Bedingung ungleich zu formulieren, können Sie entweder die beiden Symbole < > oder die Zeichen != verwenden.

Eines wird anhand der Abfrage aber auch deutlich: Die Bezeichnung ‚SCHREIBTISCH‘ ist nicht identisch mit dem Wert ‚Schreibtisch‘. Auch wenn die SQL-Syntax selbst nicht von der Groß-/Kleinschreibung abhängig ist, wird bei den Konstanten streng auf die Schreibweise geachtet. Die meisten Firmen speichern die Daten daher häufig durchgehend in Groß- oder Kleinbuchstaben, um einheitliche Datenbestände zu gewährleisten. Doch mit SQL lässt sich auch dieses Problem umgehen.

Wie in der Spaltenliste können Sie auch in den Bedingungen Spaltenfunktionen verwenden. Die Funktion UPPER beispielsweise setzt den Inhalt einer Spalte, die als Argument übergeben wird, in Großbuchstaben um und vergleicht anschließend diesen Wert mit der Konstanten. Probieren wir es aus:

```
SELECT      Artikelnummer, Bezeichnung, Preis_Netto
FROM        artikel
WHERE       UPPER(Bezeichnung) = 'SCHREIBTISCH';
```

ARTIKELNUMMER	BEZEICHNUNG	PREIS_NETTO
00100	Schreibtisch	125,00
10000	SCHREIBTISCH	134,90

Wirkungsweise der Funktion UPPER

Das Problem der Groß- und Kleinschreibung lässt sich auf diesem Weg lösen. Doch unser „Schreibtisch Eck“ wird nach wie vor nicht im Result Set angezeigt. Das Beispiel hat allerdings sehr gut gezeigt, dass Sie die arithmetischen Vergleichsoperatoren problemlos auch auf Zeichenfelder anwenden können. Die Ergebnisse hängen vom verwendeten EBCDIC-Zeichensatz ab, der die Sortierfolge bestimmt.

Häufiger als die arithmetischen Vergleichsoperatoren wird allerdings der Operator LIKE für alphanumerische Vergleiche gewählt. Möchte man Daten abrufen, die einem bestimmten Muster entsprechen, kann man mit dem Operator LIKE arbeiten. Sehen Sie sich dazu folgendes Beispiel an.

```
SELECT    Artikelnummer, Bezeichnung, Preis_Netto
FROM      Artikel
WHERE     UPPER(Bezeichnung) LIKE ,%SCHREIBTISCH%‘;
```

Das Prozentzeichen dient im LIKE-Ausdruck als Platzhalter für eine beliebige Anzahl von Zeichen. Wir haben damit alle Datensätze gesucht, bei denen in der Spalte Bezeichnung ein „Schreibtisch“ vorkommt. Mit der Anweisung

```
SELECT    Artikelnummer, Bezeichnung, Preis_Netto
FROM      Artikel
WHERE     UPPER(Bezeichnung) LIKE ,SCHREIBTISCH%‘;
```

erhält man alle Zeilen, die mit ‘Schreibtisch‘ beginnen. Alternativ zum %-Zeichen können Sie auch den Unterstrich als Platzhalter für genau ein Zeichen nutzen. Die folgende Abfrage sucht alle Kunden, die MEIER oder MEYER heißen.

```
SELECT    Kdnam1
FROM      Kunden
WHERE     kdnam1      LIKE    ,ME_ER%‘;
```

Achten Sie darauf, dass Sie neben dem Unterstrich auch das nachfolgende Prozentzeichen in der Abfrage verwenden, andernfalls würden Sie keine Ergebniszeilen im Result Set erhalten. Das nachfolgende Prozentzeichen wird bei Zeichenspalten fester Länge benötigt, um die in der Spalte vorhandenen Leerzeichen zu maskieren. Ohne Prozentzeichen dürfte die Spalte KDNAM1 nur fünf Zeichen lang sein, so dass keine nachfolgenden Leerzeichen vorhanden sind.

Nun wollen wir aber mehr als eine Bedingung formulieren. Hierfür verwenden wir die logischen Operatoren AND, OR und NOT. Mit logischen Operatoren lassen sich mehrere Bedingungen in der WHERE-Klausel einer SQL-Anweisung kombinieren.

Der Operator AND liefert das Ergebnis WAHR zurück, wenn beide Bedingungen WAHR sind. Ist eine der Bedingungen FALSCH, wird der gesamte Ausdruck mit FALSCH bewertet. Mit der folgenden Anweisung sucht man zum Beispiel alle Artikel, die zur Warengruppe '1540' gehören und mehr als 30,00 Euro kosten:

```
SELECT      Artikelnummer, Bezeichnung, Preis_Netto
FROM        artikel
WHERE       Warengruppe = '1540' and Preis_netto > 30.00;
```

ARTIKELNUMMER	BEZEICHNUNG	PREIS_NETTO	WARENGRUPPE
00100	Schreibtisch	125,00	1540
00200	Schreibtisch Eck	145,00	1540
10000	SCHREIBTISCH	134,90	1540
10340	Bücherregal	52,00	1540

Komplexe Bedingungen

Mit dem Operator OR kann man auch mehrere Bedingungen zusammenfassen. Wenn mindestens eine der Bedingungen WAHR ist, wird die Zeile selektiert. Um den Unterschied zu zeigen, führen wir die letzte Abfrage mit dem Operator OR anstelle von AND aus:

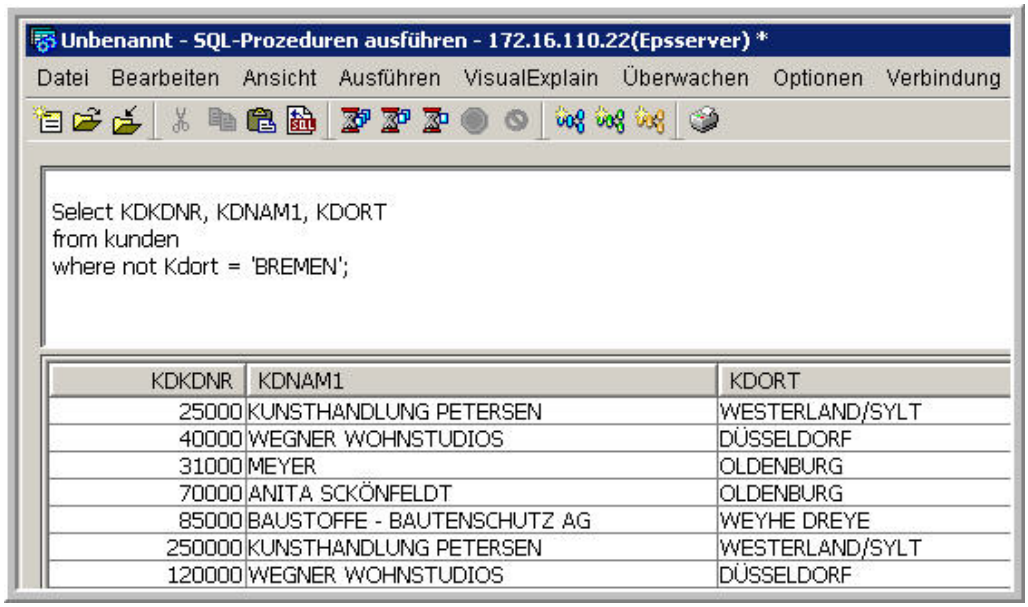
```
SELECT      Artikelnummer, Bezeichnung, Preis_Netto
FROM        artikel
WHERE       Warengruppe = '1540' OR Preis_netto > 30.00;
```

ARTIKELNUMMER	BEZEICHNUNG	PREIS_NETTO
00100	Schreibtisch	125,00
00200	Schreibtisch Eck	145,00
44552	Aktenvernichter Z100	24,17
10000	SCHREIBTISCH	134,90
10100	Schuhschrank	89,90
10340	Bücherregal	52,00

Der logische Operator OR

Sie können sehen, dass der Artikel „Aktenvernichter“ jetzt zusätzlich im Result Set erscheint, obwohl er weniger als 30,00 Euro kostet.

Wenn eine formulierte Bedingung WAHR zurückgibt, wandelt der logische Operator NOT das Ergebnis in FALSCH um. Liefert die Bedingung hingegen den Wert FALSCH, heißt das Ergebnis WAHR. Zum Beispiel selektiert die folgende SELECT-Anweisung alle Kunden, die nicht in Bremen wohnen.



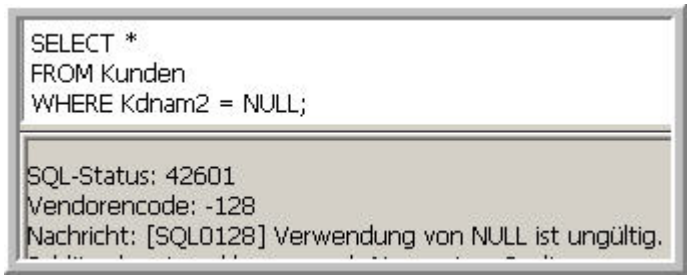
Der logische Operator NOT

In diesem Zusammenhang müssen wir noch ein wenig auf den Wert NULL eingehen. In Datenbanken bezeichnet der Wert NULL das Fehlen von Daten in einer Tabellenspalte. Das heißt nicht, dass die Spalte eine 0 oder ein Leerzeichen enthält. Die 0 oder das Leerzeichen sind echte Werte. NULL bedeutet, dass tatsächlich nichts in diesem Feld enthalten ist. Führt man einen Vergleich wie KDNAM2 = ‘HUBER‘ aus und der einzige Wert für die Spalte ist NULL, liefert der Vergleich das Ergebnis „unbekannt“. Da das Ergebnis „unbekannt“ aber etwas aus dem Rahmen fällt, ändern die meisten Datenbankhersteller das Ergebnis „unbekannt“ in FALSE und stellen den speziellen Operator IS NULL für den Test auf eine NULL-Bedingung bereit. Das folgende Beispiel demonstriert den Wert NULL. Nehmen wir die Spalte KDNAM2 in der Tabelle KUNDEN. Die Spalte darf aufgrund der Tabellendefinition NULL-Werte enthalten:

7.6.1
Seite 22

Darstellung des Wertes NULL

In der Prozedurumgebung des System i Navigator wird der Wert NULL durch ein Minuszeichen dargestellt und ist dadurch gut zu erkennen. Andere SQL-Umgebungen nutzen diese Darstellung nicht. In diesen Umgebungen bleibt die Spalte einfach leer und Sie müssen wissen, ob es sich um einen NULL-Wert oder um ein Leerzeichen handelt. Versuchen wir nun alle Zeilen der Tabelle KUNDEN aufzuspüren, die den Wert NULL in der Spalte KDNAM2 enthalten:



Die Verwendung von NULL ist ungültig

Die Abfrage wird nicht ausgeführt, wenn wir die Bedingung KDNAM2 = NULL formulieren, weil das Schlüsselwort NULL zusammen mit dem angegebenen Operator ungültig ist. NULL ist nur mit dem Operator IS oder IS NOT erlaubt. Wollen wir also alle Zeilen in der Tabelle KUNDEN finden, die NULL in der Spalte KDNAM2 enthalten, müssen wir die SQL-Abfrage folgendermaßen formulieren:

```
SELECT    KDKDNR, KDNAM1, KDNAM2
FROM      Kunden
WHERE     Kdnam2 IS NULL;
```

Sollen hingegen alle Zeilen ausgewählt werden, die nicht den Wert NULL enthalten, formulieren wir die Abfrage mit dem logischen Operator NOT:

```
SELECT    KDKDNR, KDNAM1, KDNAM2
FROM      Kunden
WHERE     Kdnam2 IS NOT NULL;
```

Viele unserer DB2-Tabellen enthalten gar keine NULL-fähigen Spalten, so dass wir derzeit noch wenig Verwendung für den Ausdruck IS NULL haben, aber wir werden demnächst temporäre Tabellen in unseren Abfragen aufbauen und dann kann sich der NULL-Operator als durchaus nützlich erweisen.

Die beiden Operatoren IN und BETWEEN sind nicht zwingend erforderlich, denn Sie könnten Ihre Abfragen auch ohne diese Operatoren realisieren. Allerdings lässt sich die ein oder andere Abfrage mit diesen Operatoren sehr viel einfacher formulieren. Schauen wir uns ein Beispiel an. Ich benötige alle Kunden aus Deutschland, Österreich und der Schweiz:

```
SELECT    Kdnam1 as Kundenname,
          Kdstr AS Strasse,
          TRIM(kdlkz) || ,-‘ || kdplz || , , || kdort as
          Adresse
FROM      Kunden
WHERE     Kdlkz = ,CH‘ OR KDLKZ = ,A‘ OR KDLKZ = ,D‘;
```

Kürzer ist diese Abfrage mit dem Operator IN zu formulieren:

```
SELECT    Kdnam1 as Kundenname,
          Kdstr AS Strasse,
          TRIM(kdlkz) || ,-‘ || kdplz || , , || kdort as
          Adresse
FROM      Kunden
WHERE     Kdlkz IN (,CH‘, ,A‘, ,D‘);
```

Das zweite Beispiel ist kürzer und leichter zu erfassen. Man kann nie im Voraus wissen, wann man wieder ein Projekt überarbeiten muss, das bereits Monate zurückliegt.

Benötigt man einen Bereich aus der Tabelle KUNDEN, weil bspw. alle Kunden des Postleitzahlengebiets 28 angeschrieben werden sollen, kann man folgende Anweisung ausführen:

```
SELECT    Kdnam1 as Kundenname,
          Kdstr AS Strasse,
          TRIM(kdlkz) || ,-‘ || kdplz || , , || kdort as
          Adresse
FROM      Kunden
WHERE     Kdplz >= ,28000‘ AND Kdplz <= ,28999‘ ;
```

Ergänzung 15/2010

7.6.1
Seite 24

Auch jetzt ist diese Anweisung wieder einfacher und besser zu formulieren als die erste, wenn man den Operator BETWEEN einsetzt.

```
SELECT    Kdnam1 as Kundenname,
          Kdstr AS Strasse,
          TRIM(kdlkz) || '-' || kdplz || ' ' || kdort as
          Adresse
FROM      Kunden
WHERE     Kdplz BEWEEN '28000' AND '28999' ;
```

Die Operatoren IN und BETWEEN sind also nicht unbedingt erforderlich, erleichtern aber in vielen Fällen die Formulierung der Abfrage.

7.6.1.2.1 Reguläre Ausdrücke verwenden

Wenn Sie bereits ein wenig mit SQL-Abfragen gearbeitet haben, werden Sie sicherlich auch mit der WHERE-Klausel vertraut sein, denn in den meisten Abfragen benötigt man lediglich einen Datenausschnitt und nur ganz selten werden wirklich alle Zeilen einer Tabelle verarbeitet.

Ist Ihnen bei der Arbeit auch aufgefallen, dass es relativ schwer ist, Zeichenketten anhand eines Musters zu durchzusuchen, um beispielsweise die korrekten Formate der Dateninhalte zu prüfen? Oder haben Sie vielleicht auch schon mal die Möglichkeit vermisst, komplexe Suchmuster aufzubauen, um einen Kunden oder Lieferanten anhand des Musters zu selektieren?

Mit Release 7.1 TR9 können jetzt sehr komplexe Suchmuster aufgebaut werden, anhand derer die Daten untersucht werden. Als Erstes wurde seitens IBM dafür das Prädikat REGEXP_LIKE eingeführt.

Schauen wir uns dazu ein einfaches Beispiel an. Ausgangsbasis ist zunächst wieder unsere Artikeltabelle:

BEZEICHNUNG	WARENGRUPPE	MINDESTBESTAND	AUSLAUFKZ	BESCHAFFUNG	MENGENEINHEIT
Mauspad Gummi	0002	10	N	L	Stck
Radiergummi 10er	0003	40	N	L	Pckg
Textmarker 10er Pack	0001	30	N	L	Pckg
Füllfederhalter	0001	10	N	L	Stck
Kugelschreiber (11)	0001	100	N	L	Pckg.
Hefter	0003	10	N	K	Stck.
Kabelbinder	0003	900	N	L	Pckg
Hefter RotBraun	0002	10	J	L	Stck.

Ausgangsbasis sind die Artikelstammdaten

Als Erstes suchen wir alle Artikel, die mit der Zeichenfolge TEXT beginnen. Die Großschreibung soll dabei ignoriert werden.

```
SELECT *
FROM ARTIKEL
WHERE REGEXP_LIKE(BEZEICHNUNG, '^TEXT' , 1, 'I' );
```

Im gezeigten Beispiel verwende ich vier Parameter für das Prädikat. Der erste Parameter bestimmt die zu durchsuchende Tabellenspalte. Dann folgt das Suchmuster. Durch das vorangestellte ^ wird festgelegt, dass der gesuchte String TEXT am Wortanfang stehen muss. Der dritte Parameter legt die Startposition für die Suche fest und der Buchstabe i – als vierter Parameter – schaltet die Groß- und Kleinschreibung aus, das heißt, wir suchen case-insensitive. Als Ergebnis dieser Abfrage erhalten wir folgende Zeile:

7.6.1
Seite 26

FIR...	ARTIKELNU...	BEZEICHNUNG	WARENGR...	MINDESTBES...	AUSLAU...
001	300	Textmarker 10er Pack	0001	30	N

Das Ergebnis

Sie haben Recht, wenn Sie jetzt sagen, dass Sie das Ergebnis auch mit einer einfachen LIKE-Abfrage realisieren können. Die nachfolgende Abfrage liefert das gleiche Ergebnis:

```
SELECT *
FROM ARTIKEL
WHERE UPPER(BEZEICHNUNG) LIKE 'TEXT%' ;
```

Nur sind wir mit den LIKE-Abfragen sehr schnell am Ende. Wie wollen Sie beispielsweise die LIKE-Abfrage gestalten, wenn Sie verifizieren müssen, ob Telefonnummern korrekt erfasst wurden oder E-Mail-Adressen logisch korrekt aufgebaut sind? Das heißt, das Prädikat REGEXP_LIKE gewinnt um so mehr an Bedeutung, je komplexer die Suchmuster werden.

Im nachfolgenden Beispiel wird beispielsweise geprüft, ob die erfassten Warengruppen ausschließlich aus Ziffern bestehen. Fehlerhaft erfasste Warengruppen werden selektiert:

```
SELECT BEZEICHNUNG , WARENGRUPPE
FROM ARTIKEL
WHERE NOT REGEXP_LIKE(WARENGRUPPE, '[0-9]{4}' );
```

Schauen wir uns den Aufbau etwas genauer an:

Im ersten Parameter steht stets die Zeichenkette, die geprüft werden soll. Das kann eine Hostvariable sein, eine Textkonstante oder – wie im gezeigten Beispiel – die Spalte einer Tabelle.

Im zweiten Parameter definieren Sie Regeln, nach denen gesucht werden soll. Diese Definition wird als Zeichenkettenmuster eingegeben und muss daher in einfache Anführungszeichen eingeschlossen werden. Auf diesen Parameter müssen wir in einem zweiten Schritt noch etwas genauer eingehen.

Der dritte Parameter ist dann bereits optional. Er bestimmt die Startposition der Suche.

Der vierte Parameter ist ebenfalls optional. Darüber können wir – wie im ersten Beispiel gezeigt – festlegen, ob die Groß- und Kleinschreibung zu beachten ist oder Leerzeichen im Suchstring ignoriert werden.

Kommen wir jetzt zurück zur Suchmusterdefinition:

```
WHERE NOT REGEXP_LIKE(WARENGRUPPE, '[0-9]{4}' );
```

Im gezeigten Beispiel prüfe ich, ob in den ersten vier Zeichen der Spalte „Warengruppe“ nur die Zeichen 0 bis 9 verwendet werden. Wäre meine Warengruppennummer komplexer aufgebaut, ließe sich auch das abbilden. Stellen wir uns folgenden Aufbau vor:

- Zeichen A, B oder C
- Zeichen beliebige Ziffer
- Zeichen beliebige Ziffer
- Zeichen Ziffer 3 oder 4

Der zugehörige Suchstring könnte folgendermaßen aussehen:

```
'[A-C]{1}[0-9]{2}[3-4]{1}'
```

In eckigen Klammern geben Sie stets den Gültigkeitsbereich an. Darauf folgt in geschweiften Klammern die Anzahl dieser Zeichen. Haben Sie keinen fortlaufenden Gültigkeitsbereich, können Sie die einzelnen Werte durch Komma trennen, wie im nachfolgenden Beispiel gezeigt wird.

```
'[A-C]{1}[0-9]{2}[3,7]{1}'
```

Schauen wir uns ein weiteres Beispiel an: Wie werden Telefonnummern gewöhnlich erfasst (DIN5008)?

+49 442112345-23

Versuchen wir auch hierfür ein Suchmuster aufzubauen:

```
WHERE REGEXP_LIKE
(
trim(customer_telephone), '^(\+[1-9]{2})[\s]{1}[0-9]{2,}
      (\-[0-9]{1,})$|^(\+[0-9]{2})[\s]{1}[1-9]{2,}'
);
```

Das Suchmuster ist sehr viel komplexer, aber die Regeln sind auch deutlich komplizierter:

Am Anfang soll die Telefonnummer mit einem Pluszeichen beginnen, gefolgt von zwei Ziffern aus dem Wertebereich 1 bis 9.

Diese Regel implementieren wir, indem wir folgendes Suchmuster hinterlegen:

```
^(\+[1-9]{2})
```

7.6.1
Seite 28

Das Zeichen ^( ) bestimmt, dass die Zeichenfolge am Anfang steht. Da das Plus-Zeichen selbst auch ein Metazeichen innerhalb des Suchstrings sein kann, benötigen wir den Backslash, um kenntlich zu machen, dass wir ein einfaches Pluszeichen als Bestandteil des Suchstrings meinen. Es folgen zwei Ziffern.

Danach soll ein Leerzeichen folgen:

```
[\s]{1}
```

Das Leerzeichen ist ebenfalls ein Sonderzeichen und wird durch \s dargestellt.

Nun folgen mindestens zwei beliebige Ziffern aus dem Wertebereich 0 bis 9:

```
[0-9]{2,}
```

Dass wir zwei oder mehr Ziffern erwarten, machen wir deutlich, indem wir hinter der Anzahl ein Komma verwenden.

Abschließend enthält unsere Telefonnummer noch die Durchwahl. Sie endet in diesem Fall mit einem Bindestrich gefolgt von einer beliebigen Anzahl Ziffern.

```
(\-[0-9]{1,})$
```

Wieder wird der Suchausdruck geklammert und ein Dollarzeichen hintenangestellt, um kenntlich zu machen, dass es sich um das Ende der Zeichenfolge handelt. Auch das Minuszeichen selbst muss wieder mit einem Backslash maskiert werden, andernfalls würde es als Metazeichen interpretiert werden.

Vielleicht haben aber nicht alle Telefonnummern eine Durchwahl? Dafür könnten Sie einen Alternativsuchstring aufbauen. Eingeleitet wird die Oder-Bedingung mit einem Pipe-Zeichen. Anschließend folgt dann wieder das bekannte Suchmuster – allerdings ohne Ende-Bedingung.

```
|^(\+[0-9]{2})[\s]{1}[1-9]{2,}
```

Der Aufbau der Suchmuster ist anfangs etwas gewöhnungsbedürftig und erfordert ein wenig Übung, aber Sie finden eine Menge von Beispielen im Internet und eine nähere Beschreibung der Metazeichen und deren Bedeutung unter dem folgenden Link:

http://www-01.ibm.com/support/knowledgecenter/ssw_ibm_i_72/db2/rbafzregexp_like.htm?lang=en

Aber es wurde für die Verwendung von regulären Ausdrücken nicht nur das Prädikat REGEXP_LIKE eingeführt. Zusätzlich gibt es eine Reihe skalarer Funktionen, die ebenfalls mit Suchmustern arbeiten. Eine kurze Beschreibung dieser Funktionen mit Beispiel finden Sie im Kapitel „SQL-Funktionen im praktischen Einsatz“.

7.6.1.3 SQL-Spaltenfunktionen

Funktionen kennen Sie sicherlich aus Ihrem Mathematikunterricht. Hinter einem schlichten Wort wie sin (=Sinus) verbirgt sich eine komplexe mathematische Operation. Eine Funktion wenden Sie wie eine Blackbox an, ohne sich über deren Realisierung Gedanken zu machen. Es genügt zu wissen, was die Funktion bewirkt. So verhält es sich auch mit SQL-Funktionen. Häufig wiederkehrende Operationen, die ein Ergebnis abhängig von bestimmten Werten liefern, lassen sich komfortabel in Funktionen realisieren. Funktionen sind somit benannte Befehlsfolgen, die eine festgelegte Aktion ausführen und als Ergebnis einen einzelnen Wert zurückgeben. Dabei unterscheidet man im SQL zwischen vordefinierten Funktionen und benutzerdefinierten Funktionen. Hier soll es sich zunächst nur um vordefinierte SQL-Funktionen handeln. An die meisten Funktionen kann man beim Aufruf ein oder mehrere Argumente übergeben. Diese Argumente können Sie sich wie eine Variable vorstellen, die einen Wert von der aufrufenden SQL-Anweisung an die Funktion übermittelt. Die Funktion liefert dann das entsprechende Ergebnis zurück.

Eine Funktion lässt sich überall dort verwenden, wo die Syntax einer Anweisung einen einzelnen Wert erwartet. Somit können Sie Funktionen z. B. in der WHERE-Klausel verwenden, um logische Bedingungsausdrücke zu bilden oder in der Spaltenliste der SQL-Abfrage. Erste Verwendungen haben Sie bereits im Kapitel 7.6.1.2 gesehen. Nun wollen wir uns detaillierter mit dem Einsatz dieser Funktionen und deren Möglichkeiten beschäftigen. In SQL unterscheidet man zunächst einmal zwischen einer Aggregatfunktion und einer Skalarfunktion.

Aggregatfunktionen gehören zum ‚harten Kern' von SQL. Sie sind in jeder Datenbank zu finden und gehören zum ANSI-Standard. Aggregatfunktionen bilden aus einer Menge von Werten genau einen Ergebniswert und geben diesen an die SQL-Abfrage zurück. Die Aggregatfunktion COUNT(*) haben Sie bereits kennen gelernt. Sie arbeitet mit allen Zeilen der Tabelle und liefert als Ergebniswert die Zeilenanzahl. Aggregatfunktionen sind in erster Linie in Verbindung mit den Klauseln GROUP BY und HAVING vorgesehen, können aber auch auf ganze Tabellen angewendet werden.

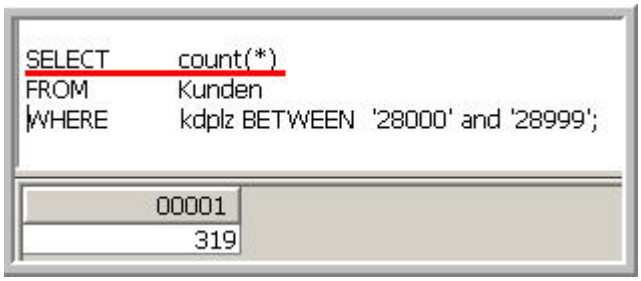
Skalarfunktionen arbeiten hingegen mit einzelnen Spaltenwerten und liefern einen einzelnen Ergebniswert. Die Eingabedaten bezieht eine Skalarfunktion entweder aus den übergebenen Argumenten oder vom zugrunde liegenden Betriebssystem. Auch eine Skalarfunktion haben wir bereits eingesetzt. Die Funktion UPPER erhielt in unserem Beispiel den Spaltennamen als Argument und wandelte für jede einzelne Zeile die Dateninhalte in Großbuchstaben um. Während im SQL-Standard-92 nur eine Hand voll Funktionen definiert war, hat der Standard SQL-99 die Palette wesentlich erweitert. Außerdem hat IBM – wie die meisten Datenbankhersteller – viele eigene Funktionen implementiert.

Ergänzung 15/2010

7.6.1.3.1 Aggregatfunktionen

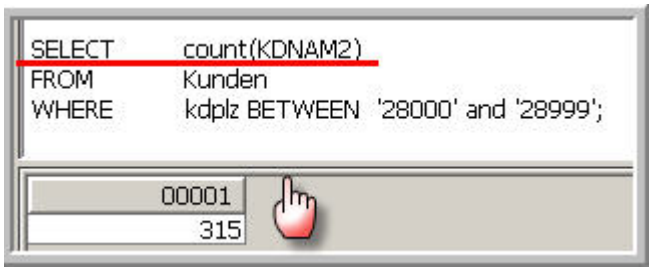
Die Aggregatfunktionen AVG(=Durchschnittswert), SUM(=Summe bilden), MAX(=Maximalwert), MIN(=Minimalwert) und COUNT(=Anzahl) sind schnell erklärt und sicherlich kennen Sie diese Funktionen auch bereits, wenn Sie mit Query/400 oder dem Query Manager arbeiten.

Die Funktion COUNT liefert die Anzahl der Zeilen, die der Bedingung in der WHERE-Klausel entsprechen. Zum Beispiel lässt sich mit der folgenden Anweisung die Anzahl der Kunden in Bremen ermitteln:



Die Aggregatfunktion COUNT(*)

319 Kunden werden für Bremen ermittelt. Ich kann an die Aggregatfunktion aber auch einen Spaltennamen als Argument übergeben:



Die Aggregatfunktion COUNT(spaltenname)

Nun liefert mir die Abfrage als Ergebnis den Wert 315! Das Beispiel zeigt deutlich die unterschiedlichen Wirkungsweisen: Während COUNT(*) die Anzahl Zeilen der Tabelle ermittelt, zählt COUNT(spaltenname) alle Einträge in der jeweiligen Spalte, die nicht NULL sind und wir wissen ja bereits, dass die Spalte KDNAM2 unserer Tabelle Kunden NULL-Werte in der Spalte KDNAM2 zulässt.

Die Funktion COUNT_BIG ähnelt der Funktion COUNT, lässt aber ein Ergebnis über dem Maximalwert von INTEGER zu. COUNT_BIG setzen Sie also ein, wenn Ihre Tabelle bspw. mehr als 2.147.483.647 Zeilen enthält.

Die Funktion SUM liefert die Summe aller Werte einer Spalte zurück. Die Funktion SUM arbeitet nur mit numerischen Feldern. Wendet man die Funktion auf ein nichtnumerisches Feld an, erhält man eine Fehlermeldung. Wir wollen jetzt ermitteln, welchen Auftragswert wir haben:

```
SELECT
SUM(auftragswert) AS „Auftragswert(Netto)“,
SUM(auftragswert * 1.19) AS „Auftragswert(Brutto)“,
COUNT(*) AS „Anzahl Aufträge“
FROM Auftrag;
```

Sie sehen in der Abfrage, dass ich an die Aggregatfunktionen aber nicht nur einzelne Tabellenspalten sondern auch Spaltenausdrücke übergeben kann. Außerdem kann ich die mittels Aggregatfunktion aufgebauten temporären Spalten auch wieder in Ausdrücken verwenden:

```
SELECT
SUM(auftragswert) AS „Auftragswert(Netto)“,
SUM(auftragswert * 1.19) AS „Auftragswert(Brutto)“ ,
COUNT(*) AS ANZAHL,
SUM(auftragswert) / COUNT(*) AS „Durchschnittl. Auftragswert“
FROM Auftrag;
```

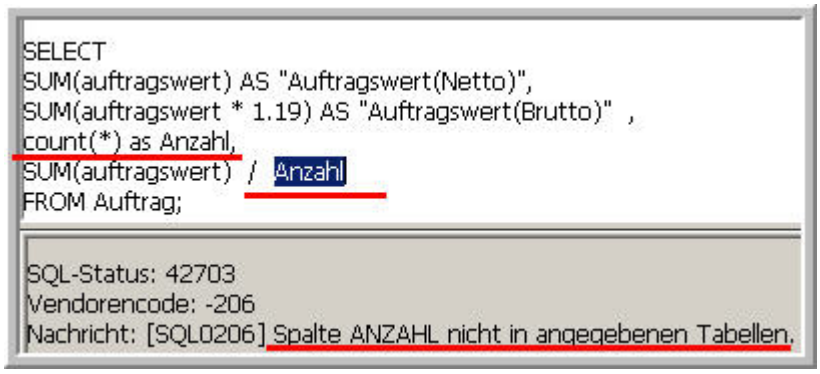
Im Result Set haben wir jetzt die folgenden Informationen:

Auftragswert(Netto)	Auftragswert(Brutto)	ANZAHL	Durchschnittl. Auftragswert
55544.50	66097.9550	43	1291.73

Einsatzmöglichkeiten Aggregatfunktionen

7.6.1
Seite 32

An dieser Stelle kommt oft der Wunsch auf, die im Result Set vorhandene Spalte ANZAHL für die Durchschnittsberechnung direkt zu verwenden; dies würde schließlich viel Schreibarbeit ersparen:



Spalte Anzahl nicht in der angegebenen Tabelle

Sie erhalten eine Fehlermeldung, weil Sie in der Spaltenliste nur Spaltennamen der Tabelle AUFTRAG verwenden dürfen. Sie müssen also stets – wie im ersten Beispiel gezeigt – den gesamten Ausdruck wiederholen oder andere Techniken anwenden, die wir zum jetzigen Zeitpunkt allerdings noch nicht kennen.

Die Funktion AVG berechnet den Mittelwert einer Spalte. Wir könnten daher auch das eben gezeigte Beispiel mit der Aggregatfunktion AVG lösen:

```
SELECT
SUM(auftragswert) AS „Auftragswert(Netto)“,
AVG(Auftragswert) AS „Durchschnittl. Auftragswert“
FROM Auftrag;
```

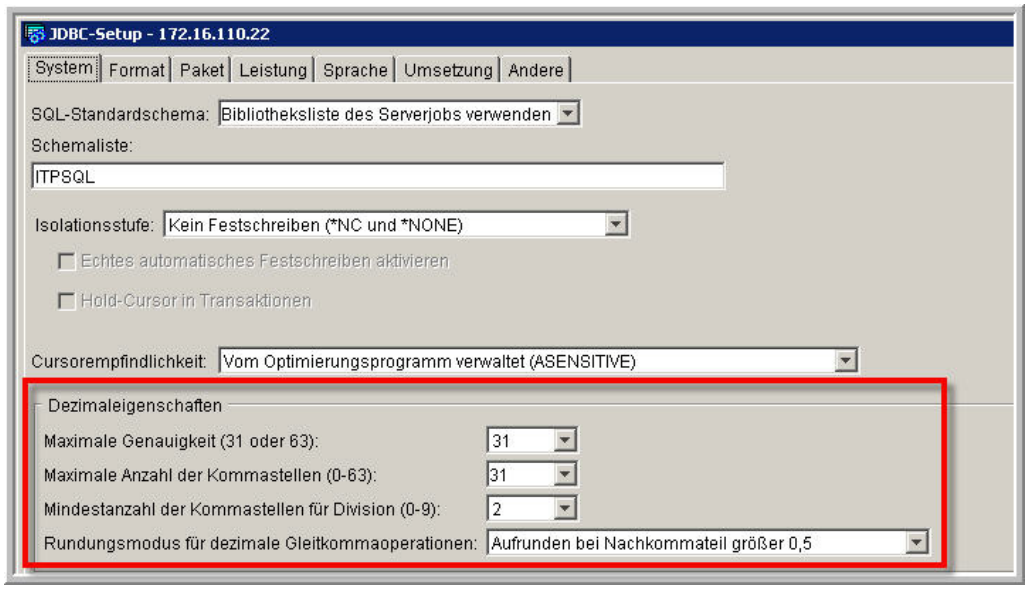
Vergleichen wir einmal unsere beiden Ergebnisse:

Auftragswert(Netto)	Durchschnittl. Auftragswert	AVG: Durchschnittl. Auftragswert
55544.50	1291.73	1291.732558139534883720930 2

Wirkungsweise der Aggregatfunktion AVG

Unser „manuell“ berechneter Durchschnittswert hat zwei Nachkommastellen. Die Anzahl der Nachkommastellen wird hierbei durch die beteiligten Operatoren bestimmt – in unserem Beispiel bestimmt die Definition der Spalte AUFTRAGSWERT (DEC 11, 2) also die Anzahl der Nachkommastellen im Ergebnis.

Anders die Funktion AVG. Hier entscheidet die Konfiguration der JDBC-Schnittstelle des System i Navigators, mit wie vielen Nachkommastellen gerechnet wird. Die JDBC-Schnittstelle bestimmt im Übrigen auch, wie bei internen Rechenoperationen zu runden ist:



Das Interface hat Einfluss auf interne Rechenoperationen

Wenn Sie das JDBC-Setup-Register „System“ öffnen und dort weniger Dezimaleigenschaften angezeigt werden als in der oberen Abbildung, dann ist es vielleicht an der Zeit, System i Navigator zu aktualisieren?

Kommen wir wieder zu unseren Aggregatfunktionen zurück. Mit der Funktion MAX lässt sich der größte Wert einer Spalte ermitteln und die Funktion MIN liefert – wie nicht anders zu erwarten – das kleinste Element einer Spalte. Finden wir doch einmal heraus, welches der günstigste und welches der höchste Artikelpreis in unserem Sortiment ist:

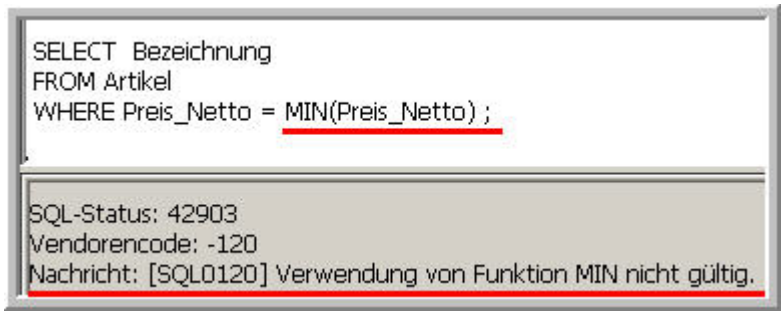
```
SELECT MAX(preis_netto) as "Max_Preis",
MIN(Preis_Netto) as "Min_Preis"
FROM Artikel;
```

Max_Preis	Min_Preis
145.00	1.25

Die Aggregatfunktionen MIN und MAX

7.6.1
Seite 34

Wir wissen jetzt, dass der teuerste Artikel 145,00 Euro und der günstigste Artikel 1,25 Euro kostet. Können wir herausfinden, welcher Artikel am günstigsten ist?



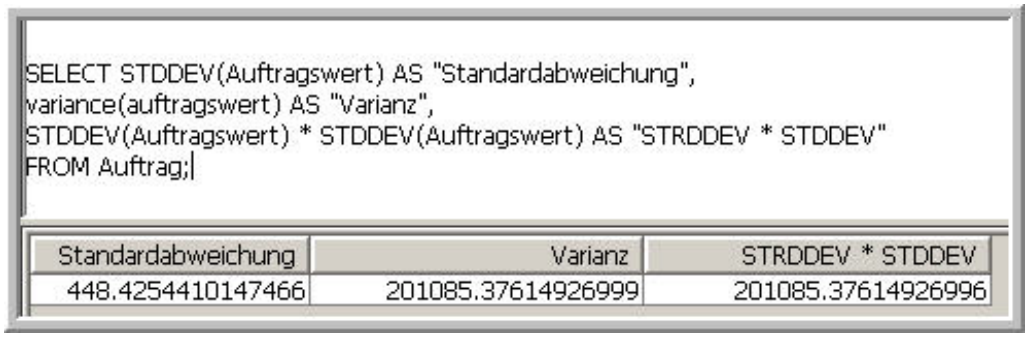
Ungültige Verwendung der Aggregatfunktion

Leider funktioniert das nicht. Die Fehlermeldung erinnert daran, dass die Aggregatfunktionen nicht in einer WHERE-Klausel verwendet werden dürfen. Einen Ausweg lernen wir im nächsten Beitrag kennen.

Die Gruppenfunktion STDDEV berechnet die Standardabweichung einer Zahlenspalte. Lassen Sie uns die Standardabweichung unserer Aufträge ermitteln:

```
SELECT    STDDEV(Auftragswert)
FROM      Auftrag;
```

Wir haben in unserem Auftragsbestand eine Standardabweichung von 448,42544 Euro. Bildet man das Quadrat der Standardabweichung, erhält man die Varianz oder die Streuung einer Zufallsgröße. Die Varianz ist das Quadrat der Standardabweichung oder die mittlere quadratische Abweichung. Beide Werte verwendet man vor allem in statistischen Berechnungen.



Die Funktionen VARIANCE und STDDEV

Es versteht sich von selbst, dass Sie die Statistikfunktionen STDDEV und VARIANCE ausschließlich auf numerische Spalten anwenden dürfen.

Hinter diesen Statistikfunktionen verbergen sich natürlich entsprechende Berechnungen. Hier ein Beispiel, wie die Varianz bzw. Standardabweichung mathematisch ermittelt wird:

Gehen wir davon aus, dass die Spalte Auftragswert die Werte 100, 130 und 160 enthält. Zunächst wird der Durchschnittswert ermittelt, also der Wert 130. Für jeden Wert der Tabelle wird anschließend die Differenz zum Durchschnittswert ermittelt. Wir erhalten die Werte -30, 0 und 30. Diese Werte werden quadriert. So ergeben sich die Werte 900, 0 und 900. Hieraus wird die Summe gebildet und durch die Anzahl der Werte dividiert. Wir erhalten das Ergebnis 600 als Varianz. Die Wurzel aus 600 ergibt die Standardabweichung in Höhe von 24,494897.

7.6.1.3.2 Skalare Funktionen

Skalare Funktionen bieten uns die Möglichkeit, die Inhalte der Felder oder Spalten eines bestehenden Datensatzes zu verarbeiten. Diese Funktionen können nicht nur in Verbindung mit einer SELECT-Anweisung oder HAVING-Option zum Einsatz kommen, sondern auch als Teil einer WHERE-Abfrage oder zum Join-ähnlichen Zugriff auf die Inhalte mehrerer Dateien. Darüber hinaus lassen sich skalaren Funktionen auch in Verbindung mit GROUP BY oder ORDER BY einsetzen. Allgemein können wir sagen, dass in jedem Fall, in dem ein Feldname oder ein Wert angegeben werden kann, die Angabe einer skalaren Funktion möglich ist.

Die DB2/UDB der iSeries unterstützt derzeit mehr als 150 solcher skalaren Funktionen, die wir für die Kontrolle jedes Feldtyps – von numerischen Feldern über Zeichenketten bis hin zu Datumsfeldern – einsetzen können. Einige dieser Funktionen werden Ihnen sicher bekannt vorkommen. Das ist nicht verwunderlich, da einige der unterstützten skalaren Funktionen in leicht abgewandelter Form auch in anderen Programmiersprachen verfügbar sind.

IBM bietet uns verschiedene Werkzeuge, mittels derer wir die SQL-Funktionen erstellen können. Weit verbreitet ist in diesem Zusammenhang der Einsatz des System i Navigators. Dort hat man versucht, die skalaren SQL-Funktionen in Bereiche zusammenzufassen. Einige dieser Bereiche sind:

- Skalarumsetzungsfunktionen
- Datenverbindungsskalarfunktionen
- Skalarfunktionen für Datum/Uhrzeit
- Partitionierungsskalarfunktionen
- Verschiedene Skalarfunktionen
- Numerische Skalarfunktionen
- Zeichenfolgeskalarfunktionen

Nachfolgend ein Auszug aus den zur Verfügung stehenden skalaren Funktionen. Hier sei der Hinweis gestattet, dass die Auflistung NICHT vollständig ist. IBM erweitert die Funktionen meist mit neuen Versionen des IBM i. Deshalb gilt als Empfehlung, dass man bei der Suche nach einer geeigneten skalaren Funktion stets das Informationscenter des genutzten Release durchsuchen sollte.

SQL-Funktionen – Übersicht:

- ABS
  liefert den absoluten Wert einer Nummer.
- ACOS
  liefert den Arcuscosinus des Arguments als Winkel angegeben in einem Bogenmaß.
- ADD_MONTHS
  Mit ADD_MONTHS erhalten wir einen Datumswert oder eine Zeitmarke in der Form des Ausdrucks plus dem Wert des Monats.
- ANTILOG
  gibt den Antilogarythmus (Basis 10) einer Nummer zurück.
- ASCII
  liefert einen ASCII-Code-Wert des linken Zeichens als Integer.
- ASIN
  gibt den Winkel im Bogenmaß (Radiant) zurück, dessen Sinus dem angegebenen Float-Ausdruck entspricht.
- ATAN
  gibt den Winkel im Bogenmaß zurück, dessen Tangens der angegebene Float-Ausdruck ist. Dies wird auch als Arkustangens bezeichnet.
- ATANH(number)
  liefert den hyperbolischen Arcustangens von number. Die Werte für number sind nicht eingeschränkt.
- ATAN2(number, nbr)
  liefert den Arcustangens von number und nbr. Die Werte für number und nbr sind nicht eingeschränkt, aber die Ergebnisse liegen zwischen -1 und 1 und werden in Bogenmaß ausgedrückt. ATAN2(x,y) ähnelt ATAN(y/x), nur mit dem Unterschied, dass die Vorzeichen von x und y über den Quadrant des Resultats entscheiden.
- BIGINT(expression)
  liefert eine 64-Bit-Integer-Darstellung von expression, wobei expression eine Zahl, ein String, ein Datum, eine Uhrzeit oder ein Zeitstempel sein kann.
- BINARY(string)
  konvertiert string in einen Binärstring.
- BIT_LENGTH
  liefert die Länge einer string expression in Bits.
- BLOB(string[,length])
  liefert eine BLOB-Darstellung von string, wobei string eine Stringdarstellung beliebigen Typs sein kann.

Ergänzung 15/2010

7.6.1
Seite 38

- CEILING(expression)
  Die Funktionen CEIL oder CEILING liefern den kleinsten Integer-Wert, der größer oder gleich expression ist.
- CHAR(expression[,length])
  liefert eine Stringdarstellung fester Länge von expression, wobei der optionale Parameter length das Längenattribut des Rückgabezeichenwerts ist.
- CHARACTER_LENGTH(expression)
  Die Funktionen CHARACTER_LENGTH oder CHAR_LENGTH liefern als Rückgabewert die Länge von expression.
- CHR(number)
  liefert das Zeichen mit dem ASCII-Codewert number.
- CLOB(string[,length])
  liefert eine CLOB-Darstellung von string.
- COALESCE(expression[,length])
  liefert das erste Argument, das nicht NULL ist.
- COMPARE_DECFLOAT
  liefert eine Sortierung für DECFLOAT Werte.
- CONCAT(string1, string2)
  liefert eine Verkettung von string1 mit string2 und ist äquivalent zum Verkettungsoperator (||).
- CONTAINS
  sucht einen Textsuchindex anhand der mit dem Suchargument angegebenen Kriterien. Das Ergebnis liefert dann einen Indikator für gefunden bzw. nicht gefunden.
- COS(number)
  liefert den Kosinus von number als Winkel in Bogenmaß.
- COSH(number)
  liefert den hyperbolischen Kosinus von number.
- COT(number)
  liefert den Kotangens von number.
- CURDATE()
  liefert das aktuelle Datum im Format JJJJ-MM-TT oder JJJJMMTT, je nachdem, ob die Funktion in einem String- oder einem numerischen Kontext verwendet wird.
- CURTIME()
  liefert die aktuelle Uhrzeit im Format HH:MM:SS oder HHMMSS, je nachdem, ob die Funktion in einem String- oder einem numerischen Kontext verwendet wird.

- DATABASE
  liefert den Namen der aktuellen Datenbank.
- DATAPARTITIONNAME
  liefert den Namen der Partition, in der eine Zeile gefunden wurde. Wenn das Argument eine nichtpartitionierte Tabelle beinhaltet, wird eine leere Zeichenkette zurückgegeben.
- DATAPARTITIONNUM
  liefert die Data-Partition-Nummer einer Zeile. Wenn das Argument eine nichtpartitionierte Tabelle beinhaltet, wird der Wert 0 zurück gegeben.
- DATE(expression)
  liefert einen Datumswert von expression. Wenn expression ein Integerwert ist, ist das zurückgelieferte Datum expression-1 Tage nach dem 1. Januar 0001.
- DAY(expression)
  liefert die Tageszahl in einem Datumsausdruck.
- DAYNAME(expression)
  liefert den Tagesnamen in expression.
- DAYOFMONTH(expression)
  liefert einen Integerwert zwischen 1 und 31, der den Tag des Monats darstellt.
- DAYOFWEEK (expression)
  liefert den Tag der Woche bzw. des Jahres in expression. DAYOFWEEK gibt einen Integerwert zwischen 1 und 7 zurück. Die Woche beginnt stets mit 1 = Sonntag.
- DAYOFWEEK_ISO(expression)
  DAYOFWEEK_ISO hat denselben Wertebereich wie DAYOFWEEK. Abweichend zu DAYOFWEEK ist der Tag 1 = Montag.
- DAYOFYEAR(date)
  liefert den Tag des Jahres für date im Wertebereich 1 bis 366. 1 ist der 1. Januar.
- DAYS(expression)
  liefert die Anzahl der Tage zwischen expression und dem 1. Januar 0001 + 1.
- DBCLOB(expression[,length])
  liefert eine DBCLOB-Darstellung eines grafischen Stringtyps. length gibt die Länge des zurückgegebenen DBCLOB-Werts an.
- DBPARTITIONNAME
  liefert den Namen der relationalen Datenbank.

7.6.1
Seite 40

- DBPARTITIONNUM(column)
  liefert die Datenbankpartitionsnummer der Zeile, die column enthält.
- DECFLOAT
  liefert einen Fließkommawert.
- DECFLOAT_SORTKEY
  liefert einen binären Wert der zum Sortieren der DECFLOAT-Werte genutzt werden kann.
- DECIMAL oder DEC (expression)
  liefert eine Dezimaldarstellung von expression.
- DECRYPT_BIT, DECRYPT_BINARY, DECRYPT_CHAR und DECRYPT_DB(data[,password])
  liefern entschlüsselte Daten unter Zuhilfenahme des optionalen passwords. Das Ergebnis von DECRYPT_BIN ist ein VARCHAR FOR BIT DATA und das von DECRYPT_CHAR ein VARCHAR.
- DEGREES(number)
  gibt das Ergebnis einer Konvertierung des in Bogenmaß ausgedrückten Arguments in Grad an.
- DIFFERENCE(expression1, expression2)
  liefert die Klangdifferenz zwischen zwei Strings anhand ihrer SOUNDEX-Werte. Das Ergebnis ist ein Integer zwischen 0 und 4, wobei 4 die beste SOUNDEX-Übereinstimmung bedeutet.
- DIGITS(number)
  liefert das number-Argument, konvertiert in einen Ziffernstring. Das Ergebnis von DIGITS enthält keine Dezimalzeichen oder Vorzeichen und kann links mit Nullen ausgepolstert werden.
- DLCOMMENT
  liefert, wenn vorhanden, den Kommentarwert einer DataLink-Wertes.
- DLLINKTYPE
  liefert den Link Type eines DataLink-Wertes.
- DLURLCOMPLETE
  liefert den vollständigen URL-Wert eines DataLink-Wertes mit Link Type URL. Dieser Wert ist derselbe, der durch Concat von DLURLSCHEME mit '://', DLURLSERVER und DLURLPATH gebildet werden kann. Wenn der DataLink über ein Attribut FILE LINK CONTROL und READ PERMISSION DB verfügt, dann beinhaltet der Wert einen File Access Token.
- DLURLPATH
  liefert den Pfad und den Dateinamen, die für den Zugriff auf eine Datei von einem DataLink-Wert mit einem Link Type URL erforderlich sind. Der Wert beinhaltet einen File Access Token.

- DLURLPATHONLY
  liefert den Pfad und den Dateinamen, die für den Zugriff auf eine Datei von einem DataLink-Wert mit einem Link Type URL erforderlich sind. Der Wert beinhaltet NIE einen File Access Token.
- DLURLSCHEME
  liefert die Übersicht eines DataLink-Wertes mit einem Link Type URL. Der Wert wird immer in Großbuchstaben ermittelt.
- DLURLSERVER
  liefert den File Server eines DataLink-Wertes mit einem Link Type URL. Der Wert wird immer in Großbuchstaben zurückgegeben.
- DLVALUE
  liefert immer einen DataLink-Wert. Wird die Funktion rechts von SET in einer UPDATE-Anweisung oder innerhalb von VALUES in einer INSERT-Anweisung angebenen, dann wird gewöhnlich auch ein Link zu einer Datei erzeugt.
- DOUBLE_PRECISION oder DOUBLE(number)
  liefert number, konvertiert in DOUBLE.
- ENCRYPT_AES
  liefert einen Wert, der das Ergebnis der Verschlüsselung data-string mit dem AES-Verschlüsselungsalgorithmus darstellt.
- ENCRYPT_RC2
  liefert einen Wert, der das Ergebnis der Verschlüsselung mit dem RC2-Verschlüsselungsalgorithmus darstellt.
- ENCRYPT_TDES
  liefert einen Wert, der das Ergebnis der Verschlüsselung mit dem Triple DES (TDES) Verschlüsselungsalgorythmus darstellt.
- EXP
  liefert einen Wert, der auf dem natürlichen Logarithmus basiert.
- EXTRACT
  ermittelt den Datumsteil eines angegebenen Wertes.
- FLOAT(number)
  liefert einen Fließkommawert von number.
- FLOOR(expression)
  liefert den größten Integerwert, der kleiner oder gleich expression ist. Rundet einen nichtganzzahligen Wert zum nächstkleineren Integer ab. Integerwerte werden unverändert zurückgegeben.
- GENERATE_UNIQUE
  liefert einen Bit-Zeichenwert mit 13 Bytes Länge (CHAR(13) FOR BIT DATA). Die Funktion ist als Not Deterministic definiert.

Ergänzung 15/2010

7.6.1
Seite 42

- GETHINT(data)
  Liefert den Passworthinweis, der in data hineincodiert ist. Das Argument data muss mit der ENCRYPT-Funktion verschlüsselt sein.
- GRAPHIC(expression[,length])
  liefert eine grafische Stringdarstellung fester Länge von expression, wobei das optionale Argument length die Länge des Ergebnisses ist.
- HASH
  liefert die Partitionsnummer eines Wertesets.
- HASHED_VALUE
  liefert die Partition-Map-Index-Nummer einer Zeile.
- HEX(expression)
  liefert eine Hexadezimaldarstellung von expression.
- HOUR(expression)
  liefert den Stundenteil des Zeitwerts in expression.
- IDENTITY_VAL_LOCAL()
  liefert den Wert, der einer Identitätsspalte als letzter zugewiesen wurde, und NULL, wenn seit der letzten COMMIT- oder ROLLBACK-Anweisung keine Identitätsspalte in eine Tabelle eingefügt wurde.
- IFNULL(expression 1, expression 2)
  liefert expression1, wenn expression1 nicht NULL ist. Andernfalls wird expression2 zurückgegeben.
- INSERT(expression1, expression2, expression3, expression4)
  liefert expression1 mit expression4, wobei expression3 Bytes an Position expression2 ersetzt werden.
- INTEGER oder INT(expression)
  liefert eine Integerdarstellung von expression.
- JULIAN_DAY(expression)
  liefert die Tageszahl des Datumswerts in expression nach dem julianischen Kalender. Der Rückgabewert ist die Anzahl der Tage zwischen dem Datumswert in expression und dem Beginn des julianischen Kalenders, dem 1. Januar 4713 v. Chr.
- LAST_DAY(expression)
  liefert einen Datumswert oder eine Zeitmarke für den letzten Tag des Monats.
- LCASE
  liefert einen Zeichensatz, in dem alle Zeichen in Kleinbuchstaben umgesetzt wurden.
- LEFT(string, length)
  liefert die ersten length Bytes von string.

- LENGTH(expression)
  liefert die Länge von expression als Integer oder NULL, wenn expression NULL ist.
- LN
  liefert den natürlichen Logarithmus einer Nummer.
- LOCATE(substring, string [,starting_pos])
  liefert den Fundort von substring in string oder null, wenn der Teilstring im string nicht gefunden wird. Die optionale starting_pos kann die Startposition in string angeben, bei der die Suche begonnen wird.
- LOG10(number)
  liefert den Basis-10-Logarithmus von number.
- LOWER
  liefert einen Wert, in dem alle Zeichen in Kleinbuchstaben umgesetzt wurden.
- LTRIM(string)
  entfernt die Leerstellen am Anfang von string.
- MAX
  Mit der Funktion MAX erhalten wir den höchsten Wert in einem Set von Werten.
- MICROSECOND(expression)
  liefert den Mikrosekundenanteil des Uhrzeitwerts in expression.
- MIDNIGHT_SECONDS(expression)
  gibt an, wie viele Sekunden zwischen Mitternacht und dem Uhrzeitwert in expression verstrichen sind.
- MIN
  die Funktion MIN liefert den niedrigsten Wert eines Wertesets.
- MINUTE(expression)
  liefert den Minutenbestandteil des Zeitwerts in expression.
- MOD
  dividiert das erste Argument durch das zweite Argument.
- MONTH(expression)
  liefert den Monatsbestandteil des Datumswerts in expression.
- MONTHNAME(expression)
  liefert für den Datumswert aus expression den Monatsbestandteil in der aktuellen Spracheinstellung der Datenbank.
- MONTHS_BETWEEN(date1, date2)
  liefert die Anzahl der Monate zwischen dem Datum date1 und dem Datum date2. Wenn date1 hinter date2 liegt, ist das Ergebnis positiv, ansonsten negativ.

7.6.1
Seite 44

- MULTIPLY_ALT(number1)
  liefert das Produkt von number1 und number2. Die Funktion MULTIPLY_ALT ist eine gute Alternative zu dem Standardmultiplikationsoperator (*), wenn die zu multiplizierenden Zahlen eine Genauigkeit größer als 63 haben.
- NEXT_DAY(date, string)
  liefert das Datum des ersten von string angegebenen Wochentags, der hinter dem date liegt. Das Argument string muss entweder der vollständige Name oder die Abkürzung eines Wochentags sein, und zwar in der für die Session gültigen Spracheinstellung für Datumswerte.
- NORMALIZE_DECFLOAT
  liefert einen DECFLOAT-Wert.
- NOW()
  gibt einen zeitmarkenbasierten Wert zurück. Dabei wird die aktuelle Zeit/Datum ermittelt.
- NULLIF(expression1, expression 2)
  liefert NULL, wenn expr1 gleich expr2; andernfalls wird expr1 zurückgegeben.
- OCTET_LENGTH
  liefert einen Integerwert, der die Anzahl der Oktette in einem anderen Wert angibt.
- PI
  liefert den Wert von ϖ 3.141592653589793.
- POSITION
  gibt einen Integer zurück, der die Startposition eines Strings im Suchstring angibt.
- POSSTR(source, search)
  liefert die Zeichenposition des ersten Auftretens von search in source, wobei das erste Zeichen die Position 1 hat.
- POWER
  liefert den Wert von X hoch Y.
- QUARTER(expression)
  liefert einen Integer zwischen 1 und 4, der angibt, auf welches Quartal des Jahres das Datum in expression fällt.
- RADIANS(expression)
  gibt das Ergebnis einer Konvertierung der in Grad ausgedrückten expression in Bogenmaß an.

- RAISE_ERROR(sqlstate, errorstring)
  löst bei Ausführung einen Fehler aus. Diese Funktion ist nützlich, um in komplizierten SQL-Anweisungen und gespeicherten Prozeduren wichtige Fehler auszulösen. Der Parameter sqlstate gibt Fehlercode-Informationen an die Anweisung zurück, welche die SQL-Anweisung ausführt; der Parameter errorstring kann eine benutzerdefinierte Fehlermeldung aufnehmen.
- RAND([seed])
  liefert einen willkürlichen Fließkommawert zwischen 0 und 1, wobei seed der Seedwert für den Zufallszahlengenerator ist.
- REAL(number)
  liefert eine Fließkommadarstellung von number mit einfacher Genauigkeit.
- REPEAT(string, number)
  liefert einen Zeichenstring, in dem sich der string-Wert number Male wiederholt.
- REPLACE (string, search_string, replacement_string)
  Im String wird jedes Auftreten des search_string durch replacement_string ersetzt.
- RID
  Die Funktion liefert als Rückgabewert die relative Satznummer einer Zeile als BIGINT.
- RIGHT(string, number)
  liefert die number-Anzahl der am weitesten rechts stehenden Bytes von string.
- ROUND(number[, decimal])
  liefert number, gerundet auf decimal Nachkommastellen. Wird decimal weggelassen, wird number auf einen Integer gerundet. Integer decimal kann auch negativ sein. In einem solchen Fall werden die Stellen vor dem Komma oder Dezimalpunkt gerundet.
- ROWID
  liefert eine Zeichenfolge zu einer ROW ID.
- RRN
  liefert die relative Satznummer einer Zeile.
- RTRIM(string)
  schneidet vom string-Ende alle Leerzeichen ab.
- SCORE
  sucht einen Textsuch-Index unter Verwendung der in dem Suchargument angegebenen Kriterien.

7.6.1
Seite 46

- SECOND(expression)
  liefert den Sekundenanteil des Uhrzeitwerts in expression.
- SIGN(number)
  liefert für number < 0 den Wert -1, für number = 0 den Wert 0 und für number > 0 den Wert 1.
- SIN(number)
  liefert den Sinus von number, wobei number in Bogenmaß angegeben ist.
- SINH(number)
  liefert den hyperbolischen Sinus von number.
- SMALLINT(number)
  liefert einen SMALLINT-Wert äquivalent zu number.
- SOUNDEX(string)
  liefert einen Zeichenstring mit der phonetischen Darstellung von string. Diese Funktion macht einen Gleichheitsvergleich zwischen englischen Wörtern möglich, die unterschiedlich geschrieben werden, aber ähnlich klingen.
- SPACE(number)
  liefert einen String von number-Leerzeichen.
- SQRT
  berechnet die Quadratwurzel einer Zahl.
- STRIP
  entfernt Leerstellen oder andere spezifische Zeichen vom Ende und/oder Anfang einer expression.
- SUBSTR(string, start [, length])
  Liefert einen Teilstring von string, der length Bytes lang ist und bei der start-Position beginnt. Wenn length zu lang ist, um einen gültigen Teilstring zu definieren, wird der string mit Leerzeichen am Ende so weit aufgefüllt, bis ein gültiger Teilstring ermittelt werden kann. Die Startposition „start“ muss ein Integerwert zwischen 1 und der Länge von string sein.
- SUBSTRING
  liefert eine Unterzeichenfolge.
- TAN(number)
  liefert den Tangens von number, wobei number in Bogenmaß angegeben ist.

- TANH(number)
  liefert den hyperbolischen Tangens von number.
- TIME(expression)
  liefert einen Uhrzeitwert äquivalent zu der Uhrzeit im Ausdruck expression.
- TIMESTAMP (expression1 [,expression2])
  liefert einen Zeitstempelwert, der aus den Datums- und Uhrzeitwerten aus expression1 und expression2 erstellt wird. Wenn nur expression1 angegeben wurde, muss dieser Ausdruck einen gültigen Datums- und Uhrzeitwert enthalten. Sind beide Argumente angegeben, muss expression1 eine gültige Stringdarstellung eines Datums und expression2 eine gültige Stringdarstellung einer Uhrzeit enthalten.
- TIMESTAMP_FORMAT(string, format)
  liefert einen Zeitstempelwert, der aus string extrahiert wurde. Das zweite Argument, format, enthält das Zeitstempelformat, welches in string benutzt wird.
- TIMESTAMP_ISO(expression)
  liefert einen aus expression angelegten Zeitstempelwert. Wenn expression ein Datumswert ist, besteht der Uhrzeitwert aus Nullen. Wenn expression ein Uhrzeitwert ist, wird bei der Erstellung des Zeitstempelwerts das aktuelle Datum zugrunde gelegt.
- TIMESTAMPDIFF (interval_type, tsdiff_expression)
  liefert eine geschätzte Anzahl von Intervallen zwischen zwei Zeitstempelwerten. Das Argument interval_type muss eine Zweierpotenz zwischen 1 und 256 sein.

  Dabei gelten die folgenden Werte:
  1 Millisekunden
  2 Sekunden
  4 Minuten
  8 Stunden
  16 Tage
  32 Wochen
  64 Monate
  128 Quartale
  256 Jahre
- TOTALORDER
  liefert eine Sortierung der DECFLOAT-Werte.

7.6.1
Seite 48

- TRANSLATE (string, tostring, fromstring [,pad_char])
  liefert zu dem string-Argument einen äquivalenten String, bei dem jedes in tostring vorkommende Zeichen durch ein entsprechendes Zeichen aus fromstring ersetzt wurde. Das optionale Argument pad_char kann ein Einzelbytezeichen enthalten, mit dem tostring ausgepolstert wird, wenn er kürzer als fromstring ist. Wenn dasselbe Zeichen in fromstring mehrmals auftaucht, zählt nur das erste Vorkommen, und alle weiteren werden ignoriert.
- TRIM(string)
  entfernt Leerstellen vom Anfang und/oder Ende einer Zeichenkette.
- TRUNCATE (number1, number2) oder TRUNC (number1, number2)
  liefert number1, wobei alle Zeichen, die um die mit number2 angegebenen Stellen rechts vom Dezimalpunkt stehen, durch Nullen ersetzt werden. Wenn number2 negativ ist, werden entsprechend Ziffern links vom Dezimalpunkt gekappt.
- UCASE(str)
  liefert eine Zeichenfolge, in der alle Zeichen in Großbuchstaben umgesetzt wurden. Die Umsetzung erfolgt gemäß der verwendeten CCSID.
- UPPER
  ersetzt alle Zeichen in einer Zeichenfolge durch Großbuchstaben. Es wird die CCSID genutzt. Es werden nur SBCS- and Unicode-Zeichen umgesetzt. Die Zeichen a-z werden in A-Z umgesetzt.
- VALUE
  liefert den Wert der ersten Nichtnull-expression.
- VARBINARY
  liefert einen VARBINARY-Wert.
- VARCHAR (expression[,length])
  liefert eine Zeichendarstellung variabler Länge von expression, wobei length das Längenattribut für das Ergebnis ist. Wird length weggelassen, entspricht die Länge des Ergebnisses der Länge des Ausdrucks expression.
- VARCHAR_FORMAT (expression, format)
  liefert eine Stringdarstellung des Zeitstempelwerts aus expression in dem format, welches durch das format-Argument vorgegeben wird. Dazu sind die folgenden Werte gültig:
  YYYY Jahresangabe
  MM Monatsangabe
  DD Tagesangabe
  HH24 Stundenangabe
  MI Minutenangabe
  SS Sekundenangabe

- VARGRAPHIC(expression [,length])
  liefert eine grafische Stringdarstellung variabler Länge von expression, wobei length das Längenattribut für das Ergebnis ist. Wird length weggelassen, entspricht die Länge des Ergebnisses der Länge des Ausdrucks.
- WEEK(expression)
  liefert die Woche des Jahres für den Datumswert aus expression. Die Ergebnisse liegen zwischen 1 und 54; erster Tag der Woche ist der Sonntag. Der 1. Januar ist immer in der ersten Woche.
- WEEK_ISO(expression)
  liefert die Woche des Jahres für den Datumswert aus expression. Die Ergebnisse liegen zwischen 1 und 53; erster Tag der Woche ist der Montag. Die erste Woche des Jahres enthält immer den 4. Januar.
- YEAR(expression)
  liefert den Jahresbestandteil des Datums aus expression.
- ZONED
  liefert eine gezonte dezimale Darstellung.

Dies war ein Überblick zu den SQL-Funktionen. Wie bereits erwähnt, erhebt diese Auflistung keinen Anspruch auf Vollständigkeit.

Ergänzung 15/2010

7.6.1.4 Daten gruppieren

Sehr häufig muss man die Daten im Result Set gruppieren, insbesondere im Zusammenhang mit den in Kapitel 7.6.1.3.1 vorgestellten Aggregatfunktionen. Mit der folgenden Abfrage haben wir unseren Auftragswert ermittelt:

```
SELECT    sum(auftragswert)
FROM      Auftrag;
```

Wie können wir nun herausfinden, welche Umsätze mit den einzelnen Kunden erzielt wurden?

```
SELECT    Kundennummer, sum(auftragswert)
FROM      Auftrag;
```

In der SELECT-Klausel steht eine normale Spaltenauswahl Kundennummer gefolgt von der Aggregatfunktion SUM(AUFTRAGSWERT). Diese Abfrage wird nicht ausgeführt, weil SQL die Kombination der normalen Spalte mit der Aggregatfunktion bemängelt. Diese Bedingung erfordert eine GROUP-BY-Klausel. Wir müssen unsere Daten nach der Kundennummer gruppieren, so dass die Aggregatfunktion anschließend auf die einzelnen Gruppen angewandt werden kann.

```
SELECT    Kundennummer, sum(auftragswert)
FROM      Auftrag
GROUP BY  Kundennummer;
```

Die Klausel GROUP BY führt jetzt die in der SELECT-Anweisung angegebene Aggregatfunktion für jede Gruppierung der Spalte aus, die nach der Klausel GROUP BY steht. Wir können jetzt natürlich weitere Aggregatfunktionen nutzen und beispielsweise die Anzahl der Aufträge ermitteln.

```
SELECT    Kundennummer, SUM(auftragswert), COUNT(*)
FROM      Auftrag
GROUP BY  Kundennummer;
```

```
SELECT      Kundennummer, SUM(auftragswert) as "Umsatz je Kunde" , COUNT(*) as "Anzahl Aufträge"
FROM        Auftrag
GROUP BY    Kundennummer;
```

KUNDENNUMMER	Umsatz je Kunde	Anzahl Aufträge
00008	4200.00	3
00003	3322.50	3
00400	3795.00	3

Wirkungsweise der Klausel GROUP BY

Sie müssen im Übrigen die Gruppierungsspalten nicht zwingend in der Spaltenliste aufführen. Allerdings dürfen Sie nicht Spalten in der Spaltenliste anfordern, die nicht in der Group By-Klausel aufgeführt sind. Die folgende Abfrage kann und wird SQL daher auch nicht ausführen:

```
SELECT    Kundennummer, Auftragsdatum, SUM(auftragswert),
          COUNT(*)
FROM      Auftrag
GROUP BY  Kundennummer;
```

Warum nicht? Versetzen Sie sich doch einmal in die Lage von SQL. Nehmen wir an, Sie sollen die folgenden Zeilen gruppieren und dabei die Summe und die Anzahl Aufträge für die Kundennummer 00001 bilden:

Kundennummer	Auftragsdatum	Auftragswert
00001	01.08.2010	100,00 Euro
00001	06.07.2010	125,00 Euro
00001	12.08.2010	53,00 Euro

Wenn der Benutzer verlangt, dass Sie alle drei Spalten ausgeben sollen, aber nur nach der Kundennummer gruppieren dürfen, wo bringen Sie dann das eindeutig unterscheidbare Auftragsdatum unter?

Kundennummer	Auftragsdatum	Auftragswert
00001	?? 01.08.2010 ??	278,00 Euro
	?? 06.07.2010 ??	
	?? 12.08.2010 ??	

Denken Sie daran, dass für die Ausgabe nur eine Zeile pro Gruppe zur Verfügung steht. SQL kann nicht zwei Dinge auf einmal tun und sowohl Detailinformationen in vielen Zeilen ausgeben als auch gleichzeitig einige Werte konsolidieren, indem Summen, Mittelwerte oder andere Dinge generiert werden.

7.6.1.4.1 Daten gruppieren – HAVING

Was aber, wenn Sie, nachdem die Gruppen gebildet wurden, einzelne Gruppen von der Anzeige ausschließen müssen? Sie benötigen nur die Kunden, mit denen Sie mehr als 2.000,00 Euro Umsatz getätigt haben. Die Aggregatfunktionen, das wissen wir bereits, können nicht in der WHERE-Bedingung eingesetzt werden. Hier hilft die Klausel HAVING:

```
SELECT    Kundennummer, SUM(auftragswert), COUNT(*)
FROM      Auftrag
GROUP BY  Kundennummer
HAVING    SUM(auftragswert) > 1000;
```

KUNDENNUMMER	Umsatz je Kunde	Anzahl Aufträge
00008	4200.00	3
00003	3322.50	3
00400	3795.00	3
00007	5250.00	3
00001	2357.00	3
00045	5170.00	3
00101	7050.00	6

Wirkungsweise der Klausel HAVING

Die Klausel HAVING ermöglicht den Einsatz von Aggregatfunktionen in einer Bedingung. Funktioniert HAVING auch mit Nicht-Aggregatfunktionen? Probieren wir die folgende Anweisung aus:

```
SELECT    Kundennummer, SUM(auftragswert), COUNT(*)
FROM      Auftrag
GROUP BY  Kundennummer
HAVING    auftragswert > 1000;
```

```
SQL-Status: 42803
Vendorencode: -119
Nachricht: [SQL0119] Spalte AUFTRAGSWERT oder Ausdruck in der Klausel HAVING nicht gültig.
```

Ausdruck in der Having Klausel ungültig

Die Abfrage wird nicht ausgeführt, da die Spalte AUFTRAGSWERT weder in der GROUP BY-Klausel noch in der Spaltenliste verwendet wird. Hier verhält sich die DB2 übrigens ein klein wenig anders als viele andere Datenbanken.

Wir können uns also merken, dass die HAVING-Klausel nur zum Einsatz kommt, wenn aufgrund der Aggregatfunktionen Gruppen von der Anzeige ausgeschlossen werden. Sollen hingegen einzelne Zeilen von der Gruppierung ausgeschlossen werden, nutzen wir immer die WHERE-Klausel.

7.6.1.5 Daten sortieren

Manchmal sollen die Ergebnisse einer Abfrage in einer bestimmten Reihenfolge erscheinen. Wie Sie wissen, erzeugt die Anweisung SELECT … … FROM lediglich eine Liste. Solange Sie keine Sortierfolge für die Ausgabe bestimmen, erhalten Sie die Ergebnisse der Abfrage in der Reihenfolge, in der die Daten aus der Tabelle geholt wurden. Mit der Klausel ORDER BY lassen sich nun die Ergebnisse in eine bestimmte Reihenfolge bringen. Die folgende Anweisung ordnet zum Beispiel die Kunden in Deutschland nach der Postleitzahl:

```
SELECT
Kdnam1 AS “Kundenname“,
Kdstr AS  “Strasse“,
KDPLZ AS  „PLZ“ ,
KDORT AS  „Wohnort“
FROM      Kunden
WHERE     kdlkz = ‚D‘
ORDER BY  kdplz;
```

Die Daten sind nun in der gewünschten Weise geordnet und erscheinen nicht mehr in der Eingabereihenfolge.

Wirkungsweise der Klausel ORDER BY

7.6.1
Seite 54

Wie lässt sich nun die Liste in umgekehrter Reihenfolge anzeigen – mit der größten Zahl an erster Stelle bzw. in alphabetisch umgekehrter Ordnung? Die Ergebnisliste der folgenden Abfrage ist so geordnet, dass die Postleitzahlen absteigend sortiert werden

Die Klausel ORDER BY DESC

Mit dem Schlüsselwort DESC am Ende der Klausel ORDER BY lässt sich die absteigende Reihenfolge der Datensätze in der Ausgabe festlegen. Die aufsteigende Sortierfolge ist als Standardwert in der Klausel voreingestellt und muss daher nicht ausdrücklich genannt werden. Sie können im Übrigen auch eine Sortierspalte verwenden, die nicht selbst im Result Set enthalten ist. Schauen Sie sich das folgende Beispiel an:

```
SELECT Kdnam1 as Kundenname,
Kdstr AS „Strasse“,
KDPLZ as „PLZ“ ,
KDORT as „Wohnort“

FROM      Kunden
WHERE     kdlkz in ( ‚D‘)
ORDER BY  KDLKZ ;
```

Es wird eine Kundenliste ausgegeben, die nach dem Länderkennzeichen sortiert wird, obwohl dieses Kennzeichen nicht im Result Set enthalten ist. Und noch etwas muss an dieser Stelle erwähnt werden: Sie dürfen in der ORDER BY-Klausel auch Spaltennamen verwenden, die Sie in der SQL-Abfrage zugewiesen haben. Ich könnte also folgende Abfrage problemlos ausführen, obwohl die Spalte KUNDENNAME nicht in der Tabelle KUNDEN vorhanden ist:

```
SELECT Kdnam1 as Kundenname,
Kdstr AS „Strasse“,
KDPLZ as „PLZ“ ,
KDORT as „Wohnort“

FROM      Kunden
WHERE     kdlkz in ( ‚D‘)
ORDER BY  Kundenname ;
```

Dies ist nicht möglich, wenn ich anstelle eines Spaltennamens eine Spaltenüberschrift zuweise, indem ich das Wort KUNDENNAME in doppelte Hochkomma einschließe. Spaltenüberschriften können in der ORDER BY-Klausel nicht verwendet werden. Die Verwendung der temporären Spaltennamen des Result Sets ist insbesondere dann nützlich, wenn Sie temporäre Spalten in der Abfrage aufbauen und anschließend danach sortieren möchten:

```
SELECT Artikelnummer, Bezeichnung, Preis_Netto, Preis_Netto *
1.19 AS Preis_Brutto FROM Artikel

ORDER BY Preis_Brutto;
```

Die Abfrage kann ohne Probleme ausgeführt werden, wie folgende Abbildung zeigt:

ARTIKELNUMMER	BEZEICHNUNG	PREIS_NETTO	PREIS_BRUTTO
00012	Tip Ex Roller	1.25	1.4875
00008	Kabelbinder	4.00	4.7600
00001	Textmarker	4.35	5.1765
00010	Papier (250Blatt)	5.33	6.3427
00002	Radiergummi (10)	5.65	6.7235

Temporäre Spalten in der ORDER BY-Klausel

7.6.1
Seite 56

Aber auch wenn Sie keine Spaltennamen im Result Set verwenden, können Sie nach der temporären Spalte sortieren, indem Sie entweder den gesamten Ausdruck in der ORDER BY-Klausel wiederholen oder einfach die Spalte anhand Ihrer Spaltennummer identifizieren:

```
SELECT  Artikelnummer, Bezeichnung, Preis_Netto, Preis_Netto * 1.19 AS "Preis(Brutto)"
FROM Artikel
ORDER BY Preis_Netto * 1.19;
```

Ausdrücke können in der ORDER BY Klausel verwendet werden.

```
SELECT  Artikelnummer, Bezeichnung, Preis_Netto, Preis_Netto * 1.19 AS "Preis(Brutto)"
FROM Artikel
ORDER BY 4
```

Über die Spaltennummer kann die Sortierspalte

ARTIKELNUMMER	BEZEICHNUNG	PREIS_NETTO	Preis_Brutto
00012	Tip Ex Roller	1.25	1.4875
00008	Kabelbinder	4.00	4.7600
00001	Textmarker	4.35	5.1765

Alternative Benennung einer Sortierspalte

Die alternativen Schreibweisen sind praktisch, denn es kann Situationen geben, in denen Sie Ihren temporären Spalten keine Namen geben können, weil Sie beispielsweise zwingend eine Spaltenüberschrift zuweisen müssen.

Die Klausel ORDER BY lässt sich auch auf mehrere Spalten beziehen. Die folgende Abfrage nimmt zum Beispiel eine Sortierung nach der Warengruppe und innerhalb der Warengruppe nach der Artikelbezeichnung vor. Wobei die Warengruppen absteigend und die Artikelbezeichnungen aufsteigend sortiert werden:

```
SELECT    Artikelnummer, Bezeichnung, Preis_Netto, Warengruppe
FROM      Artikel
ORDER BY Warengruppe desc, Bezeichnung
```

ARTIKELNUMMER	BEZEICHNUNG	PREIS_NETTO	WARENGRUPPE
44552	Aktenvernichter Z100	24.17	1540
10340	Bücherregal	52.00	1540
00100	Schreibtisch	125.00	1540
00200	Schreibtisch Eck	145.00	1540
10000	SCHREIBTISCH	134.90	1540
10100	Schuhschrank	89.90	0004
00012	Tip Ex Roller	1.25	0004
00006	Hefter	7.00	0003

Mehr als eine Sortierspalte verwenden

7.6.2 Wie wird ein SQL-Statement eigentlich verarbeitet?

In den Beispielen war zu erkennen, dass ich keine Unterschiede zwischen SQL-Tabellen und physischen DDS-Dateien gemacht habe; beide können problemlos im SQL-Statement genutzt werden. Nur auf die Verwendung logischer DDS-Dateien müssen Sie verzichten, denn seit dem Release V5R2 haben wir auf einem System-i-Server zwei voneinander unabhängig arbeitende SQL-Interfaces (SQL-Optimizer): die Classic Query Engine (CQE) und die SQL Query Engine (SQE). Die CQE ist alt; sie existiert seit 1989, denn bereits auf der AS/400 existiert diese SQL-Schnittstelle. IBM hat diese Schnittstelle vollständig überarbeitet und modernisiert. Das Ergebnis wurde SQE genannt.

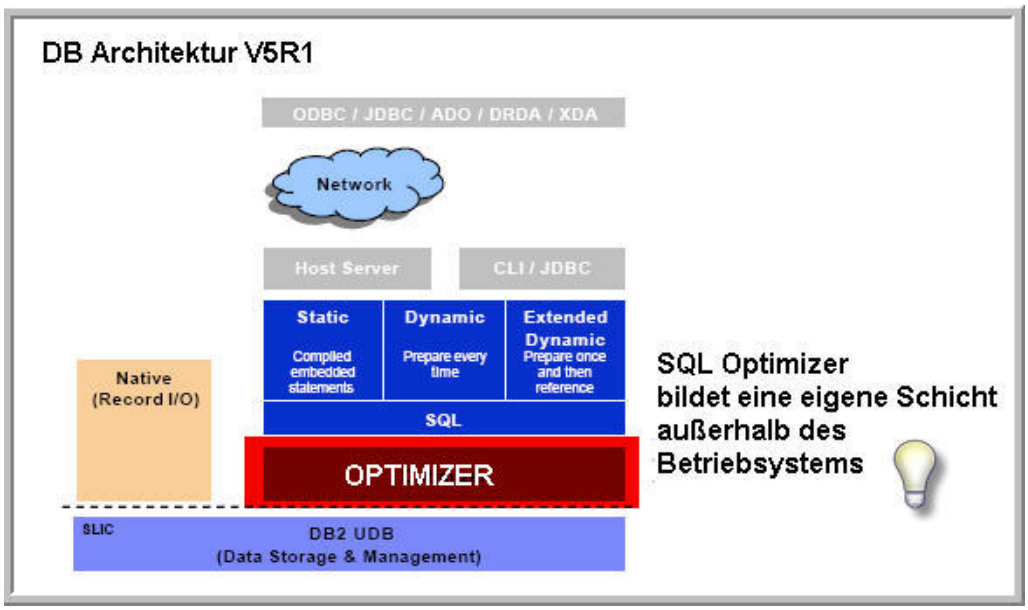
Und damit sind wir auch schon bei der Frage, wie Ihr SQL-Statement ausgeführt wird. Stellen Sie sich vor, dass Sie das folgende SQL-Statement ausführen wollen:

```
SELECT HIKDNR, HIMED
FROM HIST00
WHERE HIKDNR BETWEEN 1 AND 500;
```

SQL ist eine Sprache der vierten Generation, d. h. Sie bestimmen welche Daten benötigt werden ohne festzulegen, welche Zugriffspfade für den Datenabruf genutzt werden. Hierfür ist der SQL-Optimizer zuständig. Der SQL-Optimizer versucht für Ihre SQL Abfrage stets den optimalen Zugriff auf die Daten zu ermitteln. Hierbei arbeitet er kostenbasierend, d. h. er berechnet für unterschiedliche Implementierungen die Ausführungskosten und entscheidet sich aufgrund dieser Berechnungen für die Ausführungsstrategie mit dem vermutlich geringsten Kostenfaktor. Für diesen Entscheidungsprozess spielen einerseits die I/O-Kosten, andererseits die CPU-Kosten eine entscheidende Rolle. Um die Berechnungen durchführen zu können, sammelt der SQL-Optimizer zunächst Informationen über die Tabelle und die SQL Umgebung. Die Anzahl möglicher Tabellensätze, die Größe der Tabelle oder mögliche Indizes spielen hierbei eine entscheidende Rolle. Es wird aber auch die Anzahl der CPUs und die Größe des Arbeitsspeichers ermittelt. Auf Basis dieser Informationen bestimmt der SQL-Optimizer letztendlich den auszuführenden Zugriffsplan.

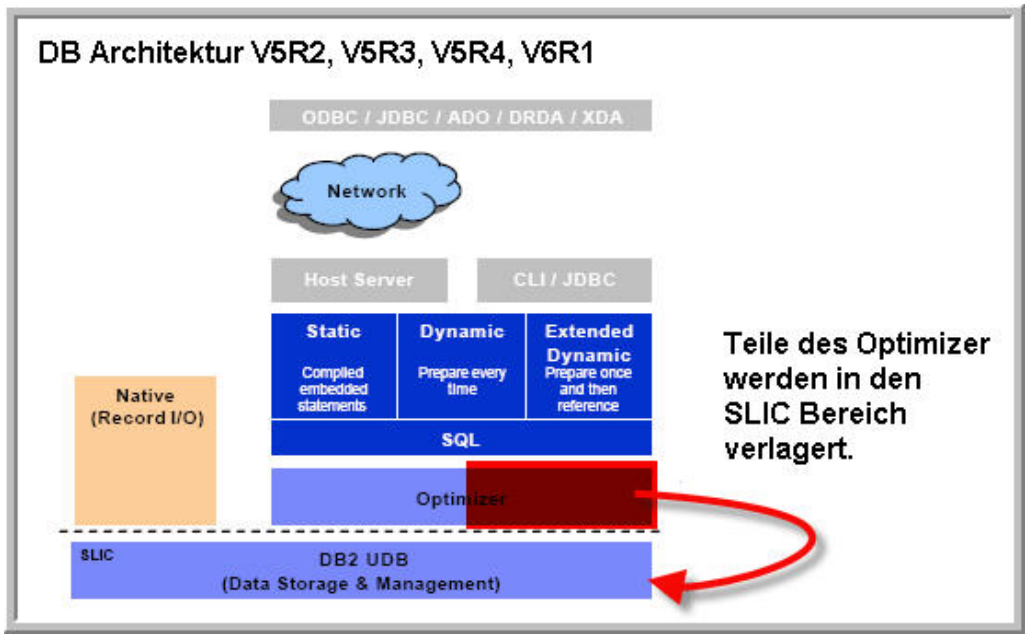
7.6.2
Seite 2

Die Wurzeln der DB2/UDB finden wir aber bereits auf dem IBM System /38. Bereits damals war die Datenbank eine in das Betriebssystem integrierte relationale Datenbank. Die AS/400 übernahm 1988 diese Datenbanktechnologie und unterscheidet sich hierdurch bis heute von allen Konkurrenzprodukten. Neben dem traditionellen nativen Interface implementierte IBM aber auf der AS/400 bereits 1988 ein eigenständiges SQL Interface. Beide Schnittstellen – native Zugriffsmethoden und SQL Zugriffe – existierten nebeneinander und erfüllten hervorragend die Kundenanforderungen der letzten 20 Jahre. Allerdings bildeten der SQL-Optimizer und die SQL Engine bis zum Release V5R2 eine eigene Schicht. Das SQL Interface war somit nicht integraler Bestandteil des Betriebssystems.



DB Architektur vor V5R2

Doch gerade in den letzten Jahren wandelten sich die Anforderungen an moderne Datenbanken. Immer mehr Applikationen nutzten aufgrund neuer Technologien SQL für den Datenbankzugriff. Immer schneller musste die Datenbank neue Anforderungen erfüllen und neue Features enthalten, um am Markt bestehen zu können. Und schließlich kam der Punkt an dem IBM sich langfristig für eine Strategie entscheiden musste, um die DB2/UDB zukunftsfähig zu machen. Die ständige Weiterentwicklung des SQL-Optimizers war keine wirkliche Alternative, denn wie bei vielen „Altanwendungen" dauerte die Implementierung neuer Funktionen viel zu lang. Eine Neuentwicklung hätte die grundlegenden Vorteile der DB2/UDB gegenüber anderen Datenbanken gefährdet, denn die Datenbank des System-i-Servers zeichnete sich jahrelang durch ihre Betriebssystemintegration und der damit verbundenen einfachen Handhabung aus. So entschloss man sich für ein Reengineering und schützte dadurch auch bestehende Applikationen und Funktionen. Insbesondere der SQL-Optimizer war von diesem Prozess betroffen. IBM entwickelte große Teile des Optimizers und Teile der Runtime Engine neu, entschied sich aber den neuen Query Optimizer parallel zum bisherigen Optimizer zu installieren und garantierte dadurch, dass bestehende Applikationen weiterhin problemlos arbeiteten. Gleichzeitig konnten alte und neue Applikationen von der neuen Technologie profitieren. Diese neue Engine ist unter dem Namen SQL Query Engine (SQE) bekannt. Der bestehende Query Optimizer erhielt den Namen Classic Query Engine (CQE). Die Neuentwicklung der SQE veränderte aber auch die DB Architektur als Ganzes. IBM verlagerte nämlich große Teile der SQL Optimierung in den LIC-Bereich, so dass jetzt auch die SQL Engine integraler Bestandteil des Betriebssystems ist.

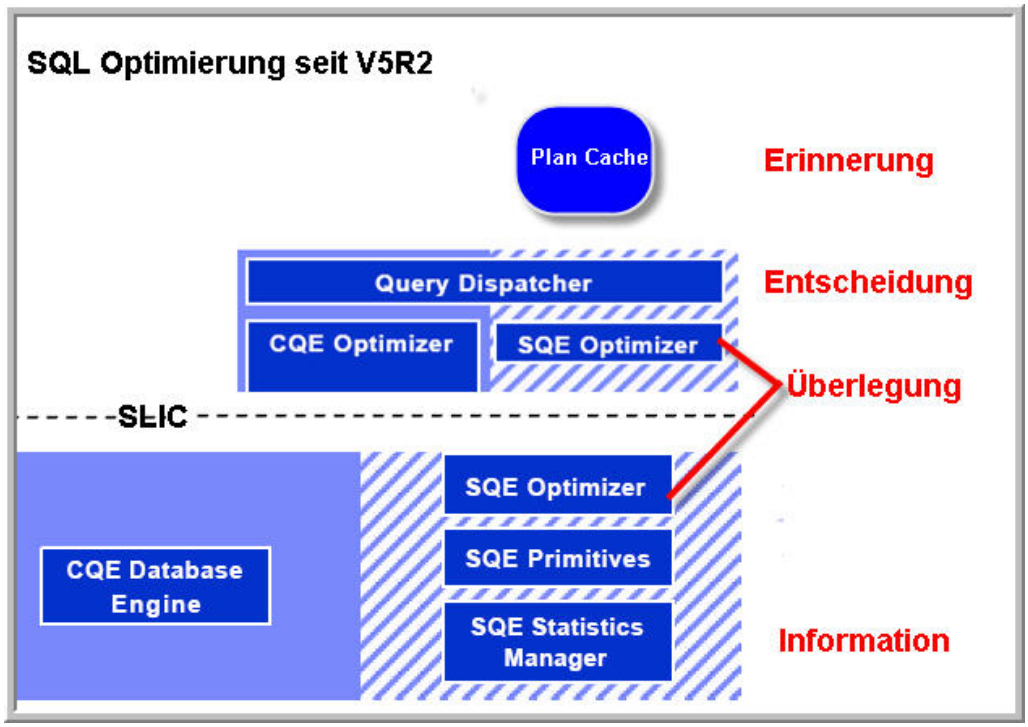


DB Architektur V5R2

7.6.2
Seite 4

Im Release V5R2 wurden zunächst nur lesende Abfragen von der SQE unterstützt. Nach und nach erhöhte sich die Menge der ausgeführten SQL Statements auf Seiten der SQE. Dieser Prozess wird bis zum Release V6R1 weitergeführt.

Aber es wurde nicht nur der Optimierungsprozess in den LIC verlagert, der neue Query Optimizer erhielt auch viele neue, eigenständige Komponenten, die unterschiedliche Aufgaben wahrnehmen:



SQE Komponenten Release V5R2

Beispielsweise liefert der Statistik Manager für den Optimierungsprozess der SQE die benötigten Informationen. Welche Daten der Statistik Manager zur Verfügung stellt, können Sie dem Kapitel 7.4.1.4 entnehmen. Der SQE-Optimizer führt anhand dieser Informationen anschließend die Berechnung unterschiedlicher Zugriffspläne durch und das Ergebnis der Optimierung – der Ausführungsplan – wird im Plan Cache abgelegt. Der Ausführungsplan (Access Plan) enthält alle Informationen über die Daten und Informationen darüber, wie sie gefunden werden. Die gespeicherten Ausführungspläne der SQE können Sie sehen, denn die Anzeige des Plan Cache wurde in den System i Navigator integriert (siehe Kapitel 7.4.4).

Sie sehen, es hat sich viel getan im Release V5R2. Viele neue Features sind hinzugekommen, die eigenständige Aufgaben im Gesamtoptimierungsprozess übernehmen. **Von den neuen Komponenten profitiert allerdings nur die SQE.** Sie verfügt aufgrund der Weiterentwicklung über wesentlich mehr Möglichkeiten im Optimierungsprozess und kann auch bei der Ausführung der SQL Statements sehr viel flexibler reagieren. Aber auch die CQE ist weiterhin vorhanden. Sie wird allerdings nicht weiterentwickelt, sondern verbleibt lediglich aus Kompatibilitätsgründen auf dem System. Von den technologischen Entwicklungen profitieren daher nur SQL Statements, die von der SQE ausgeführt werden.

Tipp

Stellen Sie sicher, dass Ihre SQL Abfragen von der SQE ausgeführt werden. Nur so profitieren Sie von den neuen Technologien.

Viele Anwender haben diese einschneidenden Veränderungen zunächst nur indirekt wahrgenommen, denn mit Einspielen des Release V5R2 (häufig noch deutlicher beim Release V5R3) wurden einige SQL Statements plötzlich langsamer andere schneller! Andere Anwender haben gar keine Veränderungen wahrgenommen und wundern sich nur, dass alle Welt von verbesserten Laufzeitverhalten spricht. Wie kann das sein? Nun, die beiden Query Optimizer optimieren SQL Abfragen in unterschiedlicher Weise. Auch die Indizes werden unterschiedlich eingesetzt und so kann es unterschiedliche Laufzeitergebnisse geben. Wenn Sie bislang keine Verbesserungen wahrgenommen haben, könnte es vielleicht auch daran liegen, dass Ihre Statements bislang nicht auf der neuen SQL Engine ausgeführt wurden, obwohl Sie sich bereits im Release V5R4 oder höher befinden. Sie behaupten, dass das nicht sein kann? Viele SQL Anwender nutzen logische DDS-Dateien in Ihren SQL-Statements, um die Abfragen möglichst effizient auszuführen, weil die Dateien sortiert und/oder bereits vorselektiert sind. Ein Irrtum!

Die Verwendung der logischen DDS-Dateien in einem SQL Statement verhindert aber die Verarbeitung durch den neuen Optimizer. Warum? Dem Optimierungsprozess vorgeschaltet ist der Query Dispatcher. Er hat die Aufgabe, jedes SQL Statement zu analysieren, und entscheidet anhand vorhandener Attribute, ob die CQE oder die SQE die Verarbeitung übernehmen soll. Diese „Abfertigungsstelle“ können Sie nicht umgehen und Sie haben auch keinen direkten Einfluss auf die Entscheidung des Dispatchers. Welcher Optimizer die Verarbeitung des SQL Statements übernimmt, ist abhängig von der verwendeten Syntax im SQL Statement. Je nach Release bestimmen die verwendeten Klauseln und SQL Funktionen den Ausführungsort. Die folgende Tabelle soll Ihnen einen kurzen Überblick darüber geben:

7.6.2
Seite 6

	V5R3	V5R4	V6R1
Abfragen einer Tabelle	X	X	X
GROUP BY	X	X	X
DISTINCT	X	X	X
ORDER BY	X	X	X
JOIN	X	X	X
OR bzw. IN Prädikat	X	X	X
Symmetric Multiprozessing(SMP)	X	X	X
STAR JOIN ohne QAQQINI Unterstützung	X	X	X
VIEW	X	X	X
Common Table Expression(CTE)	X	X	X
Nested Table Expression(NTE)	X	X	X
UPDATE	X	X	X
DELETE	X	X	X
SubSelect	X	X	X
UNION	X	X	X
Rekursive CTE		X	X
LIKE		X	X
SUBSTRING		X	X
Die OLAP Funktionen RANK(), ROW NUMBER		X	X
Abfragen, die LOB Columns verwenden		X	X
Sensitive Cursor und ALWCPYDTA(*NO)		X	X
das Schlüsselwort LATERAL			X
SQL Built-In-Funktionen: UPPER, LOWER und TRANSLATE			X
Abfragen, die eine CCSID Konvertierung bewirken			X
INSERT mit VALUES oder SubSelect	–	–	–
Select/Omit und abgeleitete Indizes	–	–	–
User Defined Table Function(UDTF)	–	–	–
Tabellen mit Read Trigger	–	–	–
Abfragen über verteilte Tabellen (DB2 Multisystem)	–	–	–
Verwendung logischer DDS Dateien	–	–	–
QQQQry, API, QUERY/400, OPNQRYF	–	–	–

Das Release V6R1 wird den Übernahmeprozess auf die SQE weitestgehend abschließen. Bis auf wenige Ausnahmen können alle Abfragen auf der SQE ausgeführt werden. Allerdings werden auch weiterhin SQL Abfragen mit logischen DDS Dateien nicht durch die SQE unterstützt werden.

TIPP:

Verwenden Sie keine logischen DDS Dateien in Ihrem SQL Statement, wenn Sie von den neuen Technologien profitieren wollen!

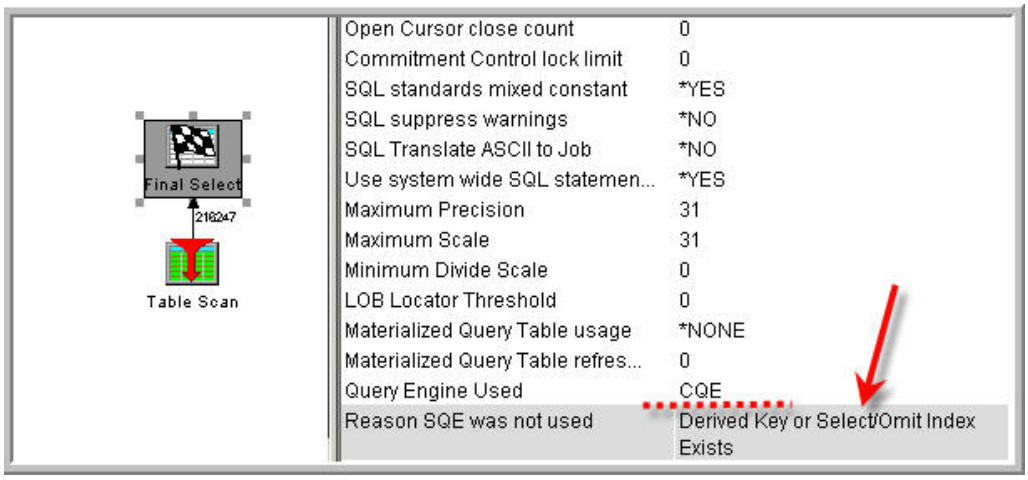
Selbst wenn Sie die logischen DDS-Dateien nicht direkt in Ihrem SQL-Statement verwenden, können sie sich negativ auf die Ausführung Ihres SQL Statements auswirken. Das SQL Statement kann unter Umständen an die CQE zurückgeleitet werden. Das heißt, die SQE beginnt mit der Optimierung, gibt das Statement dann aber an den Dispatcher zurück, der seinerseits die Abfrage zur Ausführung an die CQE übergibt. Diese Rückleitung kostet ca. 10 % – 15 % der Optimierungszeit. Natürlich entstehen diese Kosten nur ein Mal während des Optimierungsprozesses. Aber bedenken Sie, dass das Statement auch jetzt wieder von der CQE ausgeführt wird. Wie kann es zu einer derartigen Rückübertragung kommen?

Findet die SQE während der Optimierungsphase ein logisches DDS-File für die abgefragte physische Datei, dass z. B. „Select-“ und/oder „Omit“-Anweisungen oder mehrere Teildateien enthält, kann die SQE derartige logische DDS-Dateien nicht im Optimierungsprozess verwenden und leitet das Query-Statement an den Dispatcher zurück, unabhängig davon, ob diese logische DDS-Datei auch tatsächlich für die Ausführung verwendet wird. Als Begründung für die Verwendung der CQE finden Sie stets den Hinweis „Derived Key or Select/Omit Index Exists“.

Schauen wir uns hierzu ein einfaches Beispiel an:

```
SELECT HIKDNR, HIMED, HIDATE

FROM HISTOO
WHERE HIKDNR BETWEEN 1 AND 500;
```

Diese Abfrage wird auf der CQE ausgeführt, obwohl das Statement alle Kriterien für die Verarbeitung durch die SQE erfüllt:



„Abgeleitete Indizes“

Der Grund hierfür ist die logische DDS-Datei „HIST03“:

```
     R SVHIST00                  PFILE(ITPCR/HIST00)
       HIKDNR
       HITEXT
     K HIKDNR
     S HIKDNR                    RANGE(1 10000)
```

DDS Beschreibung SVHIST03

Diese logische Datei könnte für die SQL-Abfrage ein potentieller Zugriffspfad sein. Sie enthält aber Selektionskriterien, die dafür sorgen, dass die SQE das SQL Statement nicht ausführt. Sie können derartige Rückleitungen unterbinden, indem Sie die Verwendung aller „abgeleiteten Indizes“ global unterbinden. Hierzu müssen Sie im Release V5R4 das Attribut „IGNORE_DERIVED_INDEX“ in der Attributdatei QAQQINI auf *YES setzen. Ab Release V6R1 ist diese Option im Übrigen per Default entsprechend gesetzt.

TIPP:

Unterbinden Sie die Rückleitungen an die CQE, indem Sie in der Attributdatei QAQQINI die Option IGNORE_DERIVED_INDEX auf den Wert *YES setzen.

Dieses Szenario gilt natürlich nicht, wenn Sie eine SQL-View mittels der Anweisung CREATE VIEW erstellen.

7.6.3 Tabellenverknüpfungen (JOIN)

Mit SQL am Inhalt einer Tabelle zu arbeiten, ist sicher weit verbreitet. Allerdings ist es oft auch notwendig, mit dem Inhalt mehrerer Tabellen zeitgleich arbeiten zu können. Diese Möglichkeit ist mit SQL in Form von sogenannten JOINs gegeben.

Meist verhält es sich so, dass Tabelleninhalte bzw. Spalten eine Beziehung zum Inhalt einer anderen Tabelle haben.

Nehmen wir an dieser Stelle einmal unsere bekannte Tabelle ARTIKEL als Beispiel her. Darin enthalten sind einige Standardspalten, die wir in einem Artikelstamm wiederfinden können und entsprechend als Artikelstammdaten bezeichnen. Die Artikelkennzahlen sind fest an den Artikel gebunden und variieren nicht, wenn man den Artikel zum Beispiel an unterschiedlichen Lagerorten verwaltet.

Genau diesen Fall wollen wir uns nun in unserem Beispiel als Grundlage für die Vermittlung des Wissens rund um die JOIN-Möglichkeiten in SQL zu nutze machen.

Dazu werden wir uns nicht nur mit dem Inhalt der bereits vorgestellten Tabelle ARTIKEL beschäftigen, sondern diese als Grundlage nehmen, um zum Beispiel den Lagerbestand des jeweiligen Artikels bestimmen zu können. Diesen Lagerbestand finden wir in einer weiteren Tabelle „LAGERBEST“. Der Aufbau dieser Tabelle schaut folgendermaßen aus:

Details zum fernen System | Tasks | iSeries-Tabellensicht | iSeries-Befehlsprotokoll | Datenausgabe | iSeries-Feldtabellensicht

Datei SQLDEMO/LAGERBEST (7 Felder)

Name	Satz	Art	Länge	Text	Aliasname
FIRMA	LAGERBEST	Zeichen	3	Firma	
ARTIKELNR	LAGERBEST	Gepackt dezimal	5,0	ArtikelNr	
LAGERORT	LAGERBEST	Zeichen	3	LagerOrt	
BESTAND	LAGERBEST	Gepackt dezimal	5,0	Bestand	
MEH	LAGERBEST	Zeichen	10	Mengeneinheit	
SPERRKZ	LAGERBEST	Zeichen	1	Sperrkennzeichen	
INVDATUM	LAGERBEST	Datum	10	InventurDatum	

Aufbau der Tabelle LAGERBEST

In der Lagerbestandsdatei befinden sich die Bestandsinformationen pro Firma, Teil und Lagerort.

Einen Auszug der Beispieldaten zeigt exemplarisch deren Aufbau und den Inhalt der Tabelle LAGERBEST:

Details zum fernen ... | Tasks | iSeries-Tabellensicht | iSeries-Befehlspro... | Datenausgabe | iSeries-Feldtabelle... | iSeries-Datentabel...

assql:SQLDEMO/LAGERBEST(LAGERBEST)

*RCDNBR	FIR...	ARTIKELNR	LA...	BESTAND	MEH	SPERRKZ	INVDATUM
1	001	10.001	001	10	ST	*NULL	*NULL
2	001	10.002	001	15	ST	*NULL	*NULL
3	001	10.002	200	20	STK	*NULL	*NULL
4	001	10.003	001	10	STK	*NULL	*NULL
5	001	10.004	001	100	STK	*NULL	*NULL
6	001	10.005	001	50	STK	*NULL	*NULL
7	001	10.005	200	11	STK	*NULL	*NULL
8	001	10.005	300	7	STK	*NULL	*NULL
9	001	10.006	001	17	STK	*NULL	*NULL
10	001	10.006	200	1	STK	*NULL	*NULL
11	001	20.001	001	113	STK	*NULL	*NULL
12	001	20.001	200	11	STK	*NULL	*NULL

Beispieldaten

Basierend auf diesen Informationen und den Artikelstammdaten wollen wir uns nun die Verkettung von Informationen aus mehreren Tabellen zur gemeinsamen Auswertung mit SQL ansehen.

Die Beziehung zwischen der Tabelle ARTIKEL und der Tabelle LAGERBEST wird über die folgenden Spalten hergestellt:

- Firma
- Firma
- ArtikelNr
- ArtikelNr

Wir können die beiden Tabellen also über die Spalten Firma und ArtikelNr miteinander in Beziehung bringen. In unserem Beispiel haben die Spalten zufällig in beiden Tabellen dieselben Bezeichnungen oder besser Spaltennamen. Das muss natürlich nicht so sein und ist keine Voraussetzung für das Erstellen der Verbindung zwischen den Tabellen.

Die JOIN-Angabe ermöglicht das Erstellen von Beziehungen zwischen Informationen aus verschiedenen Tabellen. Mit dieser Beziehungserstellung stehen uns im weiteren Verlauf alle Spalten der angesprochenen Tabellen zur Verarbeitung mit SQL zur Verfügung.

Nehmen wir einmal an, wir wollen alle Datensätze der Tabelle ARTIKEL auslesen und deren Bestände aus der Lagerbestandsdatei LAGERBEST ausgeben. Ein Beispiel für eine solche SELECT-Anweisung zeigt das folgende Codebeispiel.

```
SELECT a.firma, a.Artikelnr, L.Bestand
  FROM SQLDEMO.ARTIKEL a inner JOIN SQLDEMO.LAGERBEST L on
  (a.Firma = L.Firma)
```

Die SELECT-Anweisung haben wir bereits zuvor behandelt. In ihr sind einzelne Spalten zur Anzeige definiert (die Spalten Firma und ArtikelNr aus der Tabelle ARTIKEL sowie die Spalte „Bestand" aus der Tabelle LAGERBEST). Die Besonderheit bei der Angabe der Spaltennamen liegt in der Qualifizierung der Spaltennamen. Sie ist deshalb sinnvoll bzw. erforderlich, weil wir mit mehreren Tabellen arbeiten und es eben nicht nur in unserem Beispiel vorkommen kann, dass Spalten in unterschiedlichen Tabellen denselben Namen haben.

In dem Beispiel werden zwei Tabellen ausgewertet. Diese sind nach der Anweisung FROM definiert:

- Tabelle ARTIKEL mit der Qualifikation „a"
- Tabelle LAGERBESTAND mit der Qualifikation „L"

Diese Form der Tabellenangabe weist den SQL-Interpreter an, die Spalten der Tabellen nicht nur mit dem eigentlichen Spaltennamen zu verarbeiten, sondern ihn mit der Ergänzung in Form der Qualifikation zu verwenden. Für die Spalten der Tabelle ARTIKEL hat dies zur Folge, dass wir die Spaltennamen mit einem führenden „a." angeben; Spaltennamen der Tabelle LAGERBEST werden mit einem führenden „L." verarbeitet. Auf diese Weise kann vermieden werden, dass es bei gleichnamigen Spalten aus verschiedenen Tabellen zu Verwechslungen kommt.

Anmerkung

Die Verwendung von Qualifikationskürzeln (auch Alias) ist optional, erleichtert bei der Arbeit mit verschiedenen Tabellen in SQL jedoch die Zuordnung der verwendeten Spalten zu den jeweiligen Tabellen. Die Verwendung des Abkürzungszeichens (in unserem Beispiel A und L) ist frei wählbar. Es kann, wie in unserem Beispiel, einstellig sein oder auch durch mehrere Zeichen definiert werden. Ebenso ist die Schreibweise der Abkürzung – gemeint ist hier die Groß-/Kleinschreibung – frei wählbar, da die SQL-Anweisungen auf IBM i in Großbuchstaben umgewandelt und interpretiert werden.

Die Ergebnisspalten und die FROM-Anweisung haben wir bereits im Kapitel zur SELECT-Behandlung besprochen. Neu ist hier die ON-Erweiterung. Sie stellt die Beziehungsdefinition zwischen den beiden Tabellen dar.

IBM bietet mit SQL auf IBM i die bekannten Verbindungsfunktionen für die Erstellung von Beziehungen zwischen Informationen aus verschiedenen Tabellen.

Solche Beziehungen basieren auf Schlüsselwerten, die in beiden Tabellen enthalten sein müssen. Die Beziehungserstellung basiert dabei auf der Verbindungsoption JOIN. Ein JOIN basiert auf mindestens einer Verbindung von zwei

Ergänzung 15/2010

oder mehr Tabellen, die über bestimmte Spalten miteinander in Beziehung gebracht werden. Auf unser Beispiel bezogen sind dies die Werte in den Spalten „Firma“ und „Artikelnr“.

Anmerkung

JOIN-Angaben können in SQL für alle Spalten einer Tabelle genutzt werden. Dabei ist es unerheblich, ob die Spalte selbst zum Schlüssel der Tabelle gehört. Somit kann eine beliebige Anzahl von Beziehungen bzw. Verbindungen zwischen den Zeilen einer Tabelle zu einer anderen Tabelle hergestellt werden. Zwar ist IBM i mit den SQL-Engines in der Lage, die Zugriffssteuerung derart zu optimieren, dass in den meisten Fällen die schnellste Ausführung der JOIN-Abfragen erstellt wird – aber letztlich steuert der Entwickler bzw. SQL-Ersteller die Performance der Auswertung durch geschickte JOIN-Definitionen mit. Diese sollte so ausgelegt sein, dass nach Möglichkeit bestehende Indizes genutzt werden können und das Subset an Ergebnissen genau die Informationen liefert, die gewünscht sind. Vermeiden Sie zu komplexe JOIN-Bedingungen und gliedern Sie diese im Bedarfsfall so auf, dass sie auch für andere Entwickler verständlich sind!

Außerdem ist wichtig zu wissen, dass die JOIN-Spalten nicht Bestandteil des gewünschten Ergebnisses sein müssen. Die JOIN-Spalten dienen nur der Erstellung der Beziehung und der damit verbundenen Abfragen, wirken sich aber nicht auf das Ergebnis und dessen Spalten aus.

Die folgende Abbildung zeigt exemplarisch die Beziehung von Spalten unterschiedlicher Tabellen zueinander.

Die sollte eigentlich so ausschauen:

Spaltenname Tabelle ARTIKEL	Art	In Auswertung		Spaltenname Tabelle LAGERBEST	Art	In Auswertung
Firma	3 A	Ja		Firma	3 A	
ArtikelNr	5,0	Ja	→	ArtikelNr	5,0	
Bezeichnung	50 A	Ja		Lagerort	3, A	
Sortiment	10 A			Bestand	5,0	Ja
AuslaufKz	1 A			MEH	3,A	
SperrKz	1 A			Sperrkz	1,A	
Beschaffung	1 A			InvDatum	Date	
Anlagedatum						

Nehmen wir einmal an, wir wollen aus der Lagerbestandsdatei die Artikel auflisten. Neben den Beständen soll auch die Artikelbezeichnung in der Auswertung enthalten sein. Wie Sie sehen können, ist diese nicht in der Lagerbestandsdatei enthalten, sondern im Artikelstamm zu finden.

Folglich müssen wir eine Beziehung zwischen Artikelstamm und Lagerbestand herstellen, damit das System in der Lage ist, die gewünschte Kombination aus den Informationen den beiden Tabellen liefern zu können.

Als Schlüssel bzw. Verbindung zwischen den beiden Tabellen kommen die folgenden Spalten in Frage:

- Die Firmennummer
  (Spalte FIRMA in der Tabelle ARTIKEL und Spalte FIRMA in der Tabelle LAGERBEST)
- Die Artikelnummer
  (Spalte ArtikelNr in der Tabelle ARTIKEL und Spalte ArtikelNr in der Tabelle LAGERBEST)

Diese beiden Spalten bilden eine Beziehung und können somit zusammen für den Zugriff auf die Artikelstammtabelle genutzt werden, um die Artikelbezeichnung für eine Lagerbestandsliste zu ermitteln.

Ein zugehöriges SQL-Beispiel könnte wie folgt aussehen:

```
SELECT a.firma, a.Artikelnr, a.bezeichn, L.Bestand
  FROM SQLDEMO.ARTIKEL a inner JOIN SQLDEMO.LAGERBEST L on
(a.Firma = L.Firma)
```

Nach der Ausführung dieser SQL-Anweisung erhalten wir folgendes Ergebnis:

SQL-Editor

Nachrichten | Parameter | Ergebnisse | Profilerstellungsdaten

FIRMA	ARTIKELNR	BEZEICHN		BESTAND
001	10001	Schnellhefter A4	...	10
001	10002	Büroklammern 1000 Stück	...	15
001	10002	Büroklammern 1000 Stück	...	20
001	10003	Bleistift		10
001	10004	Ringbinder	...	100
001	10005	Radiergummi	...	50
001	10005	Radiergummi	...	11
001	10005	Radiergummi	...	7
001	10006	Locher		17
001	10006	Locher		1
001	20001	Schere		113
001	20001	Schere		11
001	10030	Fachbuch SQL	...	3
001	10030	Fachbuch SQL	...	1
001	10030	Fachbuch SQL	...	5

Ergebnis des JOIN

7.6.3
Seite 6

Wir sehen an diesem Beispiel die Lagerbestände der unterschiedlichen Artikel. Dabei haben wir die Ergebnisspalten aus den beiden Tabellen zur Anzeige erhalten:

- Die Spalte Firma kommt aus der Tabelle ARTIKEL.
- Die Spalte ArtikelNr kommt aus der Tabelle ARTIKEL.
- Die Spalte Bezeichn kommt aus der Tabelle ARTIKEL.
- Die Spalte Bestand kommt aus der Tabelle LAGERBEST.

Auf diese Weise haben wir also die Inhalte zweier Tabellen zusammen in einer SQL-Anweisung verarbeitet und auch Spaltenwerte aus verschiedenen Tabellen als Werte ermittelt.

Aber ist Ihnen etwas aufgefallen? Schauen wir uns die beiden Tabelleninhalte noch einmal einzeln an:

Nachfolgend der Inhalt der Tabelle ARTIKEL mit den Artikelstamminformationen:

Details zum fernen System | Tasks | iSeries-Tabellensicht | iSeries-Befehlsprotokoll | Datenausgabe | iSeries-Feldtabellensicht | iSeries-Datentabellensicht

assql:SQLDEMO/ARTIKEL(ARTIKEL)

*RCDNBR	FIRMA	ARTIKELNR	BEZEICHN	SORTIMENT	A	S	B	ANLAGEDT
1	001	10.001	Schnellhefter A4	Standard	N	N		2009-08-08
2	001	10.002	Büroklammern 1000 Stück	Standard	N	N		2009-08-08
3	001	10.003	Bleistift	Standard	N	N		2009-08-08
4	001	10.004	Ringbinder	Standard	N	N		2009-08-08
5	001	10.005	Radiergummi	Standard	N	N		2009-07-01
6	001	10.006	Locher	Standard	N	N		2008-10-10
7	001	10.008	Drucker Toner	Drucker	N	N		2009-08-08
8	001	10.009	Mauspad	Computer	N	N		2009-08-01
10	001	10.011	Post It Mittel	Standard	N	N		2009-08-01
11	001	10.012	Post It groß	Standard	N	N		2009-08-01
12	100	10.010	DIN A4 Druckerpapier weiß	Papier	N	N		2009-08-01
13	001	10.015	DIN A4 Druckerpapier weiß	Standard	N	N		2009-08-01
14	001	20.001	Schere	Standard	N	N		2009-08-01
15	010	10.011	Ladegerät Universal	Elektro	N	N		2009-08-01
16	010	10.020	Heftklammern 1000 Stück	Standard	N	N		2009-08-01
17	010	10.030	USB Anschlusskabel	IT	N	N		2009-08-01
18	001	10.030	Fachbuch SQL	Buecher	N	N		2009-01-01
21	001	20.015	Atlas 2010	Standard	N	N		2009-01-01
22	1	10.100	CD Hüllen	Standard	N	N		*NULL
23	1	10.101	DVD Huellen	Standard	N	N		*NULL
24	1	10.102	Reinigungsband	IT	N	N		*NULL
25	1	10.103	LTO Sicherungsband	Standard	N	N		*NULL
26	1	10.104	LTO Reinigungsband	Standard	N	N		*NULL
27	1	10.105	ISDN Mobiltelefon	Standard	N	N		*NULL
28	1	10.106	ISDN Basiseinheit	Standard	N	N		*NULL
29	1	10.107	Papierkorb	Standard	N	N		*NULL
30	1	10.108	Schere klein	Standard	N	N		*NULL
31	1	10.109	Schere groß	Standard	N	N		*NULL
32	1	10.110	Schreibtischleuchte	Standard	N	N		*NULL
33	2	20.001	Toner Drucker A	Zubehoer	N	N		*NULL
34	2	20.007	Toner Drucker B	Zubehoer	N	N		*NULL
35	2	20.003	Toner Drucker A	Zubehoer	N	N		*NULL
36	200	*NULL	*NULL	*NULL	N	N		*NULL
37	2	20.011	Tinte Drucker A	Zubehoer	N	N		*NULL

Auszug aus dem Inhalt der Tabelle ARTIKEL

Und hier die Zeilen oder Datensätze aus der Lagerbestandsdatei LAGERBEST:

Details zum fernen System | Tasks | iSeries-Tabellensicht | iSeries-Befehlsprotokoll | Datenausgabe | iSeries-Feldtabellensicht

assql:SQLDEMO/LAGERBEST(LAGERBEST)

*RC...	FI...	ARTIKELNR	LAGERORT	BESTAND	MEH	SPERRKZ	INVDATUM
1	001	10.001	001	10	ST	*NULL	*NULL
2	001	10.002	001	15	ST	*NULL	*NULL
3	001	10.002	200	20	STK	*NULL	*NULL
4	001	10.003	001	10	STK	*NULL	*NULL
5	001	10.004	001	100	STK	*NULL	*NULL
6	001	10.005	001	50	STK	*NULL	*NULL
7	001	10.005	200	11	STK	*NULL	*NULL
8	001	10.005	300	7	STK	*NULL	*NULL
9	001	10.006	001	17	STK	*NULL	*NULL
10	001	10.006	200	1	STK	*NULL	*NULL
11	001	20.001	001	113	STK	*NULL	*NULL
12	001	20.001	200	11	STK	*NULL	*NULL
13	001	10.030	001	3	STK	*NULL	*NULL
14	001	10.030	200	1	STK	*NULL	*NULL
15	001	10.030	111	5	STK	*NULL	*NULL

Inhalt der Tabelle LAGERBEST

Betrachten wir die beiden Tabellen und das Ergebnis unserer ersten JOIN-Anweisung, dann stellen wir fest, dass offensichtlich nicht alle Datensätze verarbeitet wurden. Die Artikelstammtabelle beinhaltet weitaus mehr Artikelstämme, als in unserem JOIN-Ergebnis ausgewiesen sind – und das obwohl wir in der SELECT-Anweisung keine Einschränkungen in Form einer WHERE-Bedingung angegeben haben. Wie kann das sein?

Eine Betrachtung der Tabelle LAGERBEST bringt uns da schon ein wenig weiter. Schauen Sie sich die vorhergehende Abbildung nochmals an. Dort finden wir in der ersten Spalte die Satznummer. Sie ist in der Anzeige enthalten, da die Tabelle über keinen Primärschlüssel verfügt. Die Satznummer zeigt die Anzahl der in der Tabelle befindlichen Datensätze bzw. Zeilen an. Vergleichen wir diese mit unserem JOIN-Ergebnis, so stellen wir fest, dass die Anzahl der ermittelten Ergebniszeilen exakt mit den 15 Zeilen aus der Lagerbestandstabelle übereinstimmen. Folglich scheint diese „führend“ bei der Ermittlung der Ergebnisse der JOIN-Anweisung zu sein.

Die Antwort ist in der genutzten JOIN-Variante zu suchen. JOIN-Anweisungen sind in SQL in den verschiedensten Varianten möglich. Dabei müssen jeweils die verschiedenen möglichen Konstellationen der Abhängigkeiten und Vorkommen der Daten in den verknüpften Tabellen genau bedacht werden.

Stellen wir uns dazu ein typisches Szenario vor:

Eine Tabelle A beinhaltet Datensätze, die mit dem Inhalt einer Tabelle B mit einem JOIN in Beziehung gebracht werden sollen.

Tabelle A kann, muss aber nicht für jeden Datensatz in Tabelle B einen passenden Eintrag haben. Genauso kann es Konstellationen geben, bei denen in Tabelle A ein Eintrag existiert – ein Gegenstück in Tabelle B jedoch fehlt.

7.6.3
Seite 8

Damit sind wir mit den möglichen Konstellationen noch nicht am Ende:

Mehrfachvorkommen in einer der beiden Tabellen oder gar in beiden Tabellen runden die möglichen Varianten von Datenbeziehungen weiter ab.

Um ein gewünschtes Ergebnis erreichen zu können, muss der Anwender bzw. SQL-Entwickler abwägen, welche der zur Verfügung stehenden JOIN-Derivate für den einzelnen Fall zum Einsatz kommen sollen.

Anmerkung

Die Verwendung von JOIN-Anweisungen bedarf einer besonderen Kenntnis der möglichen Datenkonstellationen und Abhängigkeiten in den zu verarbeitenden Tabellen. Wenn Sie JOIN einsetzen, dann sollten Sie bei der Erstellung der JOIN-Abfrage sehr genau die möglichen Konstellationen bedenken und im Zweifelsfall verschiedene JOIN-Abfragen testen und deren Ergebnisse miteinander vergleichen. Nicht selten liefern SQL-Abfragen „falsche“ Ergebnisse, weil JOIN-Anweisungen in der falschen Variante zum Ermitteln der Informationen eingesetzt wurden.

Ohne an dieser Stelle auf die verschiedenen JOIN-Möglichkeiten bereits eingehen zu wollen, betrachten wir hier die Syntax der genutzten JOIN-Variante:

```
SELECT Tabelle1.Spaltenname1, Tabelle2.Spaltenname2 FROM
Tabellenname1
 {JOIN Tabellenname2 ON Tabellenname1.Spalte1  =
 Tabellenname2.Spaltenname3}
```

Dies ist eine von verschiedenen JOIN-Anweisungen. Diese Variante haben wir im zuvor gezeigten Beispiel verwendet.

Schauen wir uns nun die verschiedenen Möglichkeiten der JOIN-Nutzung einmal genauer an.

7.6.3.1 INNER JOIN

Eine Version der Verwendung von JOIN-Anweisungen haben wir bereits kennengelernt.

```
SELECT a.firma, a.Artikelnr, a.bezeichn, L.Bestand
  FROM SQLDEMO.ARTIKEL a JOIN SQLDEMO.LAGERBEST L on
  (a.Firma = L.Firma)
```

Bei diesem Beispiel handelt es sich um die klassische Form der Verwendung der JOIN-Anweisung. Sie wird auch als „INNER JOIN“ oder „Condition JOIN“ bezeichnet. Die Bezeichnung rührt von der Verwendung der Bedingung „ON“ her, die mit der JOIN-Definition angegeben wird.

Bei einem INNER JOIN wird die Kombination von Zeilen aus zwei Tabellen genutzt, die in einer definierten Beziehung zueinander stehen. Bei diesem JOIN bzw. diesem Abgleich werden also die Inhalte zweier Spalten miteinander verglichen und bei Übereinstimmung wird die entsprechende Konstellation als „Ergebnis“ für die weitere Verarbeitung zur Verfügung gestellt.

Diese Form der Verbindung bedingt unweigerlich das Vorhandensein gleichartiger bzw. miteinander in Verbindung zu bringender Datensätze oder Zeilen in verschiedenen Tabellen. Bezogen auf unser Beispiel bedeutet dies, dass das Ergebnis deshalb zu erklären ist, weil in der Lagerbestandsdatei nur für 15 Artikel Bestandsdaten zu verschiedenen Lagerorten verfügbar sind. Durch die Notwendigkeit, dass bei der verwendeten JOIN-Anweisung die Ergebnisse in beiden Tabellen existieren müssen, hat die Lagerbestandsdatei die Anzahl der gelieferten Datensätze maßgeblich beeinflusst.

Ein INNER JOIN ist auf IBM i – wie auch auf anderen Plattformen – als SQL-Standard nach den Vorgaben von SQL92 ausgelegt.

Die Syntax dieser ersten JOIN-Variante sieht dabei wie folgt aus:

```
SELECT Tabelle1.Spaltenname1, Tabelle2.Spaltenname2 FROM
Tabellenname1
 {JOIN Tabellenname2 ON Tabellenname1.Spalte1  =
 Tabellenname2.Spaltenname3}
```

Ergänzung 15/2010

Anmerkung

Bei der klassischen Verwendung von JOIN haben wir zwei verschiedene Schreibweisen. Sehen Sie sich deshalb die folgende Schreibweise an, die exakt dasselbe Ergebnis liefert wie die vorherige JOIN-Syntax:

7.6.3
Seite 10

```
SELECT Tabelle1.Spaltenname1, Tabelle2.Spaltenname2 FROM
Tabellenname1
 {INNER JOIN Tabellenname2 ON Tabellenname1.Spalte1  =
 Tabellenname2.
Spaltenname3}
```

Beachten Sie in diesem Beispiel die Verwendung von „INNER JOIN" – also die Angabe von „INNER". Dieses Beispiel zeigt die vollständige Schreibweise der klassischen INNER-JOIN-Anweisung. Fehlt diese Angabe bei einer JOIN-Konstruktion, dann interpretiert die SQL-Ausführungsumgebung den JOIN automatisch als INNER JOIN.

Die Verwendung von „INNER" ist optional, also nicht erforderlich, erleichtert jedoch die Lesbarkeit und beugt eventuellen Verwechslungen vor. Ich empfehle deshalb, „INNER" stets mit anzugeben, um Verwechslungen mit den übrigen JOIN-Varianten zu vermeiden!

Bei einem INNER JOIN bzw. einem Condition JOIN wird in der Bedingung die Art der Verbindung der Datensätze verschiedener Tabellen definiert.

Doch wie hat die JOIN-Verarbeitung eigentlich intern stattgefunden?

Mit der Angabe, wie die Tabellen zueinander in Beziehung gebracht werden sollen, haben wir SQL angewiesen, näher bestimmte Vergleiche durchzuführen. Diese Vergleiche muss das System je nach Komplexität der JOIN-Anweisung und der vorhandenen Zugriffswege bzw. SQL-Komponenten unter Umständen mehrfach pro Zeile durchführen. Das Ergebnis wird in einem virtuellen Bereich angelegt und später zur Ausgabe gebracht.

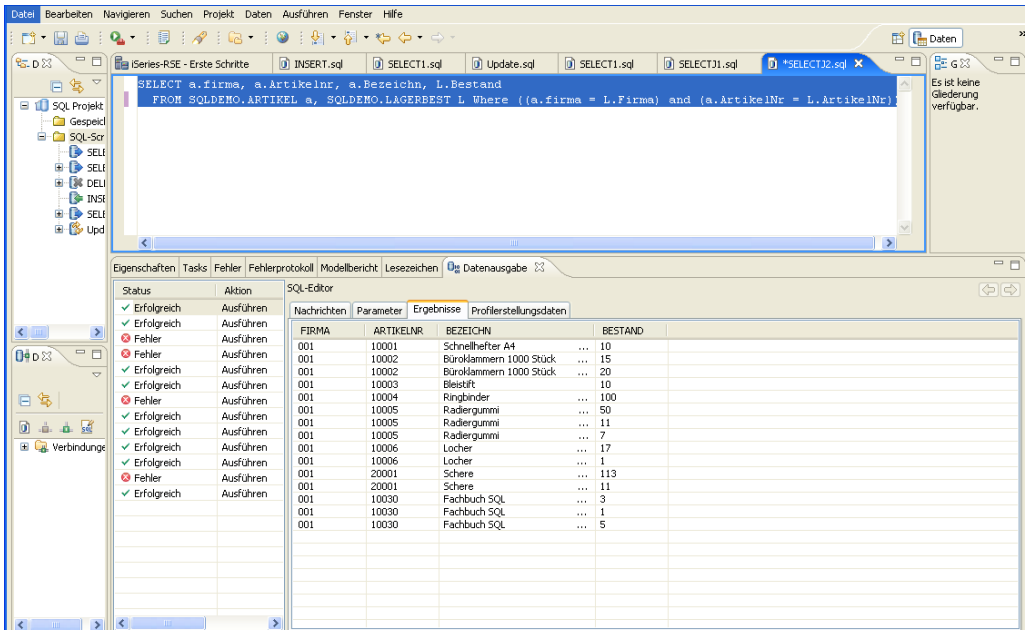
SQL hat demzufolge den Inhalt der Tabellen ARTIKEL und LAGERBEST zueinander in Beziehung gesetzt und deren Inhalte mit den Spalten FIRMA und ARTIKEL verglichen. Für jeden Eintrag, der in den beiden Tabellen gefunden wurde, wurde ein Ergebnissatz bereitgestellt, der – nachdem er virtuell zwischengespeichert wurde –, abschließend als Ergebnis mit den gewünschten Spalten ausgegeben wurde.

INNER-JOIN-Anweisungen existieren auf IBM i bereits seit geraumer Zeit. Ebenso sind diese INNER-JOIN-Anweisungen auch auf anderen Plattformen seit Jahren verfügbar. Wer sich mit dieser Form der Verknüpfung von Tabelleninhalten bereits vor einigen Jahren beschäftigt hat, der wird sich vielleicht ein wenig über die zuvor gezeigten Beispiele wundern. Das liegt an der Verwendung der verschiedenen SQL-Standards. SQL92 – einer der am weitesten verbreiteten SQL-Standards wurde bereits 1992 angekündigt. Diesem folgten 1999 SQL3; 2003 SQL:2003 und 2008 SQL:2008.

Ältere SQL-Standards erlauben auch die folgende Verwendung eines INNER JOINS:

```
SELECT a.firma, a.Artikelnr, a.Bezeichn, L.Bestand
  FROM SQLDEMO.ARTIKEL a, SQLDEMO.LAGERBEST L Where ((a.firma
= L.Firma) and (a.ArtikelNr = L.ArtikelNr))
```

Führen wir dieses SQL-Beispiel einmal aus. Das Ergebnis zeigt die folgende Abbildung:



Variante des INNER JOIN

Das Ergebnis dieser abgewandelten JOIN-Anweisung ist dasselbe wie im ersten gezeigten Codebeispiel. Auch dieser Code zeigt eine INNER-JOIN-Variante – allerdings in der veralteten Form.

Die Möglichkeit der Eingabe eines INNER JOIN war bis zum Erscheinen von SQL92 Standard. Der Tradition verpflichtet, hat IBM auch in diesem Bereich ältere Syntax und Standards auf dem System i verfügbar gehalten. Damit können ältere SQL-Anweisungen auch heute noch auf modernsten Modellen und Betriebssystemen von IBM i ohne Anpassungen ausgeführt werden.

Und hierin liegt auch der Grund, warum ich Ihnen diese eigentlich veraltete INNER-JOIN-Angabe vorstelle. Nicht selten finden wir auch heute noch Programme oder SQL-Definitionen vor, die vor Jahrzehnten erstellt und seitdem nicht mehr verändert wurden. Sie verrichten nach wie vor ihren Dienst. Sind allerdings Anpassungen notwendig, dann fragt sich der eine oder andere Entwickler oder SQL-Anwender, was sich hinter diesem eigenartigen Code verbirgt. Aus diesem Grund stelle ich hier auch einige „veraltete“ SQL-Derivate vor.

7.6.3
Seite 12

Anmerkung

Auch wenn die älteren SQL-Standards auf IBM i noch unterstützt werden, so empfiehlt es sich doch, die neuesten Versionen der SQL-Codierung zu verwenden. Damit erleichtert man sich nicht nur selbst die Arbeit, sondern auch Kollegen den Einstieg in die Arbeit mit SQL auf IBM i. Zudem ist man mit der Verwendung neuerer und modernerer SQL-Varianten auch für den SQL-Einsatz auf Nicht-IBM-i-Systemen besser vorbereitet.

Der Einsatz moderner SQL-Standards dient nicht nur der Lesbarkeit und der Vergleichbarkeit von SQL-Anweisungen auf verschiedenen Plattformen, sondern ist in gewissen Fällen auch die Voraussetzung dafür, dass SQL-Optimierung in der SQE durchgeführt werden kann. Auch aus diesem Grund sollte der Einsatz der SQL-Standards ab SQL92 bevorzugt genutzt werden.

Doch wo liegt nun der wesentliche Unterschied zwischen der „modernen" Variante des INNER JOIN und der klassischen Version?

Die Antwort zeigt sich uns bei der Betrachtung der WHERE-Bedingung, die in der klassischen Form vorkommt.

Beachten Sie im folgenden Beispiel, dass wir hier eine WHERE-Bedingung nutzen.

```
SELECT a.firma, a.Artikelnr, a.Bezeichn, L.Bestand
  ROM SQLDEMO.ARTIKEL a, SQLDEMO.LAGERBEST L Where ((a.firma
= L.Firma) and (a.ArtikelNr = L.ArtikelNr))
```

In dieser Form des INNER JOIN wird die Bedingung des Vergleichs über die WHERE-Klausel abgebildet – eine Verwendung von „JOIN" kommt in dem SQL-Beispiel nicht vor.

Andere Plattformen als IBM i kennen im SQL-Standard zudem einige Abwandlungen des INNER JOIN. Sollten Sie sich mit SQL auf anderen Betriebssystemen beschäftigt haben, so haben Sie vielleicht mit NATURAL JOIN gearbeitet. Solche Derivate des INNER JOIN stehen auf IBM i nicht zur Verfügung.

7.6.3.1.1 INNER JOIN mit mehr als zwei Tabellen

In den Beispielen zuvor haben wir zwei Tabellen für den Einsatz mit JOIN verarbeitet. Es kann in der Praxis natürlich auch Konstellationen geben, bei denen mehrere Tabellen mit einer SQL-Anweisung verarbeitet werden müssen. Das ist natürlich auch bei der Verwendung von INNER JOINS möglich.

Schauen wir uns dazu ein kurzes Beispiel an.

Wir greifen auf die bisher bekannten Tabellen ARTIKEL und LAGERBEST zurück und ermitteln mit einer dritten Tabelle LAGWERT den Lagerwert.

Die Tabelle LAGWERT hat folgenden Aufbau:

Details zum fernen ... | Tasks | iSeries-Tabellensicht | iSeries-Befehlspro... | Datenausgabe | iSeries-Feldtabelle... | iSerie

Datei SQLDEMO/LAGWERT (5 Felder)

Name	Satz	Art	Länge	Text
FINR	LAGWERT	Zeichen	3	Firma
ARTNR	LAGWERT	Gepackt dezimal	5,0	ArtNr
EPREIS	LAGWERT	Gepackt dezimal	9,2	Einzelpreis
CURC	LAGWERT	Zeichen	3	Waehrung
PRDAT	LAGWERT	Datum	10	WertDatum

Spalten der Tabelle LAGWERT

Der Beispielinhalt dieser Tabelle sieht wie folgt aus:

etails zum fernen ... | Tasks | iSeries-Tabellensicht | iSeries-Befehlspro... | Datenausgabe | iSeries-Feldtabelle... | iSeries-Datentabel...

sql:SQLDEMO/LAGWERT(LAGWERT)

*RCDNBR	FINR	ARTNR	EPREIS	CURC	PRDAT
1	001	10.001	1,17	EUR	*NULL
2	001	10.002	5,20	EUR	*NULL
3	001	10.003	17,30	EUR	*NULL
4	001	10.006	7,10	EUR	*NULL

Beispielinhalt der Tabelle LAGWERT

Wie Sie sehen können, befindet sich in der Tabelle pro Firma und Artikel ein Eintrag mit dem Einzelpreis des Artikels. Nehmen wir einmal an, wir wollen eine Auswertung erstellen, die folgende Informationen beinhaltet:

- Firma (aus Artikelstamm)
- ArtikelNr (aus Artikelstamm)
- Bezeichnung (aus Artikelstamm)
- Lagerbestand (aus Lagerbestandstabelle)
- Lagerwert (Einzelpreis aus Lagerwerttabelle)

7.6.3
Seite 14

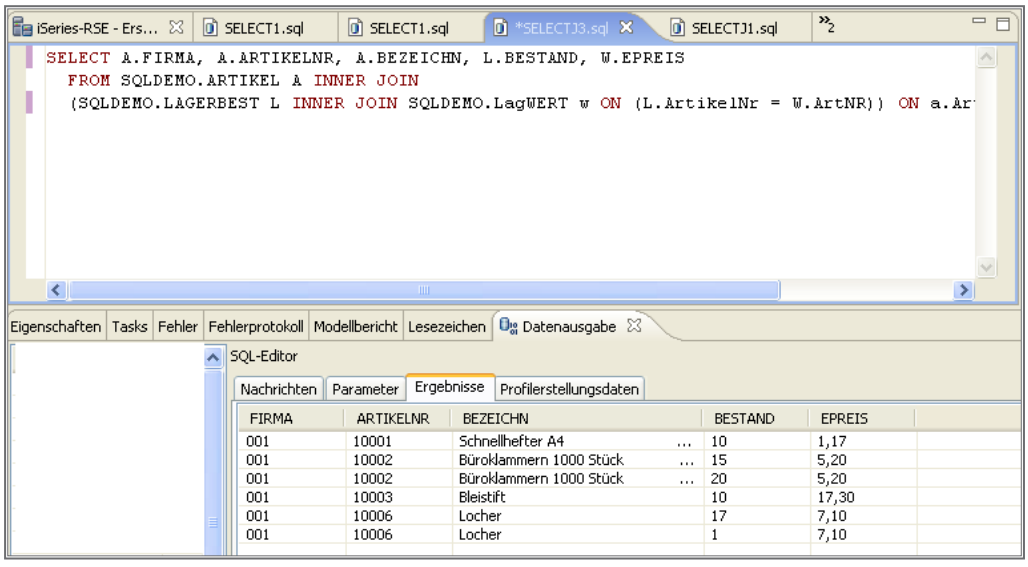
In diesem Fall müssten wir die drei Tabellen miteinander verbinden. Als Schlüsselspalten können dabei die Firmennummer und der Artikel genutzt werden. Für das Abrufen der Werte aus der Lagerwerttabelle benötigen wir noch den Lagerort, der in der Lagerbestandstabelle enthalten ist.

Beachten Sie, dass die Spalte „Firmennummer“ in allen drei Tabellen vorhanden ist. In der Tabelle „LAGWERT“ trägt die Spalte aber einen anderen Namen, nämlich „FINR“. Dieses Beispiel wurde bewusst so gewählt, um Ihnen zu zeigen, dass Spalten mit unterschiedlichen Namen in verschiedenen Tabellen miteinander für einen Vergleich bzw. zwecks Verbindung genutzt werden können.

Die SQL-Anweisung kann dazu wie folgt aufgebaut sein:

```
SELECT A.FIRMA, A.ARTIKELNR, A.BEZEICHN, L.BESTAND, W.EPREIS
  FROM SQLDEMO.ARTIKEL A INNER JOIN
  (SQLDEMO.LAGERBEST L INNER JOIN SQLDEMO.LagWERT w ON (L.
ArtikelNr = W.ArtNR))
ON a.ArtikelNr = L.ArtikelNr
```

Das Ergebnis der Ausführung dieser SQL-Anweisung sieht so aus:



Ergebnis der SQL-Ausführung

Die Verwendung von mehr als zwei Tabellen in einer JOIN-Angabe bedarf der Einhaltung strikter Zuordnungsregeln. Klammerkonstrukte helfen dabei, die Zuordnungen und einzelnen Vergleiche übersichtlicher zu gestalten. Die Reihenfolge der Angaben ist beim Einsatz eines INNER JOIN letztlich unbedeutend. Sie kann beliebig erfolgen. Bestimmt wird das Ergebnis von den Übereinstimmungen in den verwendeten Tabellen.

Tipp

JOIN-Verwendung ist nicht auf die Größe einer Tabelle beschränkt. Dennoch gilt beim Zugriff auf Tabellen mit einer großen Anzahl Zeilen bzw. Datensätze, dass diese bei komplexen Abfragen wesentlich die Ausführungsgeschwindigkeit beeinflussen können.

In Bezug auf Leistungsoptimierung gilt bei der Nutzung von JOIN die grobe Regel, dass zunächst die Tabellen verarbeitet werden sollen, die über weniger Inhalte – also Zeilen – verfügen und danach die größeren Tabellen mit JOIN behandelt werden. Verbunden ist diese Richtlinie mit der Tatsache, dass systemintern die ermittelten Inhalte mit virtuellen Tabellen verarbeitet werden. Diese Verarbeitung wird erleichtert, wenn zunächst geringere Datenmengen verarbeitet werden müssen. Eine Leistungsoptimierung lässt sich gegebenenfalls auch mit Hilfe von WHERE-Klauseln erreichen, mit denen die zu verarbeitenden Tabelleninhalte auf ein Minimum bzw. Optimum beschränkt werden können.

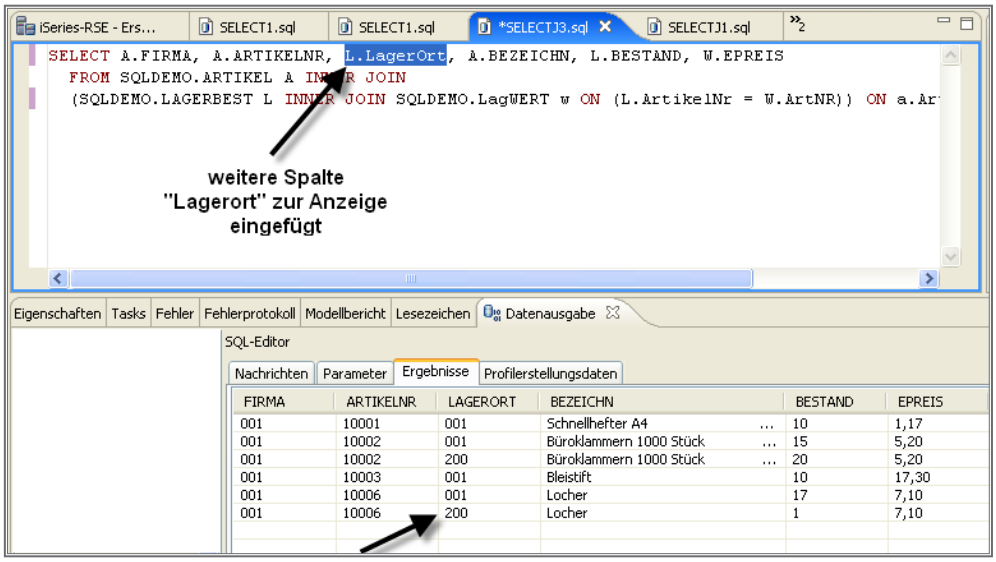
In Verbindung mit der SQL-Optimierung lassen sich so bei SQL JOINs zum Teil deutlich spürbare Leistungsoptimierungen erreichen!

Beachten Sie, dass für jeden Datensatz mit einem Bestand in der Tabelle LAGERBEST ein Ergebnis zu finden ist. Allerdings nur für die Datensätze, zu denen in der dritten Tabelle LAGWERT ebenfalls ein entsprechender Eintrag existiert. Auch beim Einsatz von mehr als zwei Tabellen gilt folglich, dass der INNER JOIN das Vorhandensein der Datensätze in allen Tabellen erfordert, damit SQL diese in das Ergebnis stellt. Mit dieser Begründung erhalten wir das Subset an Datensätzen, das in der vorhergehenden Abbildung zu sehen ist. Einige der darin enthaltenen Artikel erscheinen „doppelt“. Das begründet sich damit, dass für einige Artikel mehrere Lagerorte einen Bestand aufweisen. Da die Spalte „Lagerort“ in der Ergebnisanzeige nicht vorgesehen ist, wirken die Ergebnisdatensätze quasi „doppelt“.

Tipp

Bei Mehrfachvorkommen gleichartiger Zeilen in den Ergebnissen einer SQL-Abfrage sollte man stets darum bemüht sein, etwaige Doppelausgaben klar zu verifizieren. Dazu gehört auch, dass wirkliche Doppelangaben klar zu erkennen sein müssen und nicht wie im zuvor gezeigten Beispiel der Eindruck doppelter Zeilen oder Datensätze suggeriert wird. Es gilt als Empfehlung, dass man in SQL-Ergebnissen alle Spalten einsehen können sollte, durch die die Eindeutigkeit einer Zeile klar erkennbar wird. Für unser Beispiel bedeutet das, dass mit dem Anzeigen der Spalte „Lagerort“ die Eindeutigkeit der Datensätze klar dargestellt ist, wie nachfolgende Abbildung zeigt.

7.6.3
Seite 16



Eindeutiges Ergebnis

7.6.3.1.2 Die USING-Klausel V5R3

Wenn man zwei oder mehr Tabellen miteinander verknüpft, heißt das nichts anderes, als dass man eine Mengenoperation ausführt. In relationalen Datenbanken sind vor allem die Konzepte der Vereinigungsmenge und der Schnittmenge wichtig. Wir konnten im vorangegangenen Kapitel sehen, dass sich so mit SQL problemlos Daten aus zwei oder mehr Tabellen miteinander verknüpfen lassen. Dabei liefert der INNER JOIN stets nur die Schnittmenge an Daten, die beiden Tabellen gemeinsam sind. Die Qualifizierung der Tabellen durch so genannte ALIASE ist hierbei immer dann notwendig, wenn in unterschiedlichen Tabellen Spalten identisch benannt sind. Mit Release V5R3 führte IBM die USING-Klausel ein, die die ALIASE überflüssig machte. Seither können wir die folgende Abfrage:

```
SELECT a.Firma, a.ArtikelNummer, Bezeichnung, Bestand
FROM Artikel a
INNER JOIN Lagerbestand l ON l.ArtikelNummer =
a.ArtikelNummer AND l.Firma = a.Firma;
```

auch kodieren, indem wir die Klausel USING verwenden. Schauen Sie sich die folgende Abbildung an:

```
SELECT Firma, ArtikelNummer, Bezeichnung, Bestand
FROM Artikel02
INNER JOIN Lagerbestand02   USING(Firma, ArtikelNummer) ;
```

FIRMA	ARTIKELNUMMER	BEZEICHNUNG	BESTAND
001	00001	Mauspad	4
001	00001	Mauspad	4
001	00002	Radiergummi (10)	50
001	00002	Radiergummi (10)	7
001	00002	Radiergummi (10)	50
001	00002	Radiergummi (10)	7
001	00004	Füllfederhalter	89
001	00005	Kugelschreiber (10)	9

Die Klausel USING

In der JOIN-Klausel kann im oberen Beispiel aufgrund der Klausel USING auf die Qualifizierung verzichtet werden, so dass die Bedingung sehr viel kürzer und meiner Meinung nach auch übersichtlicher formuliert werden kann. Etwas ärgerlich war bislang die Tatsache, dass Sie dennoch die Spalten qualifizieren mussten, wenn doppelt vorkommende Spaltennamen im Result Set als Ergebnisspalten verwendet wurden. Diese Restriktion wurde jedoch mit Release V6R1 aufgehoben, so dass Sie die oben gezeigte Abfrage im Release V6R1 problemlos ohne jede Qualifizierung verwenden können; selbst dann, wenn – wie in unserem Beispiel – die Artikel- und Firmennummer im Result Set benötigt wird.

7.6.3.1.3 Kreuzprodukt – Cross Join

Zu den sinnlosesten JOIN-Typen gehört für viele sicherlich der Cross Join, der oftmals auch als Kreuzprodukt oder kartesisches Produkt bezeichnet wird. Sicherlich werden Sie nicht häufig einen Cross Join in Ihren SQL-Abfragen einsetzen, aber es ist durchaus sinnvoll, das Konzept zu verstehen und in einigen Fällen kann es sogar praktisch sein, einen Cross Join zu verwenden.

Ein Cross Join ist ein Join, bei dem jede Zeile der Tabelle A mit jeder Zeile aus Tabelle B verknüpft wird. Schauen wir uns hierzu ein Beispiel an. Ausgangspunkt der Betrachtung sind die Tabellen HERSTELLER und ARTIKEL.

Tabelle Hersteller	
Herstellernummer	Herstellername
00001	Herlitz
00002	Bürodesign KG
00003	MAC Press

Tabelle ARTIKEL		
Artikelnummer	Bezeichnung	Lieferant
66111	Locher	00001
00003	Textmarker Rot (10)	00001
00010	Papier (250 Blatt)	00002
00002	Radiergummi (10)	00003

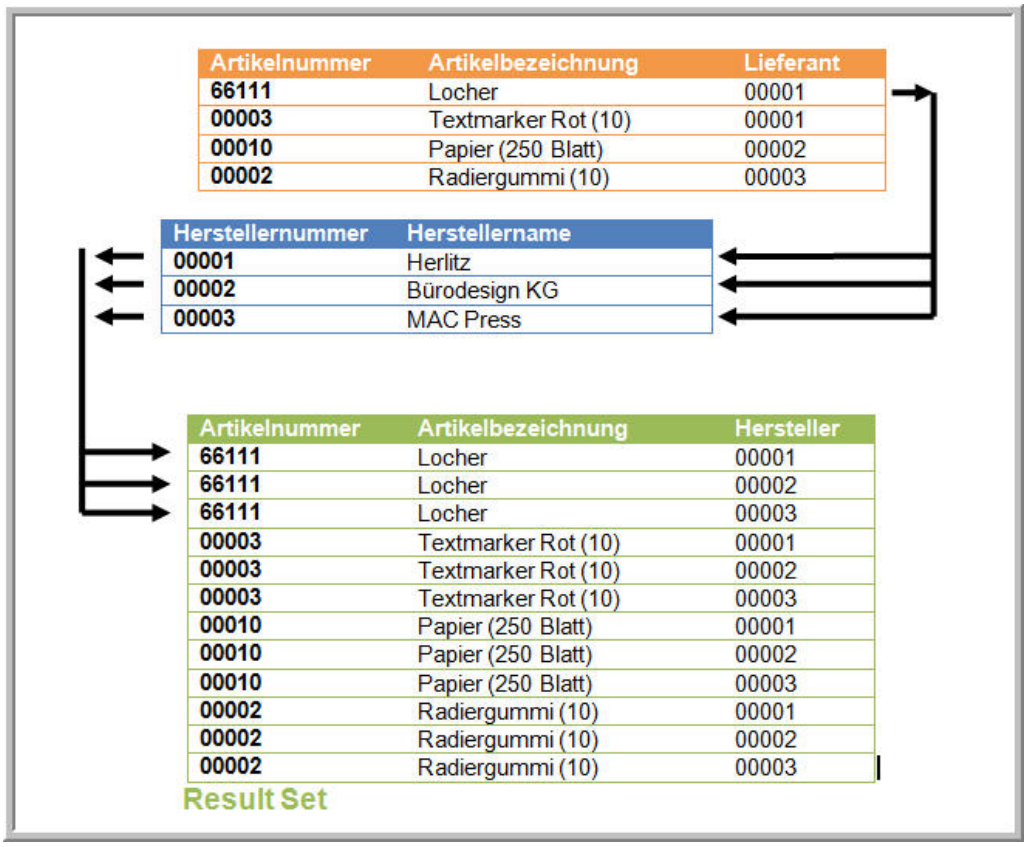
Die einfachste Methode, einen Cross Join zu formulieren besteht darin, die beiden Tabellen einfach in der FROM-Klausel zu spezifizieren, ohne entsprechende Join-Kriterien in der WHERE-Klausel zu verwenden.

```
SELECT Herstellername, Bezeichnung FROM Artikel, Hersteller
```

Alternativ könnten Sie auch die JOIN-Syntax verwenden:

```
SELECT Herstellername, Bezeichnung FROM Artikel CROSS JOIN
Hersteller
```

Unabhängig von der verwendeten Syntax wird diese Abfrage ausgeführt, indem zunächst die erste Zeile aus der Tabelle ARTIKEL gelesen und diese Zeile mit allen vorhandenen Zeilen der Tabelle HERSTELLER verknüpft wird. Anschließend wird mit der zweiten, dritten und vierten Zeile der Tabelle ARTIKEL ebenso verfahren. Da die Tabelle ARTIKEL über vier Zeilen und die Tabelle HERSTELLER über drei Zeilen verfügt, enthält das Result Set des Cross Join 4 * 3 Zeilen – also 12 Zeilen.



Wirkungsweise eines Cross Join

Sie haben recht, wenn Sie diese Abfrage für sinnlos halten. Sie sollte auch lediglich die Wirkungsweise eines Cross Joins dokumentieren.

Eine sinnvolle Einsatzmöglichkeit eines Cross Join zeigt ein anderes Beispiel: Wir wollen feststellen, mit wie vielen Artikeln unsere Lieferanten uns beliefern und dabei feststellen, wie viel Prozent des Gesamtsortiments durch den einzelnen Lieferanten abgedeckt werden. Das Ergebnis der Abfrage soll folgendermaßen aussehen:

Hersteller	Artikelanzahl	%-Anteil Sortiment
00002	9	25,71
00003	14	40,00
00001	12	34,29
Gesamt	35	100,00

7.6.3
Seite 20

Wie viele Artikel die jeweiligen Lieferanten liefern, kann schnell ermittelt werden. Allerdings benötige ich zur Lösung der Aufgabe zusätzlich die Tabelle ARTLIEF, da ein Artikel von verschiedenen Herstellern geliefert wird:

```
SELECT  Hersteller ,COUNT(Artikelnummer)
FROM Artikel    JOIN Artlief    USING(Artikelnummer)
GROUP  BY Hersteller;
```

Diese Abfrage liefert mir allerdings nur das folgende Ergebnis:

Hersteller	Artikelanzahl
00002	9
00003	14
00001	12

Um den Prozentwert zu ermitteln, benötigen wir die Artikelanzahl unseres gesamten Sortiments. Auch diese Abfrage ist schnell geschrieben:

```
SELECT   COUNT(Artikelnummer) G_ANZ
FROM Artikel
```

Doch jetzt müssen wir die folgende Formel für die dritte Spalte des Result Sets umsetzen:

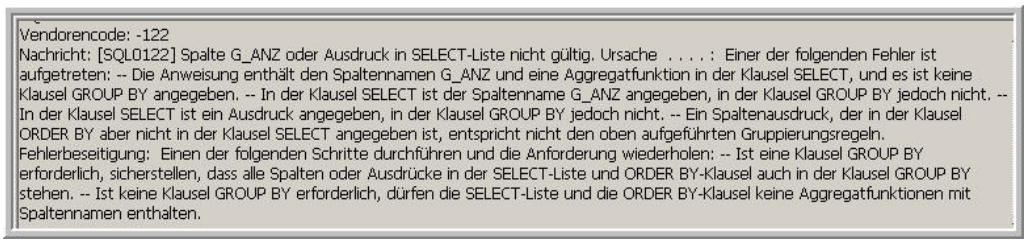
```
Artikelanzahl je Hersteller * 100 / Gesamtartikelanzahl
```

Wir benötigen daher zur Berechnung der Prozentwerte die Spalte G_ANZ. Wie können wir jetzt elegant die Spalte G_ANZ unserer Abfrage hinzufügen, um unsere Berechnung durchzuführen? Die Antwort ist ein Cross Join in Verbindung mit einer Unterabfrage.

```
SELECT
Hersteller  as "Hersteller",
COUNT(A.Artikelnummer) as  "Artikelanzahl"',
COUNT(A.Artikelnummer) * 100.00 / G_ANZ

FROM Artikel  A JOIN ARTLIEF AL USING(Artikelnummer)

CROSS JOIN (SELECT  COUNT(Artikelnummer) G_ANZ
            FROM Artikel  ) AS  Temp

GROUP  BY Hersteller, G_ANZ
```

Auf den ersten Blick sieht das Ganze wie „normales“ SQL aus. Doch achten Sie genau auf die FROM-Klausel dieser Abfrage. Anstatt nur den Tabellennamen anzugeben, enthält die FROM-Klausel zusätzlich eine SELECT-Anweisung. Ich verwende zur Lösung der Aufgabe eine Unterabfrage. Manchmal wird diese Technik auch mit dem Begriff „allgemeiner Tabellenausdruck“ umschrieben. Wenn Sie sich die Unterabfrage etwas genauer ansehen, werden Sie feststellen, dass mit der Abfrage lediglich die Gesamtanzahl unserer Artikel ermittelt wird. Die Unterabfrage steht stets in Klammern, und nachdem die Klammer geschlossen wird, erhält das Result Set der Unterabfrage den temporären Tabellennamen TEMP. Die Unterabfrage erstellt somit die virtuelle Tabelle TEMP mit der Spalte G_ANZ. Wenn Sie auf diesem Weg temporäre Tabellen erstellen, können Sie in der Hauptabfrage auf jede Spalte der temporären Tabelle verweisen.

Nun gehen wir einen Schritt weiter und verknüpfen diese virtuelle Tabelle TEMP über einen Cross Join mit der Tabelle ARTIKEL und der Tabelle ARTLIEF. Das Ganze kann nur funktionieren, weil die Tabelle TEMP lediglich über einen Datensatz verfügt. Nehmen wir an, dass in der Tabelle ARTIKEL 30.000 Zeilen vorhanden wären. Da in der Tabelle TEMP nur eine Zeile vorhanden ist, beträgt die Anzahl der Zeilen, die vom Cross Join zurückgegeben wird, 1 * 30.000 Zeilen. Anschließend sorgt die GROUP BY-Klausel dafür, dass die Datensätze aggregiert werden und der jeweilige Prozentwert ermittelt wird. Damit SQL allerdings in dieser Abfrage den Spaltennamen G_ANZ in der übergeordneten Abfrage akzeptiert, muss ich die GROUP BY-Klausel um diese Spalte ebenfalls erweitern, andernfalls würden Sie stets die folgende Fehlermeldung erhalten:


Vendorencode: -122
Nachricht: [SQL0122] Spalte G_ANZ oder Ausdruck in SELECT-Liste nicht gültig. Ursache . . . . : Einer der folgenden Fehler ist aufgetreten: -- Die Anweisung enthält den Spaltennamen G_ANZ und eine Aggregatfunktion in der Klausel SELECT, und es ist keine Klausel GROUP BY angegeben. -- In der Klausel SELECT ist der Spaltenname G_ANZ angegeben, in der Klausel GROUP BY jedoch nicht. -- In der Klausel SELECT ist ein Ausdruck angegeben, in der Klausel GROUP BY jedoch nicht. -- Ein Spaltenausdruck, der in der Klausel ORDER BY aber nicht in der Klausel SELECT angegeben ist, entspricht nicht den oben aufgeführten Gruppierungsregeln.
Fehlerbeseitigung: Einen der folgenden Schritte durchführen und die Anforderung wiederholen: -- Ist eine Klausel GROUP BY erforderlich, sicherstellen, dass alle Spalten oder Ausdrücke in der SELECT-Liste und ORDER BY-Klausel auch in der Klausel GROUP BY stehen. -- Ist keine Klausel GROUP BY erforderlich, dürfen die SELECT-Liste und die ORDER BY-Klausel keine Aggregatfunktionen mit Spaltennamen enthalten.

Die Fehlermeldung

Diese Aufgabe lässt sich auf viele verschiedene Arten lösen. Ein Lösungsweg ist der Cross Join, ein weiterer wird im Kapitel 7.6.5.2 und 7.6.5.4 beschrieben.

7.6.3.2 OUTER JOIN

Bei einem OUTER JOIN werden nicht nur die Datensätze angezeigt, die in direkter Beziehung zueinander stehen, sondern auch Datensätze, die keinen direkten Bezug zu Zeilen der anderen Tabelle haben. Dies ist dann nützlich, wenn man alle Datensätze aus einer gegebenen Tabelle anzeigen möchte, unabhängig davon, ob die verknüpfte Datei Daten enthält oder nicht. Bislang wurden lediglich alle Artikel im Result Set aufgeführt, für die ein Lagerbestand verfügbar ist. Was aber, wenn wir alle Artikel anzeigen wollen – auch jene, die derzeit nicht am Lager verfügbar sind? Das folgende SQL-Statement löst diese Aufgabe:

```
SELECT Firma, ArtikelNummer, Bezeichnung, Bestand
FROM Artikel
LEFT OUTER JOIN Lagerbestand USING(Firma, ArtikelNummer);
```

Die Abfrage unterscheidet sich lediglich in der JOIN-Verbindung, denn dieses Mal schreiben wir LEFT OUTER JOIN oder einfach nur LEFT JOIN.

FIRMA	ARTIKELNUMMER	BEZEICHNUNG	LAGERORT	BESTAND
001	00001	Textmarker	25	4
001	00001	Textmarker	10	4
001	00002	Radiergummi (10)	27	50
001	00002	Radiergummi (10)	33	7
001	00002	Radiergummi (10)	20	50
001	00002	Radiergummi (10)	13	7
001	00003	Textmarker Rot (10)	-	-
001	00004	Füllfederhalter	21	89
001	00005	Kugelschreiber (10)	22	9
001	00006	Hefter	-	-
001	00008	Kabelbinder	-	-

Das Ergebnis der LEFT OUTER JOIN Verbindung

Dieses Statement ähnelt zwar dem vorangegangenen INNER JOIN-Beispiel, aber in diesem Fall werden auch der Textmarker und die Hefter angezeigt, obwohl bei diesen Artikeln kein Lagerbestand vorhanden ist. Sie können erkennen, dass zunächst aufgrund des LEFT OUTER JOIN alle Zeilen aus der ersten, linken Tabelle ARTIKEL selektiert werden. Anschließend wird die Verbindung zur zweiten, rechten Tabelle LAGERBESTAND hergestellt. Dabei werden Spalten der Tabelle LAGERBESTAND auf den Wert NULL gesetzt, wenn es keine Entsprechung gibt.

Mit diesem Wissen können wir jetzt auch sehr schnell herausfinden, welche Lagerartikel derzeit nicht am Lager verfügbar sind:

```
SELECT Artnr, Bezeichnung,
Mindestbestand AS „Sollbestand“,
Bestand AS „Istbestand“
FROM Artikel
LEFT OUTER JOIN Lagerbestand USING(Firma, ArtikelNummer)
WHERE Beschaffung = ‚L‘
AND Bestand IS NULL;
```

ARTNR	BEZEICHNUNG	Sollbestand	Istbestand
00003	Textmarker Rot (10)	30	-
00008	Kabelbinder	900	-
66111	Locher	10	-

Nicht verfügbare Artikel am Lager

Stellen wir uns jetzt vor, dass wir überprüfen wollen, ob wir in unserer Lagerbestandsdatei Artikel führen, für die keine Stammdaten existieren. Natürlich sollte das Result Set bei dieser Abfrage keine Daten enthalten, weil wir es ansonsten mit inkonsistenten Daten zu tun hätten. Dennoch kann eine Überprüfung nicht schaden!

Wir müssen die zweite – also die rechte Tabelle LAGERBESTAND – vollständig anzeigen und die Stammdaten aus der linken Tabelle ARTIKEL ergänzen. Dies gelingt, indem wir die zuvor ausgeführte Abfrage verwenden, aber an Stelle der Klausel LEFT OUTER JOIN die Klausel RIGHT OUTER JOIN verwenden.

```
SELECT Firma, ArtikelNummer, Bezeichnung, Bestand
FROM Artikel
RIGHT OUTER JOIN Lagerbestand USING(Firma, ArtikelNummer);
```

Meine Daten sind tatsächlich konsistent, da die Abfrage keine Daten im Result Set zeigt, deren Artikelbezeichnung NULL ist:

FIRMA	ARTIKELNUMMER	BEZEICHNUNG	BESTAND
001	00001	Mauspad	4
001	00002	Radiergummi (10)	50
001	00002	Radiergummi (10)	7
001	00001	Mauspad	4
001	00002	Radiergummi (10)	50
001	00002	Radiergummi (10)	7
001	00004	Füllfederhalter	89
001	00005	Kugelschreiber (10)	9

Das Ergebnis der RIGHT OUTER JOIN-Anweisung

Natürlich erreiche ich das gleiche Ergebnis auch dann, wenn ich einfach die Reihenfolge der Tabellen in der Abfrage tausche:

```
SELECT Firma, ArtikelNummer, Bezeichnung, Bestand
FROM Lagerbestand
LEFT OUTER JOIN Artikel USING(Firma, ArtikelNummer);
```

Tipp:
Häufig haben Sie die Wahl zwischen einem LEFT und einem RIGHT-Join. Entscheiden Sie sich in diesem Fall immer für einen LEFT-JOIN. So ersparen Sie dem Optimizer Arbeit!

7.6.3.3 EXCEPTION-JOIN

In den Datenbanken vieler Unternehmen entsteht häufig „Datenmüll". Wie oft hat man schon davon gehört – oder es sogar selbst erlebt, dass während der Konvertierung zu einem neuen Softwareprodukt eine große Menge ungültiger Daten in der Datenbank auftauchen?

Was können Sie tun, um das bei Ihrer Datenbank zu verhindern? Sie könnten referenzielle Integritätsbedingungen verwenden, wie es im Kapitel 7.2.4.1 dieses Buchs beschrieben ist.

Falls Sie aber keine Datenbankbedingungen einsetzen können, um Ihre Datenbank vor Datenmüll zu schützen, sollten Sie zumindest von Zeit zu Zeit Ihre Daten einer Überprüfung unterziehen. LEFT und RIGHT EXCEPTION-JOIN sind großartige Anweisungen, um ungültige Daten ausfindig zu machen. In dem vorangegangenen Kapitel haben wir hierfür einen OUTER-JOIN zusammen mit einer entsprechenden WHERE-Bedingung eingesetzt, um alle Artikel ohne Lagerbestand ausfindig zu machen. Wir können diese Artikel aber auch mit der folgenden Abfrage ermitteln:

```
SELECT A.Firma, A.Artikelnummer, Bezeichnung, Bestand
FROM Artikel A
LEFT EXCEPTION JOIN Lagerbestand l on A.Firma = L.Firma
AND A.ArtikelNummer = L.Artikelnummer
ORDER BY A.Firma, A.Artikelnummer;
```

Bei Verwendung des LEFT EXCEPTION JOIN werden alle Datensätze der links von der LEFT EXCEPTION-Anweisung stehenden Tabelle verarbeitet. Dabei werden allerdings nur Zeilen selektiert, für die keine übereinstimmenden, „passenden" Datensätze innerhalb der rechts neben der Join-Anweisung definierten Tabelle gefunden werden. Feldinhalte der Tabelle, die auf der rechten Seite angegeben wurden, werden im Falle einer Nichtübereinstimmung mit NULL zurückgegeben.

FIRMA	ARTIKELNUMMER	BEZEICHNUNG	BESTAND
001	00003	Textmarker Rot (10)	-
001	00006	Hefter	-
001	00008	Kabelbinder	-
001	00010	Papier (250Blatt)	-
001	00012	Tip Ex Roller	-
001	00030	Papier 500 Blatt	-
001	00100	Schreibtisch	-
001	00200	Schreibtisch Eck	-
001	10000	SCHREIBTISCH	-
001	10001	Schnellhefter A4	-
001	10002		-
001	10100	Schuhschrank	-
001	10340	Bücherregal	-
001	44551	Notizblock	-
001	44552	Aktenvernichter Z100	-
001	66111	Locher	-
001	88000	Mauspad	-

Ergebnis der LEFT EXCEPTION JOIN Anweisung

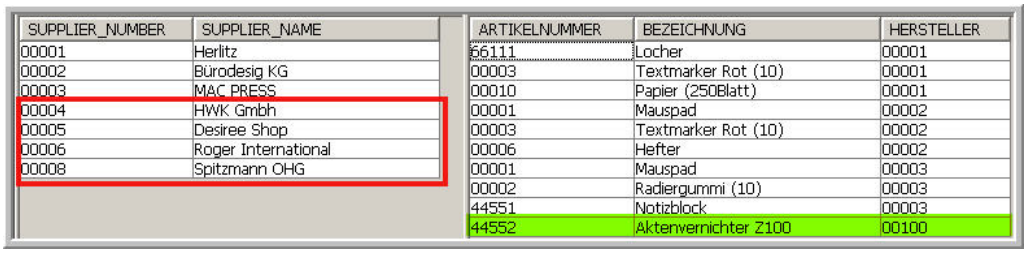
Der EXCEPTION JOIN ist für viele Überprüfungen nützlich. Prüfen Sie doch mal, ob jeder Bestellkopf entsprechende Bestellpositionen hat. Noch wichtiger wäre allerdings die Prüfung Ihrer Rechnungspositionen. Existiert für jede Rechnungsposition ein zugehöriger Rechnungskopf und können Sie für jeden Rechnungskopf auch einen Kunden finden?

Selbstverständlich können Sie auch beim EXCEPTION JOIN die USING-Klausel anstelle der im Beispiel verwendeten ALIASE nutzen:

```
SELECT Firma, Artikelnummer, Bezeichnung, Bestand
FROM Artikel
LEFT EXCEPTION JOIN Lagerbestand USING(Firma, Artikelnummer)
ORDER BY Firma, Artikelnummer;
```

7.6.3.4 FULL OUTER JOIN

Bei einem FULL OUTER JOIN handelt es sich um eine Kombination aus einem LEFT und einem RIGHT JOIN. Auch bei einem FULL OUTER JOIN sollen Datensätze verknüpft werden; nur werden hierbei sowohl alle Sätze der linken als auch alle Sätze der rechten Tabelle gesucht, für die es keine Beziehung gibt. Schauen Sie sich die folgende Abbildung an:



SUPPLIER_NUMBER	SUPPLIER_NAME
00001	Herlitz
00002	Bürodesig KG
00003	MAC PRESS
00004	HWK Gmbh
00005	Desiree Shop
00006	Roger International
00008	Spitzmann OHG

ARTIKELNUMMER	BEZEICHNUNG	HERSTELLER
66111	Locher	00001
00003	Textmarker Rot (10)	00001
00010	Papier (250Blatt)	00001
00001	Mauspad	00002
00003	Textmarker Rot (10)	00002
00006	Hefter	00002
00001	Mauspad	00003
00002	Radiergummi (10)	00003
44551	Notizblock	00003
44552	Aktenvernichter Z100	00100

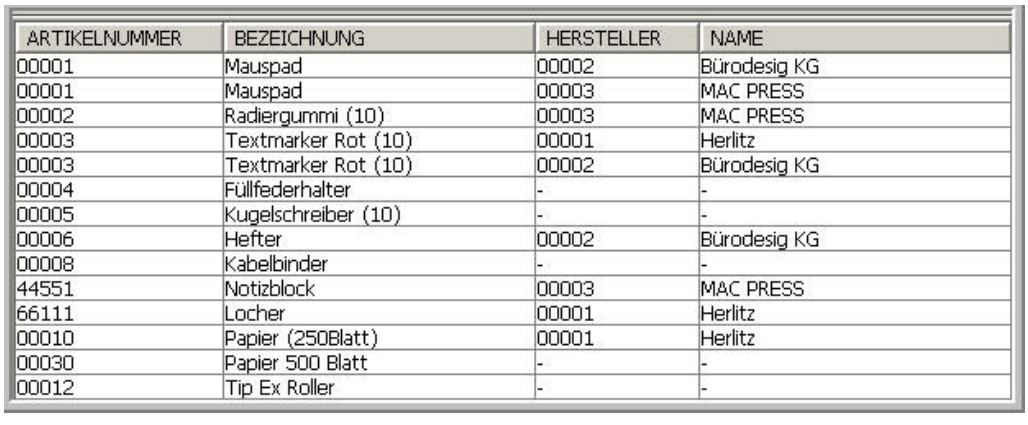
Hersteller-Artikelbeziehung

Achten Sie auf die Verbindungen. Es fällt auf, dass wir Artikel ohne Herstellerangaben, aber auch Hersteller ohne Artikel sehen. Bis zum Release V6R1 unterstützte die UDB DB2 kein FULL OUTER JOIN, daher musste man hier ein wenig tricksen:

Zunächst nutzen wir den LEFT OUTER JOIN aus dem Kapitel 7.6.3.2. Damit selektieren wir alle Artikel und, falls vorhanden, deren Hersteller. Wenn es Letzteres nicht gibt, so werden die Spalten HERSTELLER und NAME mit NULL aufgefüllt.

```
SELECT Artikelnummer, Bezeichnung, Hersteller, Name
FROM Artikel LEFT JOIN Hersteller USING(Artikelnummer);
```

Als Ergebnis erhalten wir folgendes Result Set:



ARTIKELNUMMER	BEZEICHNUNG	HERSTELLER	NAME
00001	Mauspad	00002	Bürodesig KG
00001	Mauspad	00003	MAC PRESS
00002	Radiergummi (10)	00003	MAC PRESS
00003	Textmarker Rot (10)	00001	Herlitz
00003	Textmarker Rot (10)	00002	Bürodesig KG
00004	Füllfederhalter	-	-
00005	Kugelschreiber (10)	-	-
00006	Hefter	00002	Bürodesig KG
00008	Kabelbinder	-	-
44551	Notizblock	00003	MAC PRESS
66111	Locher	00001	Herlitz
00010	Papier (250Blatt)	00001	Herlitz
00030	Papier 500 Blatt	-	-
00012	Tip Ex Roller	-	-

Artikel mit und ohne Herstellerangabe

Nun müssen wir auch noch alle Hersteller selektieren – egal ob Sie uns beliefern oder nicht:

```
SELECT Artikelnummer, Bezeichnung, Hersteller, Name
FROM Artikel RIGHT JOIN Hersteller USING(Artikelnummer);
```

Das Ergebnis ähnelt der vorangegangenen Abfrage, enthält aber zusätzlich auch Lieferanten, die uns nicht beliefern:

ARTIKELNUMMER	BEZEICHNUNG	HERSTELLER	NAME
66111	Locher	00001	Herlitz
00003	Textmarker Rot (10)	00001	Herlitz
00010	Papier (250Blatt)	00001	Herlitz
00001	Mauspad	00002	Bürodesig KG
00003	Textmarker Rot (10)	00002	Bürodesig KG
00006	Hefter	00002	Bürodesig KG
00001	Mauspad	00003	MAC PRESS
00002	Radiergummi (10)	00003	MAC PRESS
44551	Notizblock	00003	MAC PRESS
-	-	00004	HWK Gmbh
-	-	00005	Desiree Shop
-	-	00006	Roger International
-	-	00008	Spitzmann OHG

Hersteller mit und ohne Artikel

In einem weiteren Schritt müssen wir jetzt die beiden Ergebnismengen vereinen, d. h. wir verbinden die Result Sets zu einer einzigen Menge, wobei keine doppelten Zeilen vorhanden sein sollen.

Der UNION-Operator kann die Ergebnisse zweier Abfragen miteinander verknüpfen. Der Operator UNION gibt also die Ergebnisse zweier Abfragen abzüglich der doppelt vorkommenden Zeilen zurück. Das soll uns für den Moment als Erklärung ausreichen. Die Details schauen wir uns in einem späteren Kapitel an. Hier konzentrieren wir uns zunächst nur auf das Wesentliche.

```
SELECT Artikelnummer, Bezeichnung, Hersteller, Name
FROM Artikel LEFT JOIN Hersteller USING(Artikelnummer)

UNION

SELECT Artikelnummer, Bezeichnung, Hersteller, Name
FROM Artikel RIGHT JOIN Hersteller USING(Artikelnummer)
```

7.6.3
Seite 28

Das Ergebnis zeigt keine Überraschungen:

ARTIKELNUMMER	BEZEICHNUNG	HERSTELLER	NAME
00001	Mauspad	00002	Bürodesig KG
00001	Mauspad	00003	MAC PRESS
00002	Radiergummi (10)	00003	MAC PRESS
00003	Textmarker Rot (10)	00001	Herlitz
00003	Textmarker Rot (10)	00002	Bürodesig KG
00004	Füllfederhalter	-	-
00005	Kugelschreiber (10)	-	-
00006	Hefter	00002	Bürodesig KG
00008	Kabelbinder	-	-
44551	Notizblock	00003	MAC PRESS
66111	Locher	00001	Herlitz
00010	Papier (250Blatt)	00001	Herlitz
00030	Papier 500 Blatt	-	-
00012	Tip Ex Roller	-	-
00100	Schreibtisch	-	-
00200	Schreibtisch Eck	-	-
44552	Aktenvernichter Z100	-	-
10000	SCHREIBTISCH	-	-
10100	Schuhschrank	-	-
10340	Bücherregal	-	-
10001	Schnellhefter A4	-	-
10002		-	-
88000	Mauspad	-	-
-	-	00004	HWK Gmbh
-	-	00005	Desiree Shop
-	-	00006	Roger International
-	-	00008	Spitzmann OHG

Ergebnis eines FULL OUTER-JOIN bis Version 5.4

Seit Version 5.4 gibt es aber auch eine FULL OUTER JOIN Anweisung, so dass wir auf die Teilabfragen und den UNION-Operator verzichten können:

```
SELECT Artikelnummer, Bezeichnung, Hersteller, Name
FROM Artikel FULL OUTER JOIN Hersteller USING(Artikelnummer);
```

Die Syntax ist eine andere, das Ergebnis ist natürlich identisch.

7.6.3.5 Self JOIN

Dieses Kapitel zeigt Ihnen die Verwendung eines SELF JOIN. Unter einem SELF JOIN versteht man die Verbindung einer Tabelle mit sich selbst. Dieser Join-Typ ist wohl der am schwersten zu verstehende JOIN. Schauen wir uns daher eine SELF JOIN Verknüpfung an einem konkreten Beispiel an. In der Tabelle WARENGRUPPE ist jeder Warengruppe eine entsprechende Hauptwarengruppe zugeordnet. Wobei die Hauptwarengruppenbezeichnung selbst auch in dieser Tabelle gespeichert wurde:

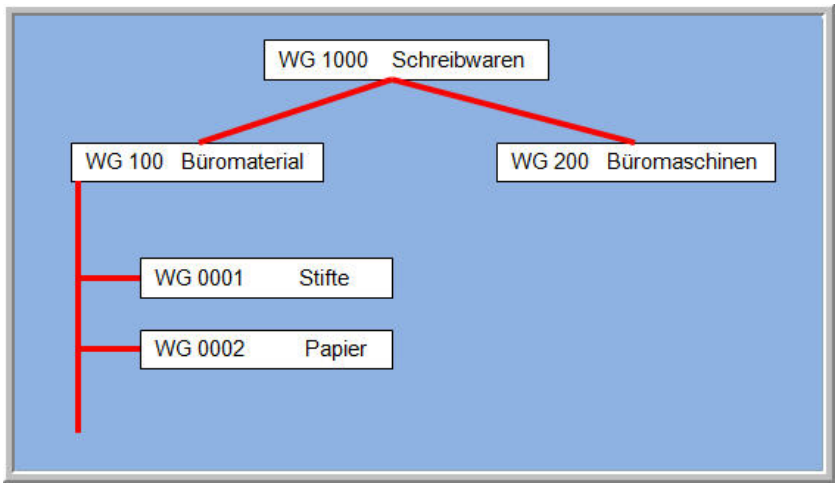
ITPSQL.WARENGRUPPE - 172.16.110.22(Epsserver)

Datei Bearbeiten Ansicht Zeilen Hilfe

WARENGRUPPE	WGBEZEICHNUNG	HAUPTWARENGRUPPE
1	Stifte	100
2	Papier	100
3	sonstiges	100
100	Büromaterial	1000
200	Büromaschinen	1000
10	Aktenvernichter	200
4	Ordner	100
20	Taschenrechner	200
1000	Schreibwaren	5000
2000	Büromöbel	2000
1500	Schreibtische	2000
1540	Schreibtisch Buche	1500

Die Tabelle WARENGRUPPE

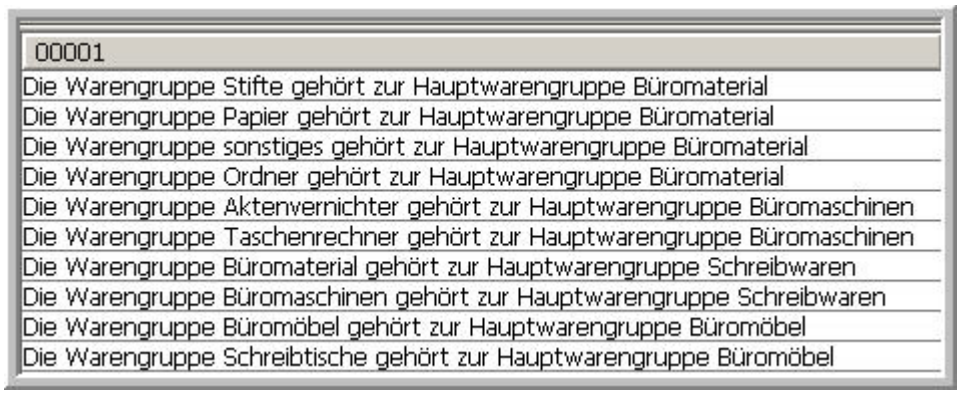
Verfolgt man die Einträge, ergibt sich folgende Hierarchie:



Warengruppenhierarchie

7.6.3
Seite 30

Jetzt wollen wir eine Abfrage schreiben, die das folgende Ergebnis liefert:



00001
Die Warengruppe Stifte gehört zur Hauptwarengruppe Büromaterial
Die Warengruppe Papier gehört zur Hauptwarengruppe Büromaterial
Die Warengruppe sonstiges gehört zur Hauptwarengruppe Büromaterial
Die Warengruppe Ordner gehört zur Hauptwarengruppe Büromaterial
Die Warengruppe Aktenvernichter gehört zur Hauptwarengruppe Büromaschinen
Die Warengruppe Taschenrechner gehört zur Hauptwarengruppe Büromaschinen
Die Warengruppe Büromaterial gehört zur Hauptwarengruppe Schreibwaren
Die Warengruppe Büromaschinen gehört zur Hauptwarengruppe Schreibwaren
Die Warengruppe Büromöbel gehört zur Hauptwarengruppe Büromöbel
Die Warengruppe Schreibtische gehört zur Hauptwarengruppe Büromöbel

Das Ergebnis des SELF-JOINs

Wir müssen also für jede Warengruppe die zugehörige Hauptwarengruppe ermitteln, wobei wir uns nicht mit der Nummer zufrieden geben, sondern die Hauptwarengruppenbezeichnung darstellen wollen.

Als erstes müssen wir die Tabelle mit sich selbst verknüpfen. Dabei ist es hilfreich, wenn Sie sich vorstellen, dass tatsächlich zwei Tabellen – die Tabelle U1 und H1 – existieren. Dies gelingt Ihnen sehr leicht, wenn Sie mit ALIASE arbeiten:

```
SELECT…
FROM warengruppe U1   JOIN warengruppe H1
ON     U1.Hauptwarengruppe =      H1.Warengruppe
```

Nun stellen wir uns vor, dass in der Tabelle U1 alle Untergruppen und in der Tabelle H1 alle Hauptgruppen zu finden sind. Daher müssen wir in der Tabelle U1 die Spalte HAUPTWARENGRUPPE nutzen, um die zugehörige Hauptwarengruppenbezeichnung in der Tabelle H1 zu ermitteln.

Anschließend stellen wir die Informationen im Result Set dar:

```
SELECT ,Die Warengruppe ,
|| U1.WGBezeichnung ||
‘gehört zur Hauptwarengruppe , ||
H1.WGBezeichnung

FROM Warengruppe U1 JOIN Warengruppe H1
ON U1.Hauptwarengruppe = H1.Warengruppe
```

Sie haben Recht, wenn Sie vermuten, dass derartige Abfragen nicht alltäglich sind, aber die SELF JOIN werden wir später ebenfalls verwenden, um Rekursionen zu kodieren, um bspw. Stücklisten aufzulösen. Außerdem kann man derartige Verknüpfungen sehr gut benutzen, um die interne Konsistenz der Daten zu überprüfen. Was würde passieren, wenn versehentlich ein Kunde doppelt mit unterschiedlichen Kundennummern erfasst wird?

KDKDNR	KDNAM1	KDNAM2	KDLKZ	KDPLZ
7	Kunsthandlung Petersen ...	-	D	28650
400	Kunsthandlung Petersen ...	-	D	28650

Doppelt erfasste Kunden

Um diese Ausgabe zu erzeugen, müssen Sie entweder wissen, welche Kunden doppelt erfasst wurden oder Sie nutzen einen SELF JOIN zur Überprüfung. Auch dieses Mal stellen wir uns einfach vor, wir hätten wieder zwei Kundentabellen:

```
FROM   Kunden K1       JOIN  Kunden K2
```

Nun prüfen wir, ob in der Kundentabelle K2 Einträge mit dem gleichen Namen und der gleichen Postleitzahl vorhanden sind wie in der Tabelle K1. Selbstverständlich können weitere Kriterien wie Straße und Hausnummer zum Vergleich herangezogen werden:

```
ON K1.Kdnam1 = K2.KdNam1 AND K1.kdplz = k2.kdplz
```

Wichtig ist jetzt nur noch, dass die Einträge eine unterschiedliche Kundennummer haben müssen, denn sonst wäre jeder Eintrag doppelt!

```
AND k1.kdkdnr <> k2.kdkdnr
```

Das Ergebnis zeigt, dass nicht nur der Kunde auf Westerland doppelt erfasst wurde:

```
Select k1.kdkdnr, k1.kdnam1, k1.kdort
from Kunden K1 JOIN Kunden K2 ON K1.Kdnam1 = K2.KdNam1 and K1.kdplz = k2.kdplz
and k1.kdkdnr <> k2.kdkdnr
order by k1.kdort desc;
```

KDKDNR	KDNAM1	KDORT
2095390	Zimmermann-Graeff & Müller	Zell
2095390	Zimmermann-Graeff & Müller	Zell
2095390	Zimmermann-Graeff & Müller	Zell
1157289	Benninger Zell GmbH	Zell
1157289	Benninger Zell GmbH	Zell

Auch andere Kundendaten wurden doppelt erfasst

Auch andere Einträge sind doppelt vorhanden. In meinem Testdatenbestand ist dies gewollt, bei Ihnen sollten doppelte Einträge sicherlich nicht vorkommen.

7.6.3.6 Non-Equi-Joins

Die in den vorangegangenen Kapiteln gezeigten Beispiele waren allesamt Equi-Joins. Wenn aber Equi-Joins unterstützt werden, könnte man auch davon ausgehen, dass es auch Nicht-Equi-Joins gibt. Diese Annahme stimmt genau! Während beim Equi-Join das Gleichheitszeichen in der WHERE-Anweisung oder der JOIN-Klausel steht, verwendet man beim Non-Equi-Join alle anderen Operatoren – außer dem Gleichheitszeichen. Sehen Sie sich folgendes Beispiel an:

```
SELECT a.Firma, a.ArtikelNummer, Bezeichnung, Bestand
FROM Artikel a
INNER JOIN Lagerbestand l ON l.ArtikelNummer > a.ArtikelNummer
```

FIRMA	ARTIKELNUMMER	BEZEICHNUNG	BESTAND
001	00001	Mauspad	50
001	00001	Mauspad	7
001	00001	Mauspad	50

NON-EQUI-JOIN

Die Liste setzt sich noch weiter fort und beschreibt alle Zeilen in der Verknüpfung entsprechend der Bedingung. Sie haben Recht. Das Ergebnis ist unsinnig! In der Praxis findet man daher auch Equi-Joins weitaus häufiger als Non-Equi-Joins. Trotzdem gibt es Anwendungen, in denen gerade ein Non-Equi-Join die perfekte Konstruktion darstellt.

Stellen wir uns vor, dass es eine Tabelle gibt, die unsere Rabattaktionen enthält:

ITPSQL.RABATTAKTION - 172.16.110.22(Epsserver)

Datei Bearbeiten Ansicht Zeilen Hilfe

	FROMDATE	TODATE	DISCOUNT
	2010-05-01	2010-05-15	.050
	2010-07-01	2010-07-31	.100
	2010-01-01	2010-01-31	.100
	2010-11-01	2010-12-31	.200

Die Tabelle RabattAktion

Um festzustellen, zu welchem Preis wir derzeit unsere Artikel verkaufen, müssen wir auf die Tabelle RABATTAKTION zugreifen, um den heute gültigen Rabattsatz zu ermitteln. Ein EQUI JOIN führt hier nicht zum gewünschten Erfolg, da das Tagesdatum ggf. nicht in der Tabelle zu finden ist. Wir benutzen also einfach das Tagesdatum und vergleichen dieses in der Verknüpfung auf größer gleich (>=) bzw. auf kleiner gleich (<=). Alternativ hätte ich selbstverständlich auch den Operator BETWEEN verwenden können, um das folgende Ergebnis zu erhalten:

```
SELECT
Bezeichnung,
EPreis as "Regulärer Preis",
EPreis * Discount as "Rabatt",
EPreis - (EPreis * Discount) as "Aktionspreis"
FROM   Artikel  a
INNER  JOIN Lagerwert Using(Firma, Artikelnummer)
JOIN RABATTAKTION ON curDate() >=FromDate and Curdate() <= ToDate;
```

BEZEICHNUNG	Regulärer Preis	Rabatt	Aktionspreis
Radiergummi (10)	10.00	2.00000	8.00000
Textmarker Rot (10)	5.12	1.02400	4.09600
Füllfederhalter	6.00	1.20000	4.80000

NON-EQUI-JOIN Bsp.02

7.6.3
Seite 34

7.6.4 Vertikale Verknüpfungen – der Operator UNION

Die in den vorangegangenen Kapiteln besprochenen Verknüpfungen können als horizontale Verknüpfungen beschrieben werden. Das heißt, dass Spalten aus verschiedenen Tabellen innerhalb desselben Datensatzes miteinander verknüpft werden. Eine andere Methode erzeugt eine vertikale Verbindung (=Union). Solch eine UNION-Verknüpfung bezeichnet man auch als Full-Select. Bei einem Full-Select werden mehrere Sub-Selects miteinander verbunden. Im Kapitel 7.6.3.4 konnten Sie sehen, dass die einzelnen Sätze nicht miteinander verknüpft, sondern die Zeilen, die sich aus jedem einzelnen Sub-Select ergeben, gemeinsam als ein Ergebnis ausgegeben werden. Der Operator UNION kennt verschiedene Ausprägungen:

UNION ALL
Alle Sätze aus dem ersten und dem zweiten Result Set werden verbunden. Doppelte Sätze werden übernommen.

UNION
Alle Sätze, die entweder aus dem ersten oder dem zweiten Sub-Select resultieren, werden zu einer Menge vereint. Dabei werden doppelte Sätze eliminiert.

EXCEPT
Alle Sätze, die nur im ersten Sub-Select und nicht im zweiten zu finden sind, werden in die Ergebnismenge übernommen.

INTERSECT
Alle Sätze, die sowohl in dem ersten, als auch im zweiten Sub-Select vorhanden sind, werden in die Ergebnismenge übernommen.

Schauen wir uns die verschiedenen Ausprägungen jetzt im Detail an, denn einige Regeln gilt es auch hier zu beachten.

In unserer Datenbank wurden die Vertriebsmitarbeiter aufgrund eines individuellen Abrechnungssystems in der Tabelle VERTRIEB, alle anderen Mitarbeiter in der Tabelle MITARBEIT gespeichert.

Ergänzung 16./2011

7.6.4
Seite 2

Die Tabelle MITARBEIT

ID	NAME	DEPT	JOB	YEARS	SALARY	MGRH
10	Sanders	20	Clerk	7	18357.50	40
30	Marenghi	38	Clerk	5	17506.75	80
60	Habers	15	Mgr	10	20659.80	80
70	Norgreen	20	Clerk	-	13504.60	150
90	Plözke	42	Mgr	7	18352.80	-
125	Nemecek	42	Clerk	6	10505.90	-
160	Molinare	10	Mgr	7	21000.00	-
175	Achtes	15	Clerk	4	12258.50	-

Die Tabelle VERTRIEB

ID	NAME	DEPT	JOB	YEARS	SALARY	BIRTHDAY	BONUS
10	Sanders	20	Sales	7	15607.53	1950-04-14	1100.00
30	Marenghi	38	Sales	5	30010.00	1950-05-10	0.00
50	Hanes	15	Sales	10	40659.80	1955-08-14	0.00
80	James	20	Sales	-	10405.02	1967-12-24	15000.00
100	Plotz	42	Sales	7	18352.80	1960-03-15	0.00
110	Ngan_Khan	15	Sales	5	12508.20	1970-03-15	0.00
120	Naughton	38	Sales	-	10954.75	1960-03-30	0.00
130	Yamaguchi	42	Sales	6	14505.90	1987-03-15	0.00
140	Fraye	51	Sales	6	21150.00	1990-01-17	0.00
150	Williams	51	Sales	6	19456.50	1950-04-14	500.00

Die Tabellen VERTRIEB und MITARBEIT

In der Abbildung können Sie erkennen, dass es Mitarbeiter gibt, die nur im Vertrieb tätig sind; andere arbeiten ausschließlich in der Verwaltung und einige Mitarbeiter sind sowohl der Verwaltung als auch dem Vertrieb zugeordnet.

Um nun eine vollständige Personalliste zu erstellen, können wir folgende SQL-Anweisung verwenden:

```
SELECT id, name FROM Vertrieb

UNION ALL
SELECT id, name FROM Mitarbeit
```

Sie sehen, im Result Set werden alle Mitarbeiter aufgeführt.

ID	NAME
10	Sanders
10	Sanders
30	Marenghi
30	Marenghi
50	Hanes
60	Habers
70	Norgreen
80	James
90	Plözke

Das Ergebnis „UNION ALL“

Das UNION ALL-Schlüsselwort verbindet die beiden Select-Statements. Das ALL-Schlüsselwort zeigt an, dass doppelte Datensätze im Result Set erlaubt sind. Verzichte ich auf das Schlüsselwort ALL, werden die Duplikate ausgeschlossen:

ID	NAME
10	Sanders
30	Marenghi
50	Hanes
60	Habers
70	Norgreen
80	James
90	Plözke
100	Plotz

Das Ergebnis „UNION“

Der oben erzeugte Union ist sortiert. Am Ende aller UNION-Statements können Sie eine ORDER BY-Klausel einfügen, um die Ergebnisse zu sortieren. Das Problem dabei ist, dass die ORDER BY-Klausel normalerweise Spaltennamen benötigt. Das wird bei Unions nicht funktionieren, weil die Spaltennamen zwischen den SELECT-Statements, die miteinander verknüpft wurden, möglicherweise nicht übereinstimmen. Sie müssen in diesen Fällen anhand der Spaltenposition sortieren. Wir wissen bereits aus den vorangegangenen Kapiteln, dass der Positionswert 1 der ersten Spalte im Result Set entspricht; der Positionswert 2 ist die zweite Spalte usw. Um die vorherige Anweisung mit Sortierung nach den Namen der Angestellten auszugeben, müssten wir also das folgende Statement verwenden:

```
SELECT id, name FROM Vertrieb
UNION ALL
SELECT id, name FROM Mitarbeit

ORDER BY 2;
```

Eines müssen Sie beim Einsatz des UNION-Operators allerdings beachten: Jedes Select-Statement muss die gleiche Anzahl an Spalten und einen gleichen Grunddatentyp für die entsprechenden Spalten haben. Der exakte Datentyp und die exakte Datenlänge sind dabei nicht entscheidend. Beispielsweise könnte eine Mengenspalte in einer Tabelle dezimal gepackt und in einer anderen Datei dezimal gezont sein. Die SQL-Engine handhabt diese Situation ohne Probleme. Was passiert jedoch, wenn eine der Tabellen eine Spalte enthält, die JAHR heißt und das Datum in einer numerischen Spalte mit vier Ziffern speichert und die andere Datei, die über UNION verknüpft wird, JAHR als vier-Byte-alphanumerisches Feld speichert? In vielen Fällen konvertiert SQL mittlerweile die Daten automatisch. Andernfalls müssen Sie die Daten explizit konvertieren, um die Durchführung der Union-Klausel zu gewährleisten.

7.6.4
Seite 4

Anders verhält es sich jedoch, wenn in einer der betreffenden Tabellen eine Spalte fehlt. Der Umgang mit einer fehlenden Spalte wird ermöglicht, indem ein Platzhalter im SELECT eingesetzt wird. Enthält beispielsweise die Tabelle VERTRIEB eine Spalte BONUS, die die Bonuszahlung enthält, könnte diese Spalte in der Tabelle MITARBEIT fehlen. Um dennoch die Spalte BONUS im Result Set darzustellen, muss eine Konstante verwendet werden, die mit der entsprechenden Spalte korrespondiert:

```
SELECT Id, Name, Salary, Bonus
FROM Vertrieb

UNION ALL

SELECT Id, Name, Salary, 0
FROM Mitarbeit
ORDER BY 1;
```

Wenn die nicht korrespondierende Spalte – anders als im Beispiel – hingegen eine alphanumerische Spalte ist, verwendet man selbstverständlich eine alphanumerische Konstante, um den Standardwert zu bezeichnen.

So weit so gut. Nun stellen wir uns vor, dass wir nur alle Mitarbeiter selektieren wollen, die ausschließlich im Vertrieb arbeiten. Wir benötigen somit alle Zeilen, die in der Tabelle VERTRIEB gespeichert sind, aber nicht in der Tabelle MITARBEIT zu finden sind. Hier hilft der Operator EXCEPT. Mit dem Operator EXCEPT können wir alle Zeilen selektieren, die nur im ersten Sub-Select und nicht im zweiten zu finden sind.

```
SELECT Id, Name
FROM Vertrieb

EXCEPT

SELECT Id, Name
FROM Mitarbeit
ORDER BY 1;
```

ID	NAME
50	Hanes
80	James
100	Plotz
110	Ngan_Khan
120	Naughton
130	Yamaguchi
140	Fraye

Der Mitarbeiter SANDERS fehlt im Result Set, da er als kaufmännischer Mitarbeiter und als Vertriebsmitarbeiter geführt

Ergebnis der Abfrage mit dem Operator EXCEPT

Wenn Sie stattdessen nur die Mitarbeiter selektieren wollen, die als Vertriebsmitarbeiter und gleichzeitig auch als Büroangestellte arbeiten, müssen Sie den Operator INTERSECT verwenden. Die SQL-Abfrage sieht dann folgendermaßen aus:

```
SELECT Id, Name  FROM   Vertrieb

INTERSECT

SELECT Id, Name  FROM Mitarbeit
ORDER BY 1;
```

ID	NAME
10	Sanders
30	Marenghi
160	Molinare
230	Lundquist
250	Wheeler
290	Quill's
330	Burke

Wirkungsweise des Operators INTERSECT

Mit Hilfe des Operators INTERSECT werden alle Mitarbeiter selektiert, die sowohl in der Tabelle VERTRIEB als auch in der Tabelle MITARBEIT gespeichert sind. Zeilen, die nur in einer der beiden Tabellen vorkommen, werden im Result Set nicht berücksichtigt.

7.6.4
Seite 6

7.6.5 Der Einsatz von Unterabfragen

Mit jeder JOIN-Variante lassen sich spezielle Anforderungen realisieren und für viele Aufgabenstellungen sind die Join-Abfragen ausreichend. Allerdings gibt es Aufgabenstellungen, die sich mit den bisher vorgestellten Techniken nicht so einfach realisieren lassen. Vielleicht müssen wir Daten manipulieren? Eine Update-Anweisung kann nicht mit JOIN-Verbindungen realisiert werden. In diesen Fällen müssen wir Unterabfragen verwenden, um zum Ziel zu gelangen.

Unterabfragen, die meistens als Subselects bezeichnet werden, werden dann verwendet, wenn das Ergebnis einer Abfrage herangezogen wird, um eine neue Abfrage zu erstellen. Unterabfragen werden deshalb in WHERE-Klauseln, in HAVING-Klauseln und als Tabellen-Ersatz in der FROM-Klausel genutzt. Seit dem Release V5R1 ist es aber auch möglich, Unterabfragen in den Spaltenlisten zu verwenden.

Für die Unterabfragen benötigen wir keine neuen Schlüsselworte oder Klauseln. Die bereits bekannten Klauseln SELECT, FROM, WHERE, GROUP BY und HAVING sind ausreichend. Die Schlüsselworte SELECT und FROM sind für jede Unterabfrage erforderlich, während die anderen Klauseln optional sind und je nach Bedarf angegeben werden können.

```
SELECT spalte1, spalte2, (SELECT...FROM...), spalte3      ---als temporäre Spalte
FROM
        (SELECT...FROM...) as Tabellenname                ---als temporäre Tabelle

WHERE Bedingung (SELECT...FROM...)                        ---als variable Bedingung

GROUP BY
HAVING    (SELECT...FROM...)                              ---als variablen Gruppenfilter

ORDER BY
```

Einsatz von Unterabfragen

7.6.5.1 Skalare Unterabfrage – eine zurückgegebene Zelle

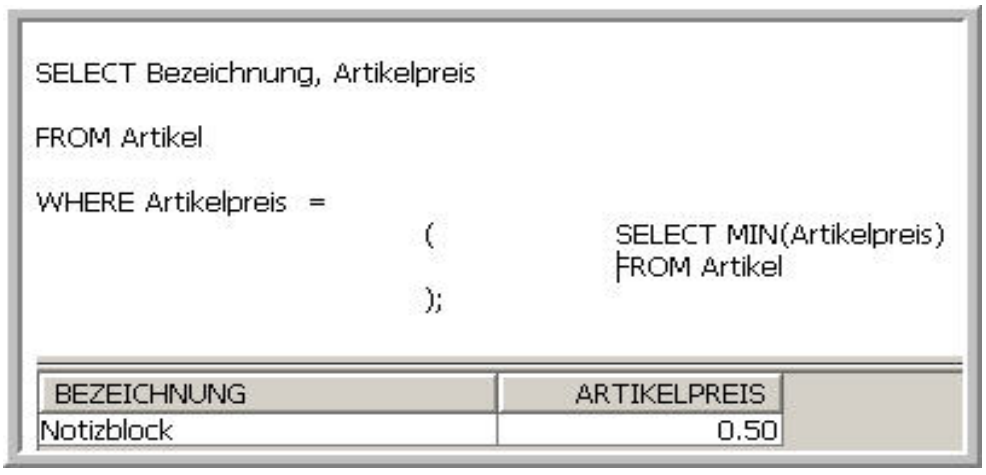
In der elementarsten Version einer Unterabfrage liefert die Unterabfrage eine Zelle zurück und ist unabhängig von der aufrufenden Abfrage. Sie wird verwendet, um Details zu einem Datensatz zu ermitteln. Nehmen wir an, dass der günstigste Artikel des gesamten Sortiments gesucht wird.

```
SELECT  MIN(Artikelpreis)
FROM Artikel;
```

Als Ergebnis erhalten wir den kleinsten Verkaufspreis. Wir erfahren aber nicht, welcher Artikel zu diesem Preis verkauft wird. Hier hilft eine Unterabfrage:

```
SELECT Bezeichnung, Artikelpreis
FROM Artikel
WHERE Artikelpreis =
                      (     SELECT MIN(Artikelpreis)
                            FROM Artikel
                      );
```

Die Unterabfrage wird als erstes aufgelöst. Sie ermittelt den niedrigsten Verkaufspreis. In unserem Beispiel sind dies 0,50 Euro. Dieser Wert wird anschließend verwendet, um die Bedingung der übergeordneten Abfrage auszuführen. Das Result Set enthält anschließend die Artikelbezeichnung und den Artikelpreis.

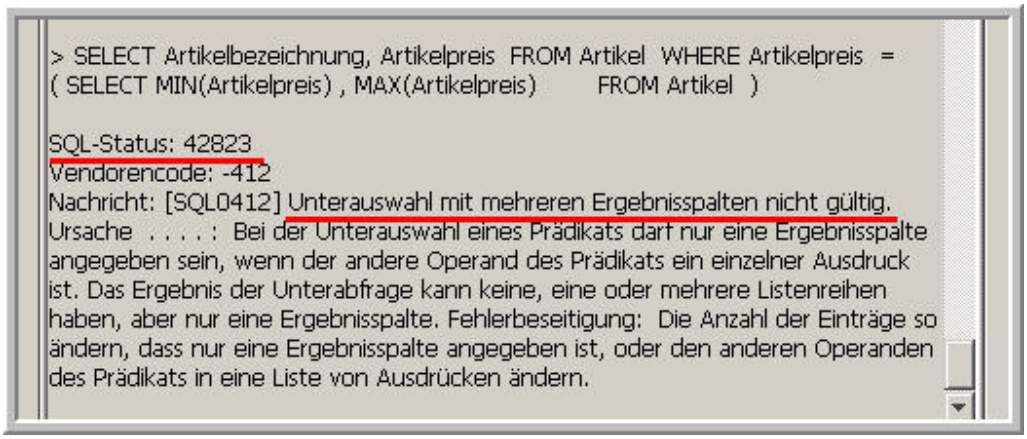


Wirkungsweise skalarer Unterabfragen

Unterabfragen, die in der WHERE-Bedingung verwendet werden, dürfen auch mehr als einen Wert ermitteln. Stellen Sie sich vor, dass der teuerste und der günstigste Artikel ermittelt werden sollen:

```
SELECT Bezeichnung, Artikelpreis
FROM Artikel
WHERE Artikelpreis=
            (  SELECT MIN(Artikelpreis) , MAX(Artikelpreis)
               FROM Artikel );
```

Die Unterabfrage ermittelt im zweiten Beispiel den niedrigsten und den höchsten Verkaufspreis. Die Unterabfrage ist korrekt formuliert und dennoch wird die Abfrage nicht ausgeführt, wie folgende Grafik zeigt:



Der SQL-Status 42823

Grund für die Fehlermeldung ist der verwendete Vergleichsoperator. Wird das Gleichheitszeichen als Vergleichsoperator eingesetzt, darf die Unterabfrage nur einen Wert zurückgeben. Wir könnten daher versuchen, den Operator IN anstelle des Gleichheitszeichens zu verwenden, damit die Abfrage fehlerlos ausgeführt wird.

```
SELECT Bezeichnung, Artikelpreis
FROM Artikel
WHERE Artikelpreis          IN
               (SELECTMIN(Artikelpreis) , MAX(Artikelpreis)
               FROM Artikel );
```

Ergänzung 17/2011

7.6.5
Seite 4

Es ist tatsächlich möglich, die Operatoren IN, BETWEEN oder das Größer- und Kleinerzeichen zu verwenden. Doch bei unserem Beispiel würde die Abfrage ebenfalls nicht ausgeführt werden, weil links vom Operator lediglich eine Spalte aufgeführt wird und darum das Result Set auch nur eine Spalte enthalten darf und nicht wie im oben aufgeführten Beispiel zwei Spalten.

Wollen wir hingegen alle Artikel ermitteln, die zum Auftrag 00201 gehören, ist dies mit den Operator IN problemlos möglich:

```
SELECT Bezeichnung, Artikelpreis
FROM Artikel
WHERE Artikelnummer   IN
                            ( SELECT Artikelnummer
       FROM Auftragsposition WHERE Auftragsnummer = '00201')
```

Aufgelöst wird hier wiederum zunächst die Unterabfrage, die das folgende Ergebnis liefern wird:

Artikelnummer
00001
00012

Dieses Result Set kann jetzt problemlos für die Hauptabfrage genutzt werden:

```
SELECT Bezeichnung, Artikelpreis
FROM Artikel
WHERE Artikelnummer IN
```

00001
00012

Und führt dann zu folgendem Endergebnis:

Bezeichnung	Artikelpreis
Mauspad	2.00
Tip Ex Roller	5.20

Bis Release V5R3 konnte nur eine Spalte in einer Unterabfrage mit einem IN-Prädikat verwendet werden. Wenn mehrere Spalten erforderlich waren, mussten diese über Verkettungsoperatoren entsprechend miteinander verbunden werden. Falls die Spalten dann auch noch unterschiedliche Datentypen hatten, war zusätzlich die Verwendung von diversen skalaren SQL-Funktionen erforderlich, um den Vergleichsstring aufzubauen. Das folgende Beispiel zeigt eine solche Abfrage:

```
SELECT * FROM Mitarbeit
WHERE Persnr CONCAT Name CONCAT Dept IN
                (
                SELECT Id CONCAT Name CONCAT Dept
                FROM Vertrieb
                );
```

Mit der Abfrage sollen alle Mitarbeiter gefunden werden, die auch in der Tabelle VERTRIEB gespeichert sind. Für den Vergleich sind die Personalnummer, der Mitarbeitername und die Abteilungsnummer notwendig. Weil nur eine Spalte in einem Subselect bis Release V5R3 erlaubt war, mussten die verschiedenen Spalten miteinander verkettet werden. Nicht nur die Verkettung ist aufwendig, häufig war der Zugriff auf die Daten mit einer solchen Abfrage auch sehr langsam, weil es dem Query-Optimizer bei dieser Abfrage nicht möglich ist, einen vorhandenen Zugriffspfad auszuwählen, da bis Release V6R1 Zugriffspfade nur über die Originalspalten einer Tabelle gebildet werden konnten. Durch die Verwendung der Verkettungsoperatoren wird aber eine neue temporäre Spalte gebildet, für die es keinen Zugriffspfad gibt und für die auch bis Release V6R1 kein Zugriffspfad angelegt werden kann.

Seit Release V5R4 können wir diese Abfrage auch anders formulieren. Die zu vergleichenden Spalten können direkt durch Komma getrennt aufgelistet werden, müssen allerdings in Klammern eingeschlossen werden.

```
SELECT * FROM Mitarbeit
WHERE (Persnr , Name , Dept )     IN
                     (SELECT Id , Name , Dept FROM
                      Vertrieb);
```

7.6.5
Seite 6

Diese Änderung bringt nicht nur eine einfachere Codierung mit sich, sondern gibt dem Optimizer auch die Chance, einen vorhandenen Zugriffspfad zu verwenden, sodass die Abfrage sehr viel schneller ausgeführt wird.

Unterabfragen können in dieser Form in der WHERE und HAVING-Klausel eingesetzt werden und durchaus sehr komplexe Formen annehmen: Eine Abfrage kann bis zu 255 Unterabfragen enthalten und die Verschachtelungstiefe kann 32 Ebenen umfassen.

7.6.5.2 Temporäre Spalten und temporäre Tabellen erstellen

Unterabfragen können Sie aber nicht nur in der WHERE-Klausel oder der HAVING-Klausel einsetzen, sondern auch auf Spaltenebene. Wenn Sie beispielsweise den Durchschnitts- und Gesamtumsatz der einzelnen Artikel ermitteln wollen, geht das sehr einfach:

```
SELECT Artikelnummer,
AVG(Artikelpreis * Menge) as "Durchschnittsumsatz je
          Artikel",

SUM(artikelpreis * Menge) AS "Umsatz je Artikel"

FROM      Auftragsposition
GROUP BY  Artikelnummer
ORDER BY  Artikelnummer
```

Das Ergebnis sieht folgendermaßen aus:

ARTIKELNUMMER	Durchschnittsumsatz je Artikel	Umsatz je Artikel
00001	151.428571428571428571428571	1060.00
00002	111.800000000000000000000000	670.80
00003	191.250000000000000000000000	1147.50
00004	705.732857142857142857142857	4940.13
00005	1302.631578947368421052631578	24750.00
00006	337.166666666666666666666666	2023.00
00008	185.666666666666666666666666	1114.00
00010	1500.000000000000000000000000	3000.00
00012	52.000000000000000000000000	52.00
00030	775.000000000000000000000000	1550.00
00100	850.000000000000000000000000	1700.00
00200	1094.500000000000000000000000	2189.00
10002	22.500000000000000000000000	22.50
10340	1500.000000000000000000000000	1500.00
44551	58.250000000000000000000000	349.50
44552	50.000000000000000000000000	50.00
66111	494.760000000000000000000000	2473.80
88000	750.000000000000000000000000	750.00

Der Ausgangspunkt

An den vielen Nachkommastellen des durchschnittlichen Umsatzes wollen wir uns für den Moment nicht stören. Dies kann zu einem späteren Zeitpunkt mit entsprechenden SQL-Funktionen einfach korrigiert werden. Vielmehr wollen wir uns die Frage stellen, wie es uns gelingt, zusätzlich den prozentualen Anteil dieses Artikels am Gesamtumsatz zu berechnen?

Auf alle Fälle benötigen wir für diese Aufgabenstellung den Gesamtumsatz:

```
SELECT  SUM(Artikelpreis * Menge) Gesamtumsatz
FROM Auftragsposition;
```

Nun müssen wir überlegen, wie wir diesen Wert in die ursprüngliche Spaltenliste einbinden. Eine Möglichkeit wäre ein Cross Join, den wir auch im Kapitel 7.6.3.1.3 verwenden und dort detailliert beschreiben. Hier wollen wir zur Lösung eine Unterabfrage verwenden, die eine temporäre Spalte aufbaut. Schauen Sie sich die folgende Abfrage an:

```
SELECT Artikelnummer,
AVG(Artikelpreis * Menge) as "Durchschnittsumsatz je
     Artikel",

SUM(artikelpreis * Menge) AS "Umsatz je Artikel",

(SELECT  SUM(Artikelpreis * Menge) FROM Auftragsposition)
     AS Gesamtumsatz

FROM Auftragsposition
GROUP BY Artikelnummer;
```

Als vierte Spaltendefinition enthält das Beispiel eine vollständige SQL-Abfrage. In der Unterabfrage wird der Gesamtwert aller Aufträge ermittelt. Zu beachten ist dabei, dass die Unterabfrage wieder in Klammern steht. Schauen wir uns das Result Set der Abfrage an:

ARTIKELNUMMER	Durchschnittsumsatz je Artikel	Umsatz je Artikel	GESAMTUMSATZ
00008	185.66666666666666666666666	1114.00	49342.23
88000	750.00000000000000000000000	750.00	49342.23
66111	494.76000000000000000000000	2473.80	49342.23
10340	1500.00000000000000000000000	1500.00	49342.23
00003	191.25000000000000000000000	1147.50	49342.23
00200	1094.50000000000000000000000	2189.00	49342.23
00010	1500.00000000000000000000000	3000.00	49342.23
00100	850.00000000000000000000000	1700.00	49342.23
00001	151.42857142857142857142857	1060.00	49342.23
00004	705.73285714285714285714285	4940.13	49342.23
00006	337.16666666666666666666666	2023.00	49342.23
44552	50.00000000000000000000000	50.00	49342.23
10002	22.50000000000000000000000	22.50	49342.23
00005	1302.63157894736842105263157	24750.00	49342.23
00002	111.80000000000000000000000	670.80	49342.23
00030	775.00000000000000000000000	1550.00	49342.23
00012	52.00000000000000000000000	52.00	49342.23
44551	58.25000000000000000000000	349.50	49342.23

Result Set mit temporärer Spalte – Unterabfrage

Mit dieser temporären Spalte können wir anschließend ebenso arbeiten wie mit den konstanten Spalten einer Tabelle. Um beispielsweise den prozentualen Anteil zu errechnen, müssen wir lediglich den Umsatz je Artikel mit 100 multiplizieren und anschließend durch den Gesamtumsatz dividieren. Starten wir einfach mal einen Versuch:

```
SELECT Artikelnummer,
AVG(Artikelpreis * Menge) as "Durchschnittsumsatz je Artikel",

SUM(artikelpreis * Menge)  as Umsatz_je_Artikel,

(SELECT  SUM(Artikelpreis * Menge) FROM Auftragsposition) AS Gesamtumsatz,

Umsatz_je_Artikel * 100 / Gesamtumsatz

FROM Auftragsposition

GROUP BY Artikelnummer
ORDER BY Artikelnummer;
```

```
Vendorencode: -206
Nachricht: [SQL0206] Spalte UMSATZ_JE_ARTIKEL nicht in angegebenen Tabellen. Ursache . . . . :
UMSATZ_JE_ARTIKEL ist keine Spalte von Tabelle *N in *N. Ist die Tabelle *N, ist UMSATZ_JE_ARTIKEL keine
Spalte einer Tabelle oder Sicht, auf die verwiesen werden kann. Fehlerbeseitigung:  Einen der folgenden Schritte
durchführen und die Anforderung wiederholen: -- Sicherstellen, dass die Spalten- und Tabellennamen in der
Anweisung korrekt angegeben wurden. -- Handelt es sich um eine Anweisung SELECT, sicherstellen, dass alle
erforderlichen Tabellen in der Klausel FROM angegeben wurden. -- Sollte die Spalte ein Korrelationsbezug sein, die
Spalte mit der korrekten Tabellenkennzeichnung qualifizieren.
```

Temporäre Spaltennamen der Unterabfrage können nicht verwendet werden

Die Grafik zeigt, dass ich eine Fehlermeldung erhalte, wenn ich versuche, die temporären Spalten mit ihrem Namen anzusprechen. Wir dürfen die Spaltennamen UMSATZ_JE_ARTIKEL und GESAMTUMSATZ leider in der Abfrage nicht direkt verwenden. Dies ist kein Zufall! Temporär aufgebaute Spalten eines Result Sets können innerhalb der gleichen Abfrage nicht mit ihrem temporären Namen angesprochen werden. Wir müssen daher anstelle des Spaltennamens wieder den entsprechenden SQL-Ausdruck verwenden:

7.6.5
Seite 10

```
SELECT Artikelnummer,
AVG(Artikelpreis * Menge) as "Durchschnittsumsatz je
        Artikel",

SUM(artikelpreis * Menge)   as Umsatz_je_Artikel,
(SELECT  SUM(Artikelpreis * Menge) FROM Auftragsposition)
        AS Gesamtumsatz,

SUM(artikelpreis * Menge)  * 100 / (SELECT
        SUM(Artikelpreis * Menge) FROM Auftragsposition)

as "Proz. Anteil"

FROM Auftragsposition
GROUP BY Artikelnummer
ORDER BY Artikelnummer;
```

Die Abfrage wird zwar fehlerfrei ausgeführt; dennoch ist das Ergebnis überraschend.

```
SELECT Artikelnummer,
AVG(Artikelpreis * Menge) as "Durchschnittsumsatz je Artikel",

SUM(artikelpreis * Menge)   as Umsatz_je_Artikel,  (SELECT  SUM(Artikelpreis * Menge) FROM Auftragsposition) AS Gesamtumsatz,

SUM(artikelpreis * Menge)  * 100 / (SELECT  SUM(Artikelpreis * Menge) FROM Auftragsposition) as "Proz. Anteil"

FROM Auftragsposition

GROUP BY Artikelnummer
ORDER BY Artikelnummer;
```

ARTIKELNUMMER	Durchschnittsumsatz je Artikel	UMSATZ_JE_ARTIKEL	GESAMTUMSATZ	Proz. Anteil
00001	151.428571428571428571428571	1060.00	49342.23	2
00002	111.800000000000000000000000	670.80	49342.23	1
00003	191.250000000000000000000000	1147.50	49342.23	2
00004	705.732857142857142857142857	4940.13	49342.23	10
00005	1302.631578947368421052631578	24750.00	49342.23	50
00006	337.166666666666666666666666	2023.00	49342.23	4

Das Ergebnis unserer Abfrage

Unsere Prozentwerte werden als ganze Zahlen dargestellt. Wir verlieren unsere Nachkommastellen bei der Berechnung. Hier helfen auch keine SQL-Funktionen, um das Problem zu lösen. Verantwortlich für das Ergebnis ist in diesem Fall der konstante Wert 100, der die Anzahl der Nachkommastellen in diesem Beispiel bestimmt. Diese Problematik müssen Sie allerdings nur beachten, wenn ausschließlich temporäre Spalten oder Ausdrücke in der Berechnung verwendet werden. Sobald Sie eine konstante Tabellenspalte verwenden oder mithilfe geeigneter SQL-Funktionen dem Ausdruck eine Länge mit entsprechender Nachkommastellendefinition zuweisen, tritt dieses Phänomen nicht auf.

SQL-Abfragen, die viele oder sehr lange Unterabfragen enthalten, können sehr schnell unübersichtlich sein. Daher schauen wir uns jetzt einen anderen Lösungsweg an: Wir können mit temporären Tabellen arbeiten. Kommen wir noch einmal auf unser Beispiel zurück. Schauen Sie sich zuerst die FROM-Klausel der folgenden Abfrage an:

```
SELECT  Artikelnummer,
AVG(Artikelpreis * Menge) AS "durchschnittlicher Umsatz
       je Artikel",
SUM(Artikelpreis * Menge) * 100.00 / Gesamtumsatz
       AS "prozentualer Anteil"

FROM Auftragsposition CROSS JOIN

       (SELECT  SUM(Artikelpreis * Menge)  AS Gesamtumsatz
             FROM Auftragsposition) as T
GROUP BY Artikelnummer, Gesamtumsatz
```

In der Beispielabfrage wird zunächst die temporäre Tabelle T aufgebaut. Der Name der temporären Tabelle ist für die weitere Abfrage irrelevant. Dennoch müssen Sie der temporären Tabelle zwingend einen Namen zuweisen, andernfalls erhalten Sie entsprechende Fehlermeldungen. Anschließend können Sie in der Hauptabfrage die Spalten der temporären Tabelle namentlich ansprechen. Nur eines müssen Sie bei Gruppierungsabfragen beachten. Die Spalten der temporären Tabelle müssen in der Gruppierung verwendet werden, wenn sie in der Spaltenliste der Hauptabfrage benutzt werden. Ich habe in diesem Beispiel damit kein Problem, da die Spalte GESAMTUMSATZ die Gruppierung inhaltlich nicht verändert.

7.6.5.3 Common Table Expressions (CTE)

Common Table Expressions sind Bestandteile des Select Statements und wurden bereits unter Release V4R4 eingeführt. Sie können am ehesten mit temporären Views, die nur für das aktuelle Select Statement gelten, beschrieben werden. CTEs werden gerne anstelle von ineinander geschachtelten Unterabfragen verwendet oder für komplexe Spaltendefinitionen.

Sie können vielleicht schon erahnen, dass SQL-Abfragen sehr komplex gestaltet werden und wie die vorangegangenen Beispiele gezeigt haben, ist ein ärgerlicher Aspekt, dass die gleichen Ausdrücke oftmals umständlich wiederholt werden müssen, manchmal sogar mehrmals. Unterabfragen in der FROM-Klausel können hier zwar ein klein wenig helfen, doch oftmals geschieht dies zu Lasten der Übersichtlichkeit. Nehmen wir nochmal das Beispiel aus dem vorangegangenen Kapitel:

```
SELECT Artikelnummer,
AVG(Artikelpreis * Menge) as "Durchschnittsumsatz je
         Artikel",

SUM(artikelpreis * Menge)   as Umsatz_je_Artikel,
(SELECT  SUM(Artikelpreis * Menge) FROM Auftragsposition)
         AS Gesamtumsatz,

SUM(artikelpreis * Menge)  * 100 / (SELECT
         SUM(Artikelpreis * Menge) FROM
         Auftragsposition) as "Proz. Anteil"

FROM Auftragsposition

GROUP BY Artikelnummer
ORDER BY Artikelnummer;
```

In diesem Beispiel sieht man, wie der gleiche Ausdruck mehrfach codiert wird, um den Gesamtumsatz zu ermitteln. Diese Art von Wiederholung ist langwierig, obgleich sie einfach zu kopieren und in den Code einzufügen ist. Es gibt jedoch eine Methode, die wiederholte Codierung solcher komplexen Ausdrücke zu umgehen. Eine Möglichkeit haben Sie bereits in Form der temporären Tabellen kennengelernt. Eine andere Möglichkeit sind so genannte Common Table Expressions (CTE).

CTEs sind einfach zu formulieren. Die Definition der CTEs erfolgt vor dem eigentlichen Select Statement, eingeleitet mit WITH. Als erstes benötigen wir also das Schlüsselwort WITH. WITH definiert einen SQL-Ausdruck, bevor die Hauptabfrage ausgeführt wird. Ich könnte das oben gezeigte Beispiel jetzt auch folgendermaßen schreiben:

```
WITH TabGesamt
AS
(SELECT  SUM(Artikelpreis * Menge) AS Gesamtumsatz FROM
             Auftragsposition)
```

Die Syntax ist nicht schwer. Nach dem Schlüsselwort WITH folgt stets der Name der CTE. Anschließend leitet das Schlüsselwort AS das SQL-Statement ein. Sie können hier wieder alle Klauseln und Ausdrücke verwenden, die wir bislang in einer SQL-Abfrage eingesetzt haben. Ich ermittle im Beispiel lediglich den Gesamtumsatz und nenne die entsprechende Spalte des Result Sets GESAMTUMSATZ.

Ohne weitere Schlüsselworte folgt anschließend die eigentliche Hauptabfrage:

```
SELECT Artikelnummer,
AVG(Artikelpreis * Menge) as "Durchschnittsumsatz je
         Artikel",
SUM(artikelpreis * Menge)   as Umsatz_je_Artikel,
         Gesamtumsatz,

SUM(artikelpreis * Menge)  * 100.00 / Gesamtumsatz
         as "Proz. Anteil"

FROM Auftragsposition CROSS JOIN TabGesamt

GROUP BY Artikelnummer, Gesamtumsatz
ORDER BY Artikelnummer;
```

Ergänzung 17/2011

In der Hauptabfrage verbinde ich die in der CTE aufgebaute temporäre Tabelle TabGesamt über einen Cross Join mit den Auftragspositionen und kann anschließend problemlos die Spalte Gesamtumsatz in der Abfrage verwenden, muss also nicht mehr die Unterabfrage mehrfach codieren.

Auch Modifikationen der Unterabfrage sind jetzt viel einfacher, da alle Änderungen zentral in der CTE erfolgen und die Hauptabfrage unverändert bleibt.

In einem Select-Statement können aber auch mehrere CTEs, durch Komma getrennt, definiert werden.

Schauen Sie sich die folgende Lösung an:

```
WITH
TabGesamt AS
(SELECT  SUM(Artikelpreis * Menge) as Gesamtumsatz FROM
 Auftragsposition) ,

Umsatz AS
(Select  artikelnummer, SUM(artikelpreis * Menge)
 as Umsatz_je_Artikel,
AVG(Artikelpreis * Menge) as Schnitt_je_artikel
 from auftragsposition
 group by artikelnummer)

-- Es folgt die Hauptabfrage

SELECT  DISTINCT Artikelnummer, Gesamtumsatz ,
Schnitt_je_Artikel AS "durchschnittlicher Umsatz",
Umsatz_je_Artikel,  Umsatz_je_Artikel * 100.00 /
 Gesamtumsatz AS "in Prozent"

FROM Auftragsposition CROSS JOIN TabGesamt JOIN Umsatz
USING(Artikelnummer)
ORDER BY Artikelnummer;
```

Ich habe im oberen Beispiel eine weitere CTE hinzugefügt. Die Berechnung des durchschnittlichen Umsatzes und die Umsatzberechnung der einzelnen Artikel habe ich in die CTE UMSATZ verlagert. Syntaktisch werden die erste und zweite CTE lediglich durch ein Komma getrennt. In der Hauptabfrage verknüpfe ich anschließend über einen INNER JOIN die CTE UMSATZ ebenfalls mit der Tabelle AUFTRAGSPOSITION. Da ich jetzt auf die Gruppierung in der Hauptabfrage verzichten möchte, schließe ich einfach die mehrfach vorkommenden Zeilen mit Hilfe des Schlüsselwortes DISTINCT aus.

Aufgrund des zusätzlichen Codes, den die WITH-Klausel erfordert, spart man in diesem relativ einfachen Beispiel mit der Vorgehensweise nicht sehr viel Schreibarbeit. Der tatsächliche Code des SELECT-Statements wird dadurch aber einfacher. In alltäglichen Anwendungen benötigt man allerdings oft SQL-Statements, die weitaus komplexer sein müssen als dieses Statement es ist. Ist dies der Fall, dann können Sie durch den Einsatz von CTEs Ihre Statements so weit wie möglich vereinfachen.

7.6.5.4 Korrelierende Unterabfragen

Bislang waren die Unterabfragen immer unabhängig von der übergeordneten Abfrage. Man konnte sie eigenständig ausführen, ihr Ergebnis in eine neue Tabelle eintragen und die übergeordnete Abfrage auf diese Tabelle anwenden.

Beim Einsatz einer Unterabfrage können die Werte der Unterabfragen aber auch mit den Werten der äußeren Hauptabfrage korrelieren. Wikipedia definiert den Begriff der Korrelation folgendermaßen:

„Die Korrelation ist eine Beziehung zwischen zwei oder mehr statistischen Variablen. Wenn sie besteht, ist noch nicht gesagt, ob eine Größe die andere kausal beeinflusst, ob beide von einer dritten Größe kausal abhängen oder ob sich überhaupt ein Kausalzusammenhang folgern lässt."

Korrelierende Unterabfragen sind demnach mehrere Unterabfragen, die miteinander in Beziehung stehen können, sich aber nicht beeinflussen müssen. Wenn bisher das innere Subselect vollständig aufgelöst wurde, um anschließend die Hauptabfrage aufzulösen, so gibt es jetzt eine Verbindung zwischen den Abfragen. Die Unterabfrage benötigt dabei immer wieder Werte aus der Hauptabfrage und kann nur sukzessive aufgelöst werden, sozusagen im Wechsel mit der übergeordneten Abfrage.

Betrachten wir auch hier ein Beispiel, das diesen Sachverhalt verdeutlicht: Es soll ermittelt werden, welches der teuerste Artikel je Produktkategorie ist. Das Ergebnis der Abfrage soll folgendes Result Set liefern:

WARENGRUPPE	ARTIKELNUMMER	BEZEICHNUNG	ARTIKELPREIS
0001	00005	Kugelschreiber (10)	15.00
0002	00030	Papier 500 Blatt	6.50
0003	66111	Locher	3.99
0004	10100	Schuhschrank	25.00
1540	00200	Schreibtisch Eck	250.00

Das Ergebnis der korrelierenden Unterabfrage

Eine einfache Gruppierungsabfrage ist nicht möglich, wenn wir die Artikelnummer und die Artikelbezeichnung im Result Set erwarten. Aber auch eine einfache Unterabfrage in der bisher gezeigten Form führt nicht zum gewünschten Ergebnis, da wir mit den bisher erlernten Techniken lediglich den teuersten Artikel im gesamten Sortiment ermitteln können. Das Interessante an dieser Aufgabenstellung ist, dass wir einen Gruppenwechsel codieren wollen. Die Unterabfrage darf nicht mehr vollständig aufgelöst werden, sondern muss für jede vorhandene Warengruppe ausgeführt werden. Die Lösung des Problems sieht folgendermaßen aus:

```
SELECT Warengruppe, Artikelnummer, Bezeichnung,  Artikelpreis
FROM Artikel  A    WHERE Artikelpreis   =
                (SELECT MAX(Artikelpreis)
                FROM Artikel B
                WHERE  B.Warengruppe = A.Warengruppe
                GROUP BY Warengruppe
                )
ORDER BY Warengruppe;
```

Bei dieser Abfrage wird die äußere Abfrage automatisch, wie bei einer einfachen mit GROUP BY gruppierten Abfrage, zunächst nach der Warengruppe gruppiert. Erst dann wird für jede Gruppe einmal die Unterabfrage aufgerufen. Und nur auf die Teilmenge der äußeren Abfrage wird das zusätzliche, durch die Unterabfrage bestimmte Kriterium angewandt. Erzwungen wird dieses Verhalten durch die Bedingung der Unterabfrage. In der WHERE-Klausel der Unterabfrage wird eine Spalte – nämlich die Warengruppe der Tabelle A – eingebunden. Die Unterabfrage kann also gar nicht aufgelöst werden, ohne dass die jeweilige Warengruppe der äußeren Abfrage herangezogen wird.

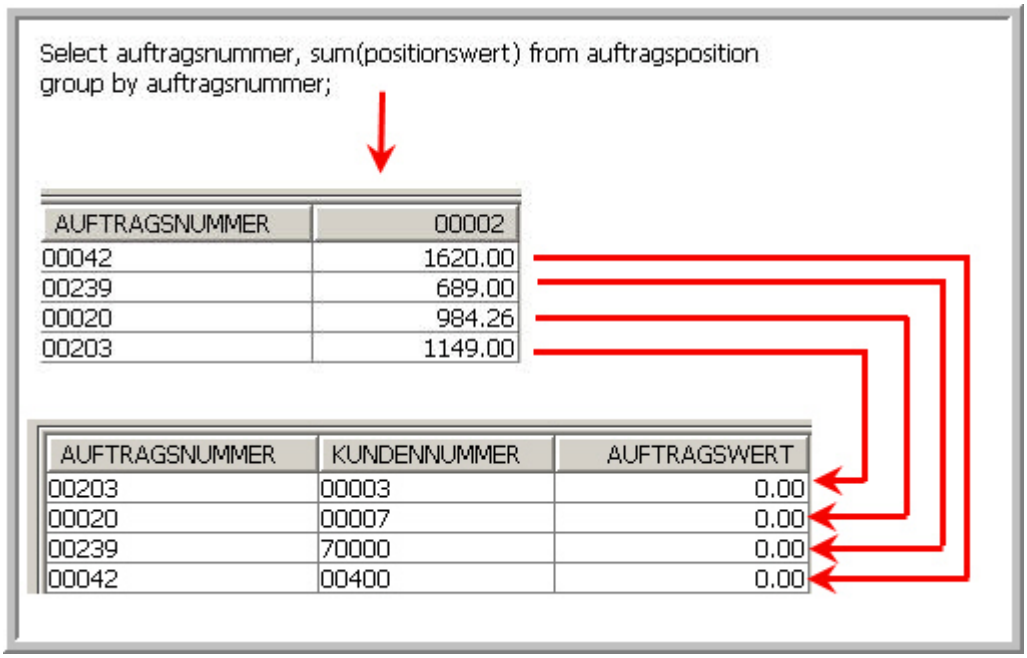
Diese Art von Unterabfragen ist insbesondere bei der Ausführung einer SQL UPDATE-Anweisung sinnvoll, in denen nicht nur ein Vergleichskriterium abgefragt werden soll, sondern Mehrfachvorkommen innerhalb einer weiteren Datei verarbeitet werden müssen.

Stellen Sie sich vor, dass im Kopfsatz eines Auftrags der Auftragsgesamtwert gespeichert wird:

AUFTRAGSNUMMER	KUNDENNUMMER	AUFTRAGSDATUM	AUSLIEFERUNGSDATUM	AUFTRAGSWERT
00201	00001	2010-03-10	0001-01-01	727.00
00203	00003	2010-03-12	2010-04-15	1672.50
00204	00007	2010-05-03	2010-05-12	1750.00
00205	00008	2010-03-14	2010-05-21	1400.00
00207	70000	2010-03-21	2010-10-30	750.00
00208	00045	2010-03-22	2010-05-01	1770.00

Die Auftragskopftabelle

Um diesen Wert zu ermitteln, müssen wir die Auftragssummen anhand der Positionszeilen ermitteln und diese Werte anschließend für die Aktualisierung der Auftragskopfdaten nutzen. Folgende Grafik verdeutlicht diesen Sachverhalt:



Schematische Darstellung der Aufgabe

Die in der Grafik gezeigte Abfrage ist schnell geschrieben; sie dient als Unterabfrage, weil diese Abfrage die Werte für die Aktualisierung zur Verfügung stellt. Die übergeordnete Abfrage muss daher die Aktualisierung übernehmen:

```
UPDATE Auftrag SET Auftragswert  =
                (
                      SELECT SUM(Positionswert)
                      FROM Auftragsposition

                      GROUP BY Auftragsnummer
                 );
```

7.6.5
Seite 18

Abschließend müssen wir die Abfrage noch ein klein wenig modifizieren, denn schließlich sollen nur Updates durchgeführt werden, wenn die Auftragsnummern in beiden Tabellen identisch sind:

```
UPDATE Auftrag   SET Auftragswert  =
                 (
                 SELECT SUM(Positionswert)
                 FROM Auftragsposition
                 WHERE Auftrag.Auftragsnummer =
                   Auftragsposition.Auftragsnummer
                 GROUP BY Auftragsnummer
                 );
```

Jetzt erzielen wir wieder den gewünschten Effekt. Für jede Gruppe – also für jede Auftragsnummer – wird die Unterabfrage einmal aufgerufen. Und nur auf die Teilmenge der äußeren Abfrage wird das Update ausgeführt.

Schauen wir uns ein weiteres Beispiel in diesem Zusammenhang an:

ITPSQL.AUFTRAGSPOSITION - 172.16.110.22(Epsserver)

Positionsmenge * Artikelpreis

Datei Bearbeiten Ansicht Zeilen Hilfe

AUFTRAGSNUMMER	ARTIKELNUMMER	POSITIONSMENGE	POSITIONSWERT	ARTIKELPREIS
00201	00001	30	60.00	2.00
00205	00002	134	160.80	1.20
00201	00012	10	52.00	5.20
00009	00001	10	20.00	2.00
00207	00003	112	168.00	1.50
00203	00001	87	0	
00204	00004	89	613.21	6.89

Aktualisierung der Positionsmenge

In der Positionstabelle soll der Positionswert aktualisiert werden. Die notwendige UPDATE-Anweisung ist der vorangegangenen Abfrage sehr ähnlich:

```
UPDATE Auftragsposition A SET Positionswert =
             (
                SELECT (Positionsmenge * Artikelpreis)
                FROM Auftragsposition B
                WHERE  A.Auftragsnummer  = B.Auftragsnummer
                AND    A.Artikelnummer   = B.Artikelnummer
             );
```

Wir benötigen lediglich für die Qualifizierung unserer Tabellen Synonyme, da die Tabellennamen in der Haupt- und Unterabfrage identisch sind. Allerdings bekommen wir folgende Fehlermeldung, wenn wir die Abfrage ausführen:

```
SQL-Status: 23502
Vendorencode: -407
Nachricht: [SQL0407] Nullwerte für Spalte oder Variable WERT nicht zulässig. Ursache . . . . : Einer
der folgenden Fehler ist aufgetreten: Spalte WERT ist eine Zielspalte in einer Anweisung UPDATE
oder INSERT für Tabelle AUFTR00001 in ITPSQL. Es wurde ein Nullwert festgelegt, um in dieser
Spalte eingefügt oder aktualisiert zu werden, oder es wurde kein Wert für die Spalte in einer
Anweisung INSERT angegeben und für die Spalte sind keine Nullwerte zulässig. Der Nullwert wurde
mit der relativen Eintragsnummer  1 in der Liste VALUES, SELECT oder in der SET-Klausel
angegeben. -- Spalte WERT ist eine Zielspalte in einer Anweisung ALTER für Tabelle AUFTR00001 in
ITPSQL. Das Attribut der Spalte WERT darf nicht in NOT NULL geändert werden, da bereits ein
```

Nullwerte für Spalte WERT nicht zulässig

Was ist passiert? Im Auftrag 00203 wurde der Artikel 00001 ohne Verkaufspreis gespeichert. Die Spalte Verkaufspreis enthält hier aber nicht den Wert 0, sondern blieb einfach leer und ist daher NULL. Folglich ist das Produkt aus Verkaufspreis * Menge ebenfalls NULL. Die Spalte WERT darf aber laut Definition keine Nullwerte enthalten, so dass die UPDATE-Anweisung daraufhin fehlschlägt. Die Fehlermeldung wird also nicht durch unseren Subselect verursacht! Wenn wir die Nullwerte abfangen wollen, gelingt das mit der SQL-Funktion IFNULL:

```
IFNULL(Ausdruck oder Spalte , Ersatzwert)
```

Die SQL-Funktion erwartet zwei Argumente. Überwacht wird das erste Argument, ist dieses Argument NULL, wird stattdessen der Ersatzwert verwendet.

```
UPDATE Auftragsposition A SET Positionswert =
       (
       SELECT  IFNULL( (Positionsmenge * Artikelpreis) , 0)
       FROM   Auftragsposition B
       WHERE  A.Auftragsnummer  = B.Auftragsnummer
       AND    A.Artikelnummer = B.Artikelnummer
       );
```

Wenn wir unsere Unterabfrage in dieser Art ergänzen, wird das UPDATE problemlos ausgeführt.

Diese Korrelationen sind im Übrigen auf allen Ebenen möglich. Stellen Sie sich vor, Sie benötigen in einer Abfrage neben dem Namen der Warengruppe auch den Namen der zugehörigen Hauptwarengruppe. Dann ist eine Möglichkeit der Self Join, wie wir ihn im Kapitel 7.6.3.5 bereits kennengelernt haben. Eine andere Möglichkeit haben wir mit einer korrelierenden Unterabfrage auf Spaltenebene. Diese SQL-Lösung sieht folgendermaßen aus:

```
SELECT
( SELECT  WgBezeichnung  FROM  Warengruppe B
  WHERE  A.Hauptwarengruppe = B.Warengruppe )
  as Hauptwarengruppe,
WGBezeichnung , Artikelnummer,  Bezeichnung
FROM  Warengruppe A JOIN Artikel USING(Warengruppe)
ORDER BY Hauptwarengruppe;
```

Wie in den zuvor gezeigten Lösungen steht die Unterabfrage als Spaltendefinition. Innerhalb der Spaltendefinition wird der Bezug zur äußeren Abfrage hergestellt, indem als Bedingung formuliert wird, dass die Hauptwarengruppennummer der Tabelle A identisch sein soll mit der Warengruppennummer der Tabelle B. Als Ergebnis erhalten wir das folgende Result Set:

HAUPTWARENGRUPPE	WGBEZEICHNUNG	ARTIKELNUMMER	BEZEICHNUNG
Büromaterial	Stifte	00001	Mauspad
Büromaterial	Stifte	00003	Textmarker Rot (10)
Büromaterial	Stifte	00004	Füllfederhalter
Büromaterial	Stifte	00005	Kugelschreiber (10)
Büromaterial	Stifte	88000	Mauspad ergo

Das Ergebnis der korrelierenden Unterabfrage auf Spaltenebene

7.6.5.5 Die Prädikate EXISTS, ALL, ANY oder SOME

Bislang war es stets so, dass die Hauptabfrage die Datenmenge der Unterabfrage zur Prüfung der Bedingung genutzt hat. Mit dem Prädikat EXISTS dagegen prüfen Sie lediglich, ob Daten in der Unterabfrage gefunden werden. EXISTS liefert den logischen Wert WAHR(TRUE) zurück, wenn der SELECT-Befehl der Unterabfrage Daten zurückliefert. Wenn keine Daten ausgewählt werden, dann ist das Ergebnis der logische Wert FALSCH(FALSE).

Schauen wir uns hierzu ein Beispiel an: Wir wollen wissen, für welche Artikel keine Bestellungen vorliegen. Wir könnten diese Aufgabe mit einer einfachen Unterabfrage und dem Prädikat NOT IN lösen:

```
SELECT Artikelnummer,Bezeichnung
FROM Artikel
WHERE Artikelnummer  NOT IN
            (  SELECT Artikelnummer
               FROM   Auftragsposition
            );
```

Ein identisches Ergebnis bekommen wir, wenn wir die Abfrage mit dem Prädikat EXISTS umsetzen.

Hierfür müssen wir als erstes die Bedingung ein wenig verändern:

```
SELECT Artikelnummer, Bezeichnung
FROM Artikel
WHERE NOT EXISTS
```

Wir verzichten auf eine Vergleichsbedingung und verwenden lediglich das Prädikat EXISTS. Anschließend folgt die Unterabfrage.

```
          (  SELECT Auftragsnummer
             FROM Auftragsposition
          );
```

Wenn Sie diese SQL-Abfrage ausführen, werden Sie feststellen, dass Ihnen alle Artikelstammdaten angezeigt werden, denn die Hauptabfrage wird ausgeführt, wenn die Unterabfrage mindestens einen Satz enthält. Wenn wir davon ausgehen, dass mindestens ein Auftrag vorliegt, wird die Unterabfrage den Wert TRUE liefern und so dafür sorgen, dass die Hauptabfrage ausgeführt wird. Um aber nur Artikel anzuzeigen, für die keine Bestellungen vorliegen, müssen wir die Unterabfrage umformulieren, der Einsatz des Prädikats EXISTS ist in der Regel nur gemeinsam mit korrelierenden Unterabfragen sinnvoll.

```
       (SELECT Auftragsnummer
       FROM Auftragsposition
        WHERE Auftragsposition.Artikelnummer =
           Artikel-Artikelnummer
       );
```

Wir wissen, dass jetzt die Unterabfrage für jede Zeile der Tabelle ARTIKEL ausgeführt wird. Allerdings wird die Unterabfrage bereits abgebrochen, wenn der erste Eintrag gefunden wird; es müssen schließlich nicht alle Einträge selektiert werden, um zu entscheiden, ob die Abfrage WAHR oder FALSCH ist. Häufig wird daher in Unterabfragen mit dem Prädikat EXISTS ganz darauf verzichtet, Daten der Tabelle in den Arbeitsspeicher zu verlagern, indem einfach Konstanten in der Spaltenliste verwendet werden:

```
SELECT Artikelnummer, Bezeichnung
FROM Artikel
WHERE NOT EXISTS
         (  SELECT 1
            FROM Auftragsposition
            WHERE Auftragsposition.Artikelnummer =
              Artikel.Artikelnummer
         );
```

Wenn Sie sich jetzt fragen, wann Sie in den Abfragen das Prädikat IN und wann Sie das Prädikat EXISTS verwenden sollten, kann ich Ihnen diese Frage nicht eindeutig beantworten; schließlich führen beide Abfragen letztendlich zum gleichen Ergebnis.

Unterschiede könnten Sie ggf. in der Ausführungsgeschwindigkeit feststellen. Wenn die Unterabfrage einer IN-Abfrage sehr viele Daten selektieren muss, dauert natürlich die Ausführung der Unterabfrage bereits eine gewisse Zeit. Außerdem wird für die Daten entsprechend Arbeitsspeicher benötigt. Die Unterabfrage der EXISTS-Abfrage wird in der Regel viel schneller ausgeführt, da lediglich ein Eintrag benötigt wird, um das Ergebnis FALSCH oder WAHR bestimmen zu können.

Allerdings müssen Sie auch berücksichtigen, dass bei der EXISTS-Abfrage die Unterabfrage für jeden Eintrag der übergeordneten SELECT-Anweisung ausgeführt wird. Hat meine Artikeltabelle beispielsweise 10.000 Einträge, wird die Unterabfrage 10.000 Mal durchlaufen. Die Unterabfrage der IN-Abfrage muss dagegen immer nur einmal ausgeführt werden.

Welches Prädikat die bessere Wahl ist, ist daher stets von Ihrer Datenlage und den vorhandenen Indizes abhängig. Im Zweifelsfall probieren Sie es einfach aus, denn wie so oft, haben Sie die Wahl und können identische Lösungen auf unterschiedliche Arten codieren.

Die Prädikate ANY, SOME und ALL arbeiten wieder ähnlich wie das Prädikat IN. Die Unterabfrage wird vollständig ausgeführt, um darauf anschließend die Bedingung zu prüfen. Als Vergleichsoperatoren für diese Prädikate sind die Vergleichsoperatoren > , >=, <, <= ebenso wie = und != erlaubt.

Schauen wir uns zunächst das Prädikat ALL an. Wenn Sie ALL einsetzen, benötigen Sie eine Unterabfrage, die zunächst vollständig ausgeführt wird. Anschließend wird die Bedingung geprüft. Die Bedingung der äußeren Abfrage ist wahr, wenn der Vergleichswert die formulierte Bedingung zu 100 % erfüllt. Falls die Unterabfrage keine Werte enthält, ist die Bedingung übrigens immer erfüllt.

In der folgenden Abfrage suche ich den höchsten Artikelpreis.

```
SELECT Bezeichnung, Artikelpreis
FROM Artikel
WHERE Artikelpreis >= ALL ( SELECT  Artikelpreis
   FROM Artikel );
```

Nur der höchste Artikelpreis erfüllt die in der Abfrage formulierte Bedingung; er ist größer als alle anderen Artikelpreise. Natürlich haben Sie recht, wenn Sie behaupten, dass Sie dieses Ergebnis auch mit der folgenden Abfrage erhalten:

```
SELECT   Bezeichnung, Artikelpreis
FROM     Artikel
WHERE    Artikelpreis  >=    ( SELECT MAX(Artikelpreis)
            FROM     Artikel );
```

Ergänzung 17/2011

7.6.5
Seite 24

Sie können auswählen, für welche Syntax Sie sich entscheiden; das Ergebnis wird stets identisch sein.

Wenn Sie hingegen anstelle des Prädikats ALL die Prädikate ANY und SOME verwenden, ist die Bedingung bereits WAHR, wenn mindestens ein Wert der Unterabfrage die Bedingung erfüllt.

Schauen wir uns jetzt abschließend noch eine Abfrage an, in der das Prädikat ANY eingesetzt werden kann. Die Abfrage selektiert alle Kunden, die nicht im Postleitzahlengebiet ‚2' wohnen und denen ein höherer Rabatt eingeräumt wurde als er in der Postleitzahlregion „2" üblich ist.

```
SELECT Kunden.Kundennummer, Kundenname, Auftragsnummer,
   Rabatt
FROM Auftrag JOIN  Kunden USING(Kundennummer)
WHERE  Kdplz  NOT BETWEEN  '20000' AND '29999'
AND  Rabatt > ANY
   (
     SELECT  Rabatt FROM  Auftrag JOIN Kunden
        USING(Kundennummer)
     WHERE  Kdplz  BETWEEN  '20000' AND '29999'
   );
```

In unserem Beispiel wird zunächst eine Liste mit den in der Postleitzahlenregion ‚2' gewährten Rabatten erstellt. Anschließend werden alle anderen Postleitzahlengebiete geprüft, um festzustellen, ob irgendein Rabattwert in diesen Regionen höher ist als irgendein Rabatt der Region ‚2'. Die Aussage könnte interessant sein, wenn wir eigentlich aufgrund einer Aktion nur die Umsätze der Region ‚2' in besonderem Maße forcieren wollen.

7.6.6 SQL-Funktionen im praktischen Einsatz

In Kapitel 7.6.1.3.2 finden Sie eine Übersicht der skalaren SQL-Funktionen. Gerade diese Funktionen sind es, die die Stärke von SQL ausmachen. Funktionen können wir als eine Art Befehl ansehen. Mit der Funktion MIN(*spaltenname*) befehlen wir dem System, uns den kleinsten Wert einer Spalte anzuzeigen. In diesem Fall wird das Ergebnis der Funktion durch seine Eingabeparameter bestimmt. Aber viele Funktionen verwenden auch gar keine Eingabeparameter, wie etwa die Funktion CURDATE(), die das aktuelle Systemdatum liefert. Welche SQL-Funktionen vorhanden sein müssen, legt der SQL-Standard fest. Darüber hinaus hat IBM eine Reihe weiterer Funktionen implementiert, die sich nicht im SQL-Standard wiederfinden. Nun wollen wir es aber nicht beim reinen Funktionsüberblick belassen, sondern uns in diesem Kapitel einige praktische Beispiele anschauen.

Ergänzung 18./2011

7.6.6
Seite 2

7.6.6.1 Mit Datum und Uhrzeit arbeiten

Wir leben in einer Gesellschaft, die von Terminen geprägt ist, daher ist es vorteilhaft, dass SQL über viele Funktionen verfügt, die sich mit Datum und Uhrzeit befassen. SQL-Fragen zu Datumsberechnungen findet man in Newsgroups fast täglich. Dieses und die folgenden Kapitel sollen helfen, häufige Fragestellungen und Probleme schnell zu lösen.

Die Basistabelle für die folgenden Beispiele ist immer die Tabelle AUFTRAG.

Inhalt von ITPSQL.AUFTRAG - 172.16.110.22(Epsserver)

	AUFTRAGSNUMMER	KUNDENNUMMER	AUFTRAGSDATUM	AUSLIEFERUNGSDATUM	AUFTRAGSWERT	VERKAEUFER	RABATT
1	00201	00001	2010-03-10	0001-01-01	112.00	10	3.20
2	00203	00003	2010-03-12	2010-04-15	1149.00	270	1.00
3	00204	00007	2010-05-03	2010-05-12	1453.21	330	3.20
4	00205	00008	2010-03-14	2010-05-21	160.80	10	3.20
5	00207	70000	2010-03-21	2010-10-30	498.50	200	3.20
6	00208	00045	2010-03-22	2010-05-01	3651.26	50	3.20
7	00009	55555	2009-03-10	2009-04-10	20.00	300	3.20
8	00210	00400	2010-11-03	0001-01-01	568.00		3.20
9	00211	00101	2010-03-12	2010-05-12	2250.00	10	3.20

Die Tabelle AUFTRAG

Die Spalten AUFTRAGSDATUM und AUSLIEFERUNGSDATUM der Tabelle sind mit dem Datentyp DATE deklariert. Da wir in DATE-Spalten nicht die Werte 0 oder LEERZEICHEN speichern können, erhalten alle Aufträge, die noch nicht ausgeliefert wurden, das Auslieferungsdatum 01.01.0001.

In einem ersten Schritt wollen wir das geplante Auslieferungsdatum berechnen. Gehen wir davon aus, dass zwischen dem Auftragseingang und der Auslieferung in der Regel zehn Tage liegen, dann müssen wir zehn Tage als Zeitintervall zu unserem Auftragsdatum addieren, um den Soll-Liefertermin zu ermitteln. Wir formulieren hierfür einfach einen entsprechenden Ausdruck, und über die Schlüsselwörter DAYS, MONTH oder YEAR spezifizieren wir das zu addierende Zeitintervall.

```
SELECT Auftragsnummer,
Kundennummer,
Auftragsdatum ,
Auftragsdatum + 10  DAYS AS "geplante Lieferung"
FROM Auftrag
WHERE Auslieferungsdatum = '01.01.0001'
ORDER BY Auftragsdatum DESC;
```

Und so sieht anschließend das Ergebis aus:

AUFTRAGSNUMMER	KUNDENNUMMER	AUFTRAGSDATUM	geplante Lieferung
00208	00045	2011-06-22	2011-07-02
00248	00201	2011-06-21	2011-07-01
00047	00101	2011-06-17	2011-06-27
00246	00103	2011-06-15	2011-06-25
00045	00102	2011-06-14	2011-06-24
00207	70000	2011-04-21	2011-05-01
00224	00045	2011-04-14	2011-04-24

Zeitintervall addieren

Ähnlich gehen wir natürlich vor, wenn wir Werte subtrahieren wollen.

7.6.6.2 DAYOFWEEK, DAYNAME und Fallunterscheidungen

Berücksichtigen müssen wir jetzt noch, dass wir am Samstag und am Sonntag keine Waren ausliefern. Leider gibt es bislang noch nicht die Standardfunktion WORKDAYS. Wir müssen dieses Problem daher „zu Fuß“ angehen.

Als Erstes muss der geplante Wochentag ermittelt werden. Um herauszufinden, auf welchen Wochentag die Lieferung fällt, stehen uns gleich drei mögliche Funktionen zur Verfügung: DAYNAME, DAYOFWEEK und DAYOFWEEK_ISO.

Erweitern wir also unsere Abfrage und verwenden dabei die genannten Funktionen:

```
SELECT Auftragsnummer,
Kundennummer,
Auftragsdatum ,
Auftragsdatum + 10  DAYS  AS "geplante Lieferung" ,

DAYNAME(Auftragsdatum + 10  DAYS)         AS "DayName",
DAYOFWEEK(Auftragsdatum + 10  DAYS)       AS "DayOfWeek" ,
DAYOFWEEK_ISO(Auftragsdatum + 10  DAYS)   AS "DayOf-Week_ISO"
FROM Auftrag
WHERE Auslieferungsdatum = '01.01.0001'
ORDER BY Auftragsdatum DESC;
```

geplante Lieferung	Dayname	DayOfWeek	DayOfWeek_ISO
2011-07-02	Samstag	7	6
2011-07-01	Freitag	6	5
2011-06-27	Montag	2	1
2011-06-25	Samstag	7	6
2011-06-24	Freitag	6	5
2011-05-01	Sonntag	1	7
2011-04-24	Sonntag	1	7
2011-04-22	Freitag	6	5
2011-04-20	Mittwoch	4	3
2011-04-20	Mittwoch	4	3

Den Wochentag ermitteln

Die Funktion DAYNAME ist selbsterklärend. Die Funktionen DAYOFWEEK und DAYOFWEEK_ISO liefern anstelle des Wochentagnamens eine Ziffer, wobei die Funktion DAYOFWEEK den Wert 1 liefert, wenn es sich um einen Sonntag handelt. Bei der Funktion DAYOFWEEK_ISO hingegen beginnt die Woche stets mit dem Montag, so dass in diesem Fall der Sonntag durch den Wert 7 repräsentiert wird.

Zurück zu unserem Problem: Fällt also der geplante Liefertermin auf einen Samstag, sind zwölf Tage zum Auftragsdatum hinzuzuaddieren, bei einem Sonntag müssen hingegen nur elf Tage addiert werden, in den übrigen Fällen wie bisher zehn Tage.

In der Praxis trifft man häufig auf Situationen, bei denen der Einsatz von SQL erwünscht ist, aber die Zahl der zu treffenden Fallunterscheidungen – wie in diesem Fall – dessen Einsatz erschwert. Hier hilft die Funktion CASE. Mit der CASE-Funktion können Sie für derartige Situationen differenzierte SQL-Statements generieren. Sie können problemlos Mehrfachverzweigungen innerhalb einer SELECT- oder UPDATE-Anweisung realisieren.

Die Case-Funktion verwenden wir für unser Problem auf Spaltenebene:

```
SELECT Auftragsnummer,
Kundennummer,
Auftragsdatum ,

CASE DAYOFWEEK_ISO(Auftragsdatum + 10  DAYS)
WHEN 6 THEN Auftragsdatum + 12  DAYS
WHEN 7 THEN Auftragsdatum +11 DAYS
ELSE Auftragsdatum + 10 DAYS
END AS "geplante Lieferung", ….
```

Eingeleitet wird die Fallunterscheidung durch den Funktionsnamen CASE. Es folgt die Spalte oder ein komplexer Ausdruck der ausgewertet werden soll. Ich verwende in unserem Beispiel den Ausdruck:

```
DAYOFWEEK_ISO(Auftragsdatum + 10 DAYS)
```

Nun folgen die diversen Fallunterscheidungen, die stets mit dem Schlüsselwort WHEN eingeleitet werden; Das Schlüsselwort THEN kennzeichnet anschließend den Wert, der verwendet werden soll, wenn die Bedingung zutrifft. Falls keine der formulierten Bedingungen wahr ist, liefert die CASE-Funktion den Wert des entsprechenden ELSE-Statements. Sie können einen ELSE-Zweig formulieren, müssen es aber nicht. Fehlt die ELSE-Bedingung, liefert das Statement für Spaltenwerte, für die keine der WHEN-Bedingungen zutrifft, den Wert NULL. Beachten Sie, dass das Wort CASE den Anfang der Funktion kennzeichnet und END die Funktion beendet. Abgeschlossen wird die CASE-Funktion somit immer durch das Schlüsselwort END.

Schauen wir uns das Ergebnis einmal an:

AUFTRAGSNUMMER	KUNDENNUMMER	AUFTRAGSDATUM	geplante Lieferung	geplanter Tag
00247	00101	2011-06-19	2011-06-29	Mittwoch
00247	00101	2011-06-17	2011-06-27	Montag
00245	00102	2011-06-14	2011-06-24	Freitag
00245	00102	2011-06-14	2011-06-24	Freitag
00247	00101	2011-06-11	2011-06-21	Dienstag
00347	00101	2011-06-11	2011-06-21	Dienstag
00247	00101	2011-06-09	2011-06-20	Montag
00245	00102	2011-06-08	2011-06-20	Montag
00247	00101	2011-06-05	2011-06-15	Mittwoch

Ergebnis mit Fallunterscheidung

Wenn Sie sich die Abbildung anschauen, werden Sie schnell feststellen, dass ich mein SQL-Statement noch ein wenig erweitert habe, damit mir wieder der Wochentagname angezeigt wird. Zu diesem Zweck habe ich eine zusätzliche Spalte mit der CASE-Funktion und der Funktion DAYNAME integriert. Auch das ist möglich: Sie können CASE-Funktionen verschachteln oder das Ergebnis der CASE-Funktion in eine weitere SQL-Funktion einbetten:

```
.....
DAYNAME
(
CASE
DAYOFWEEK_ISO(Auftragsdatum + 10  DAYS)
WHEN 6 THEN  Auftragsdatum + 12  DAYS
WHEN 7 THEN Auftragsdatum +11 DAYS
ELSE Auftragsdatum + 10 DAYS
END

)  AS  "geplanter Tag"

FROM Auftrag
WHERE Auslieferungsdatum = '01.01.0001'
ORDER BY  Auftragsdatum DESC;
```

Die CASE-Funktion kennt im Übrigen zwei unterschiedliche Formate. Sie können den auszuwertenden Ausdruck direkt hinter die Anweisung CASE schreiben, wie in unserem Beispiel, oder Sie formulieren den gesamten Ausdruck im WHEN-Zweig der Funktion:

```
SELECT Auftragsnummer,
Kundennummer,
Auftragsdatum ,

CASE
WHEN DAYOFWEEK_ISO(Auftragsdatum + 10  DAYS) = 6
THEN Auftragsdatum + 12  DAYS

WHEN DAYOFWEEK_ISO(Auftragsdatum + 10  DAYS) = 7
THEN Auftragsdatum +11 DAYS

ELSE Auftragsdatum + 10 DAYS
END AS "geplante Lieferung", ….
```

Das zweite Format ist erforderlich, wenn Sie beispielsweise die Vergleichsoperatoren größer (>), kleiner (<) oder ungleich (<>) verwenden wollen oder wenn Sie unterschiedliche Ausdrücke auswerten müssen.

In einem nächsten Schritt will ich die Spalte SOLL-Lieferdatum mit den soeben ermittelten Werten aktualisieren. Natürlich könnte ich drei eigenständige UPDATE-Anweisungen formulieren:

```
UPDATE Auftrag SET Soll_Auslieferungsdatum = Auftragsdatum
   + 12  DAYS
WHERE DAYOFWEEK_ISO(Auftragsdatum + 10  days) = 6
AND Soll_Auslieferungsdatum = '01.01.0001';
```

```
UPDATE Auftrag SET Soll_Auslieferungsdatum = Auftragsdatum
   + 11  DAYS
WHERE DAYOFWEEK_ISO(auftragsdatum + 10  days) = 7
AND Soll_Auslieferungsdatum = '01.01.0001';
```

```
UPDATE Auftrag SET Soll_auslieferungsdatum = Auftragsdatum
   + 12  DAYS
WHERE Soll_Auslieferungsdatum = '01.01.0001';
```

7.6.6
Seite 8

Einfacher geht es auch in diesem Fall mit der Funktion CASE:

```
UPDATE Auftrag

SET Soll_Auslieferungsdatum =

                CASE DAYOFWEEK_ISO(Auftragsdatum + 10  DAYS)
                WHEN  6 THEN Auftragsdatum + 12  DAYS
                WHEN  7 THEN Auftragsdatum +11 DAYS
                ELSE Auftragsdatum + 10 DAYS
                END

WHERE Soll_Auslieferungsdatum = '01.01.0001';
```

7.6.6.2.1 CASE-Funktion und Pivot-Tabellen

Der Name Pivot-Tabelle stammt aus dem französischen und bedeutet so viel wie Dreh- bzw. Angelpunkt. Mit Hilfe einer Pivot-Tabelle werden Daten verdichtet und häufig als aggregierte Daten dargestellt. Zwar führt die Verdichtung durch Aggregieren zu Informationsverlust bei den Detaildaten, aber dies ist genau der Nutzen einer Pivot-Tabelle: große Datenmengen werden durch Pivot-Tabellen auf eine überschaubare Größe reduziert!

Schauen Sie sich folgende Abbildung an:

KUNDE...	Auftragsjahr	JAN	FEB	MAR	APR	MAI	JUN	JUL	AUG	SEP	OKT	N...	DEZ	Umsatz/Jahr
00001	2010	0.00	750.00	20.00	200.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	970.00
00001	2011	0.00	0.00	40.00	0.00	0.00	3100.00	0.00	0.00	0.00	0.00	0.00	0.00	3140.00
00007	2010	0.00	0.00	920.00	8074.08	0.00	0.00	0.00	0.00	0.00	0.00	0.00	150.00	9144.08
00007	2011	0.00	0.00	40.00	1968.52	984.26	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2992.78
00008	2010	0.00	0.00	20.00	8476.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	8496.00
00008	2011	0.00	0.00	20.00	4800.00	20.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	4840.00
00101	2010	0.00	0.00	0.00	7800.00	0.00	0.00	14850...	0.00	0.00	0.00	0.00	0.00	22650.00
00101	2011	0.00	0.00	1550.00	7500.00	4650.00	7700.00	3300.00	0.00	0.00	0.00	0.00	0.00	24700.00
00102	2010	0.00	0.00	0.00	0.00	0.00	0.00	19701...	7500.00	0.00	0.00	0.00	1200.00	28401.00
00102	2011	0.00	2100.00	2189.00	0.00	2100.00	18900.00	2189.00	0.00	0.00	0.00	0.00	0.00	27478.00
00109	2010	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	6426.00	6426.00
00109	2011	0.00	0.00	0.00	0.00	0.00	4550.86	0.00	0.00	0.00	0.00	0.00	0.00	4550.86
00201	2010	0.00	0.00	0.00	0.00	6000.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	6000.00
00201	2011	0.00	0.00	0.00	0.00	10500.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	10500.00
00400	2010	0.00	0.00	20.00	360.00	0.00	0.00	16200...	0.00	0.00	0.00	0.00	0.00	16580.00
00400	2011	0.00	0.00	0.00	300.00	0.00	0.00	1620.00	0.00	0.00	0.00	0.00	0.00	1920.00
70000	2010	0.00	0.00	0.00	200.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	200.00
70000	2011	0.00	0.00	0.00	200.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	200.00

Pivottabelle – Kundenumsatz je Jahr

Die Tabelle zeigt die Kundenumsätze der Jahre 2010 und 2011 monatsweise im Vergleich. Zusätzlich aber stellt die letzte Spalte die Gesamtumsätze mit den jeweiligen Kunden dar. Diese Auswertung kann mit einer einzigen SQL-Anweisung aufgebaut werden, wenn wir die CASE-Funktion geschickt einsetzen:

7.6.6
Seite 10

```
SELECT Kundennummer, YEAR(Auftragsdatum) AS "Auftragsjahr",
SUM(  CASE WHEN MONTH(Auftragsdatum)=  1
      THEN Auftragswert ELSE 0 END)                  AS Jan,
SUM(   CASE WHEN MONTH(Auftragsdatum) = 2
      THEN Auftragswert ELSE 0 END)                  AS Feb,
SUM(   CASE WHEN MONTH(Auftragsdatum) = 3
      THEN Auftragswert ELSE 0 END)                  AS Mar,
SUM(  CASE WHEN MONTH(Auftragsdatum) = 4
      THEN Auftragswert ELSE 0 END)                  AS Apr,
SUM(  CASE WHEN MONTH(Auftragsdatum) = 5
      THEN Auftragswert ELSE 0 END)                  AS Mai,
SUM(  CASE WHEN MONTH(Auftragsdatum) = 6
      THEN Auftragswert ELSE 0 END)                  AS Jun,
SUM(  CASE WHEN MONTH(Auftragsdatum) = 7
      THEN Auftragswert ELSE 0 END)                  AS Jul,
SUM(  CASE WHEN MONTH(Auftragsdatum) = 8
      THEN Auftragswert ELSE 0 END)                  AS Aug,
SUM(  CASE WHEN MONTH(Auftragsdatum) = 9
      THEN Auftragswert ELSE 0 END)                  AS Sep,
SUM(  CASE WHEN MONTH(Auftragsdatum) = 10
      THEN Auftragswert ELSE 0 END)                  AS Okt,
SUM(  CASE WHEN MONTH(Auftragsdatum) = 11
      THEN Auftragswert ELSE 0 END)                  AS Nov,
SUM(  CASE WHEN MONTH(Auftragsdatum) = 12
      THEN Auftragswert ELSE 0 END)                  AS Dez,
SUM(Auftragswert) AS "Umsatz/Jahr"     -- Gesamtumsatz/Jahr

FROM Auftrag

WHERE   Auftragsdatum BETWEEN  '01.01.2010' AND  '31.12.2011'

GROUP  BY  Kundennummer, YEAR(Auftragsdatum)
ORDER BY  Kundennummer, 2
```

Die Gruppierung sorgt zunächst dafür, dass die Summenbildung je Kunde und je Auftragsjahr erfolgt. Bei der horizontalen Darstellung der einzelnen Monatsumsätze hilft die CASE-Funktion, die dafür allerdings in eine SUM-Funktion eingebettet wird.

7.6.6.3 Jahreszahl, Monatsname, Kalenderwoche ...

Immer wieder wird es vorkommen, dass Sie anstelle eines Datums den entsprechenden Wochentag oder Monatsnamen anzeigen müssen. Aber auch viele Auswertungen werden nur für einzelne Jahre, Monate oder Quartale erstellt. Hier helfen Ihnen die Funktionen YEAR, QUARTER, MONTH, WEEK, WEEK_ISO oder DAY. Schauen wir uns diese Funktionen an einem weiteren Beispiel an.

Ziel ist es jetzt, eine Auswertung zu erstellen, die die Abweichung zwischen dem geplanten SOLL-Lieferdatum und dem tatsächlichen Lieferdatum enthält. Die Auswertung soll sich aber immer nur auf das aktuelle Kalenderjahr beziehen. Hier hilft zunächst einmal die Funktion YEAR. Zusätzlich benötige ich die Funktion CURDATE(), um das aktuelle Kalenderjahr zu ermitteln.

```
SELECT
Auftragsdatum ,
Soll_Auslieferungsdatum ,
Auslieferungsdatum    AS    IST_Auslieferungsdatum
FROM Auftrag

WHERE YEAR(Soll_Auslieferungsdatum) = YEAR(  CURDATE()  )
AND Auslieferungsdatum <> '01.01.0001';
```

Die Funktion YEAR extrahiert die Jahreszahl aus dem Soll-Lieferdatum und vergleicht diesen Wert mit der auf gleiche Weise extrahierten Jahreszahl des Tagesdatums. Aber Vorsicht!! Wenn Sie Funktionen in der WHERE-Klausel verwenden, können häufig keine Indizes zur Ausführung der Abfrage eingesetzt werden. Sowohl die Extraktion als auch die Auflösung des Ausdrucks erfolgt in diesen Fällen im Arbeitsspeicher. Erst mit Release i6.1 haben Sie die Möglichkeit, zur Unterstützung derartiger Abfragen abgeleitete Indizes zu erstellen. Das Kapitel 7.3.5.6 zeigt, was gemeint ist.

Ähnlich wie die YEAR-Funktion arbeiten die Funktionen DAY, MONTH und QUARTER. Argument für die Funktionen sind immer Datumswerte, zurückgegeben wird der Tag, der Monat oder das entsprechende Quartal.

Ein wenig mehr Aufmerksamkeit benötigen die Funktionen WEEK und WEEK_ISO. Die Funktion WEEK liefert immer die Kalenderwoche 1 zurück, wenn die entsprechende Kalenderwoche den 01. Januar beinhaltet. Gut zu erkennen sind die Unterschiede zwischen den beiden Funktionen im vergangenen Jahr. Der 01. Januar fiel auf einen Freitag; der 2. Januar war ein Samstag. Nutzen wir nun für den 01. Januar 2010 die Funktionen WEEK und WEEK_ISO erhalten wir zwei unterschiedliche Ergebnisse:

```
SELECT WEEK('2010-01-01') as WEEK ,
WEEK_ISO('2010-01-01') as Week_iso
FROM SYSIBM.SYSDUMMY1;
```

WEEK	WEEK_ISO
1	53

Unterscheidung WEEK und WEEK_ISO

Die Funktion WEEK_ISO liefert den Wert 1, wenn mindestens vier Tage des Januars in die Kalenderwoche fallen. Nur wenn der erste Januar auf einen Montag, Dienstag, Mittwoch oder Donnerstag fällt, werden Sie keine Unterschiede zwischen den beiden Funktionen feststellen. Dies war 2009 so und wird 2013 wieder so sein:

```
SELECT WEEK('2009-01-01') as WEEK ,
WEEK_ISO('2009-01-01') as Week_iso
FROM SYSIBM.SYSDUMMY1;
```

WEEK	WEEK_ISO
1	1

Manchmal sind keine Unterschiede festzustellen

7.6.6.4 Auswertungen erstellen: Datumsdifferenzen und CTE

In einem zweiten Schritt wollen wir nun die Differenz in Tagen zwischen dem SOLL- und dem IST-Lieferdatum ermitteln. Um die Differenz zwischen dem Soll- und dem Ist-Datum zu ermitteln, könnte man auf den Gedanken kommen die beiden Spalten einfach zu subtrahieren:

```
SELECT
Auftragsdatum ,
Soll_Auslieferungsdatum ,
Auslieferungsdatum    AS    IST_Auslieferungsdatum,
Auslieferungsdatum  -  Soll_Auslieferungsdatum AS
   Diff_in_Tage
FROM Auftrag

WHERE YEAR(Soll_auslieferungsdatum) = YEAR(  CURDATE()  )
AND Auslieferungsdatum <> '01.01.0001';
```

Schaut man sich das Ergebnis an, wird schnell klar, dass die Berechnungen fehlerhaft sind.

AUFTRAGSDATUM	SOLL_AUSLIEFERUNGSDATUM	IST_AUSLIEFERUNGSDATUM	DIFF_IN_TAGE
2011-05-02	2011-05-12	2011-05-10	-2
2011-05-11	2011-05-23	2011-05-27	4
2011-03-10	2011-03-21	2011-04-10	20
2011-02-14	2011-02-24	2011-03-09	13
2011-04-14	2011-04-25	2011-05-21	26
2011-03-13	2011-03-23	2011-03-17	-6
2011-02-14	2011-02-24	2011-03-24	100
2011-04-21	2011-05-02	2011-07-30	228
2011-04-12	2011-04-22	2011-08-12	120
2011-03-17	2011-03-28	2011-04-27	30

DAS Ergebnis der Abfrage ist falsch!

Die Differenz zwischen dem 23.05.2011 und dem 27.05.2011 ist mit vier Tagen korrekt ausgewiesen. Doch zwischen dem 23.02.2011 und dem 24.03.2011 liegen sicherlich keine 100 Tage!

Solange keine Monatsgrenzen berücksichtigt werden müssen, führt die Abfrage zum korrekten Ergebnis. Doch sobald die Datumswerte in verschiedenen Monaten oder gar Jahren liegen, ist das Ergebnis nicht nachvollziehbar. Sie müssen daher die Datumswerte erst in Tage konvertieren und anschließend die Subtraktion durchführen.

7.6.6
Seite 14

```
SELECT
Auftragsdatum ,
Soll_Auslieferungsdatum ,
Auslieferungsdatum    AS    IST_Auslieferungsdatum,
DAYS(Auslieferungsdatum) - DAYS(Soll_Auslieferungsdatum)
   AS Diff_in_Tage
FROM Auftrag

WHERE YEAR(Soll_auslieferungsdatum) = YEAR(  CURDATE()  )
AND Auslieferungsdatum <> '01.01.0001';
```

Abschließend werfen wir noch einen Blick auf das Ergebnis, um zu prüfen, ob die Zeiträume jetzt korrekt berechnet wurden:

Abweichung zum Plan	SOLLLIEFERUNG	ISTLIEFERUNG
34	2011-03-22	2011-04-25
9	2011-05-01	2011-05-10
21	2009-03-20	2009-04-10
-2	2011-03-22	2011-03-20
-3	2009-12-13	2009-12-10
16	2009-08-24	2009-09-09
50	2009-04-23	2009-06-12
58	2009-04-24	2009-06-21
213	2009-05-01	2009-11-30
30	2009-04-21	2009-05-21
51	2009-04-22	2009-06-12

Jetzt ist das Ergebnis korrekt!

Nun können wir mit diesen Daten weitere Auswertungen starten:

- Wie viele Aufträge konnten fristgerecht ausgeführt werden?
- Wie viel Prozent der Aufträge wurden nicht fristgerecht geliefert?
- Um wie viele Tage wurde der Soll-Liefertermin durchschnittlich überschritten?
- Und vieles mehr ...

Jede Auswertung könnte in einer separaten Abfrage sicherlich schnell realisiert werden:

```
SELECT Count(*)

FROM Auftrag

WHERE YEAR(Soll_auslieferungsdatum) = YEAR( CURDATE() )
AND DAYS(auslieferungsdatum) - DAYS(soll_auslieferungsdatum) <= 0
AND auslieferungsdatum <> '01.01.0001';
```

00001
6

Wie viele Aufträge wurden fristgerecht geliefert?

Schwieriger wird es bereits bei der prozentualen Darstellung der nicht fristgerecht gelieferten Waren. Wir benötigen zur Berechnung einerseits die Anzahl aller Lieferungen, andererseits die Anzahl der nicht fristgerecht erfüllten Aufträge. Hier hilft eine Common Table Expression (CTE).

Zunächst ermitteln wir die Anzahl aller Aufträge:

```
SELECT Count(*)

FROM Auftrag

WHERE YEAR(Soll_Auslieferungsdatum) = YEAR(  CURDATE()  )
AND Auslieferungsdatum <> '01.01.0001';
```

7.6.6
Seite 16

Diese Abfrage hinterlegen wir als CTE:

```
WITH

Gesamtanzahl AS

(
SELECT Count(*) AS G_ANZ

FROM Auftrag

WHERE YEAR(Soll_Auslieferungsdatum) = YEAR(  CURDATE()  )
AND Auslieferungsdatum <> '01.01.0001'
)      ,
```

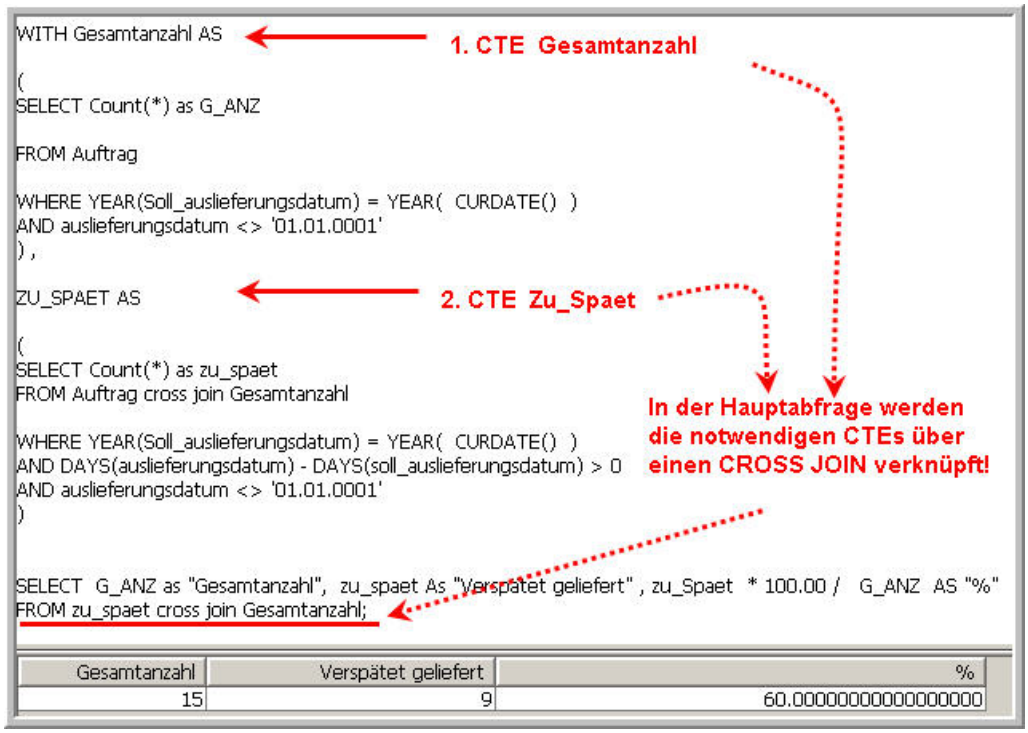
Eingeleitet wird die CTE durch das Schlüsselwort WITH. Es folgt der Name der CTE und in Klammern die zugehörige Abfrage. Wenn weitere Teilabfragen folgen sollen, dürfen Sie das Komma nicht vergessen. In einer zweiten Abfrage ermittle ich jetzt die Anzahl der nicht fristgerecht erfüllten Aufträge.

Ich beginne dieses Mal sofort mit dem Namen, gefolgt vom Schlüsselwort AS.

```
Zu_Spaet   AS

(
SELECT Count(*) AS zu_Spaet
FROM Auftrag CROSS JOIN Gesamtanzahl

WHERE YEAR(Soll_Auslieferungsdatum) = YEAR(  CURDATE()  )
AND DAYS(Auslieferungsdatum) - DAYS(Soll_Auslieferungsdatum)
> 0
AND auslieferungsdatum <> '01.01.0001'
)
```

Das eigentliche SQL-Statement wird wieder in Klammern eingeschlossen.

Anschließend kann ich in der Hauptabfrage sowohl die absoluten Werte als auch die prozentualen Ergebnisse darstellen. Voraussetzung dafür ist lediglich ein CROSS JOIN über die temporären Mengen GESAMTANZAHL und ZU_SPAET:



Einsatz einer CTE für unterschiedliche Berechnungen

Der Einsatz der CTE ermöglicht es uns, unterschiedliche Teilberechnungen durchzuführen, um diese Teilergebnisse anschließend in der Hauptfrage für weitere Berechnungen oder zur Ausgabe zu verwenden. Positiv daran ist v. a., dass die Zwischenergebnisse in der Hauptabfrage namentlich angesprochen werden können und dass es aufgrund der Teilmengen selten syntaktische Konflikte auf Spaltenebene gibt, die insbesondere bei Verwendung von Gruppierungen oder den Aggregatfunktionen auftreten können. Theoretisch kann ich die CTE jederzeit beliebig erweitern.

7.6.6
Seite 18

Wir wollten ja auch noch ermitteln, um wie viele Tage der Soll-Liefertermin durchschnittlich überschritten wird:

```
WITH Gesamtanzahl AS

(
SELECT Count(*) AS G_ANZ

FROM Auftrag

WHERE YEAR(Soll_auslieferungsdatum) = YEAR(  CURDATE()  )
AND auslieferungsdatum <> '01.01.0001'
) ,

ZU_SPAET AS

(
SELECT Count(*) AS zu_spaet
FROM Auftrag

WHERE YEAR(Soll_auslieferungsdatum) = YEAR(  CURDATE()  )
AND DAYS(auslieferungsdatum) - DAYS(soll_auslieferungsdatum)
> 0
AND auslieferungsdatum <> '01.01.0001'
) ,
```

Ich erweitere zu diesem Zweck ein weiteres Mal die Abfrage und ermittle, wie viele Tage insgesamt die Lieferungen verschoben wurden.

```
SUM_Tage_zu_spaet  AS

(
SELECT
SUM( DAYS(auslieferungsdatum) -
     DAYS(soll_auslieferungsdatum)) AS ANZ_TAGE
FROM Auftrag

WHERE YEAR(Soll_auslieferungsdatum) = YEAR(  CURDATE()  )
AND DAYS(auslieferungsdatum) - DAYS(soll_auslieferungsdatum)
> 0
AND auslieferungsdatum <> '01.01.0001'
)
```

Erst in der Hauptabfrage ermittle ich, um wie viele Tage die Waren durchschnittlich zu spät ausgeliefert werden. Dafür verwende ich die temporär aufgebauten Spalten anz_Tage und G_ANZ .

```
SELECT
G_ANZ   AS   "Gesamtanzahl",
zu_spaet AS   "Verspätet geliefert" ,
zu_Spaet  * 100.00 /   G_ANZ  AS "%",
anz_Tage / g_anz      AS "Durchschnittl. zu spät(alle
   Lieferungen)",
anz_Tage / zu_spaet AS "Durchschnittl. zu spät(verspätete
   Lieferungen)"


FROM zu_spaet CROSS  JOIN  Gesamtanzahl CROSS  JOIN
   SUM_tage_zu_spaet;
```

Nähere Erläuterungen zur verwendeten Syntax finden Sie in Kapitel 7.6.5.3.

7.6.6.5 Die Funktionen LAST_DAY, NEXT_DAY und ADD_MONTHS

Nicht immer sind die Berechnungen so einfach wie in den vorangegangenen Beispielen. In vielen Anwendungen ist es beispielsweise notwendig, das Datum des letzten Tages im Monat zu ermitteln. Wer je versucht hat, dieses Datum „zu Fuß“ zu berechnen, wird wissen, dass dies recht aufwendig sein kann. Einige Monate haben 30 Tage, andere 31 und dann ist da noch der Februar, der je nachdem, ob das entsprechende Jahr ein Schaltjahr ist oder nicht, 28 oder 29 Tage hat. Seit Release V5R4 kann man das Monatsende jedoch ganz einfach über die skalare Funktion LAST_DAY ermitteln. Die Funktion erfordert nur einen einzigen Parameter, nämlich ein Datum, eine Zeitmarke oder eine gültige alphanumerische Darstellung eines Datums oder einer Zeitmarke. Basierend auf diesem Datum wird dann das Datum des letzten Tages im Monat ermittelt.

Probieren wir es aus: Heute haben wir den 13. Mai 2011. Der letzte Tag in diesem Monat ist der 31. Mai, ein Dienstag.

```
SELECT LAST_DAY('13.05.2011') AS "Monatsende",
DAYNAME( LAST_DAY('13.05.2011')  ) AS "letzter Wochentag des Monats"
FROM SYSIBM.SYSDUMMY1;
```

Monatsende	letzter Wochentag des Monats
31.05.2011	Dienstag

Wirkungsweise der Funktion LAST_DAY

Im Gegensatz zur Funktion LAST_DAY ermittelt die skalare Funktion NEXT_DAY den nächsten Wochentag in Abhängigkeit eines Datums. Stellen wir uns vor, dass ein Auftrag am 02. Mai 2011, ein Montag, erfasst wurde. Der Kunde wird aber immer am Freitag beliefert. Das Soll-Lieferdatum wäre dann der 06. Mai 2011.

Diese Aufgabenstellung lässt sich mit der Funktion NEXT_DAY realisieren.

```
SELECT    NEXT_DAY('02.05.2011', 'FR')
   FROM SYSIBM.SYSDUMMY1;
```

Die Funktion benötigt zwei Parameter: Im ersten Parameter wird das Ausgangsdatum, eine Zeitmarke oder eine gültige alphanumerische Darstellung eines Datums oder einer Zeitmarke übergeben. Im zweiten Parameter wird der gewünschte Wochentag als Name – entweder ausgeschrieben oder als Abkürzung – angegeben. Groß- und Kleinschreibung sowie führende oder nachfolgende Blanks spielen hierbei keine Rolle. Die Wochentagnamen und deren Abkürzungen sind im Übrigen in den Message-IDs CPX9034 und CPX9039 in der Nachrichtendatei QCPFMSG hinterlegt.

```
Nachrichten-ID . . . . . :   CPX9039
Nachrichtendatei . . . . . :   QCPFMSG
  Bibliothek . . . . . :     QSYS

Nachricht . . . :   MO    DI    MI    DO    FR    SA    SO

 Nachrichten-ID . . . . . :   CPX9034
 Nachrichtendatei . . . . . :   QCPFMSG
   Bibliothek . . . . . :     QSYS

 Nachricht . . . :   Montag            Dienstag          Mittwoch
   Donnerstag        Freitag           Samstag           Sonntag
```

Die Nachrichten-ID CPX9039 und CPX9034

Als Ergebnis erhalten Sie immer eine Zeitmarke. War der Ursprungswert eine Zeitmarke, enthält das Ergebnis die gleichen Zeitangaben wie die Ausgangszeitmarke. War der Ursprungswert ein Datum, wird die Zeit mit 00:00:00.000000 initialisiert.

Auslieferungstag
2011-05-06 00:00:00.000000

Ergebnis der Funktion NEXT_DAY

7.6.6
Seite 22

Wie können wir diese Funktion jetzt in unsere Soll-Planung integrieren? Bislang haben wir bei dem Soll-Liefertermin nicht berücksichtigt, dass unsere Kunden nur an einem bestimmten Wochentag beliefert werden. Welche Tage dies sind, steht beispielsweise in einer separaten Tabelle. In meinem Beispiel ist dies die Tabelle TOURENPLANUNG. Ich kann jetzt unter Berücksichtigung der hinterlegten Wochentage nochmals meinen Soll-Liefertermin korrigieren:

```
SELECT
Kundennummer,
Soll_Auslieferungsdatum,
NEXT_DAY(
          Soll_Auslieferungsdatum ,           -- 1. Argument

          (SELECT auslieferungstag FROM tourenplanung
                                              --  2. Argument
           WHERE Tourenplanung.Kundennummer =
                                        Auftrag.Kundennummer)
           )
FROM Auftrag
WHERE Soll_Auslieferungsdatum  BETWEEN    '01.06.2011' and
'30.06.2011'
AND       Auslieferungsdatum = '01.01.0001';
```

Auffällig ist, dass ich als zweites Argument in dem Beispiel keine Spalte oder Konstante verwende, sondern ein Sub-Select, das mir den hinterlegten Liefertag für den entsprechenden Kunden ermittelt. Weitere Hinweise zum verwendeten Sub-Select enthält das Kapitel 7.6.5

Ebenfalls im Release V5R4 kam die Funktion ADD_MONTHS neu hinzu. Wie der Name es vermuten lässt, können Sie mit der Funktion Monate zu einem Datumswert hinzuaddieren. Die Funktion verlangt zwei Argumente. Das erste Argument muss eine Datumsspalte, eine Zeitmarke oder eine gültige alphanumerische Darstellung eines Datums oder einer Zeitmarke sein. Ebenso kann ein Ausdruck verwendet werden, dessen Ergebnis ein gültiges Datum ist. Das zweite Argument benennt die Anzahl Monate, die addiert werden sollen:

```
ADD_MONTHS(datum, 2) -- zwei Monate werden addiert
```

Nun, das konnten wir auch schon vorher mit der Funktion MONTH problemlos realisieren, und einfacher ist es auch:

```
Datum + 2 MONTH -- zwei Monate werden addiert
```

Tatsächlich lassen sich im Ergebnis häufig keine Unterschiede zwischen den beiden Funktionen feststellen, wie folgende Abbildung beweist:

```
SELECT
Auftragsdatum,

ADD_MONTHS(auftragsdatum, 2) AS "add_month",

Auftragsdatum + 2 MONTH  AS "MONTH"

FROM  Auftrag;
```

AUFTRAGSDATUM	add_month	MONTH
02.05.2011	02.07.2011	02.07.2011
03.12.2009	03.02.2010	03.02.2010
14.08.2009	14.10.2009	14.10.2009
13.04.2009	13.06.2009	13.06.2009
14.04.2009	14.06.2009	14.06.2009
21.04.2009	21.06.2009	21.06.2009
11.04.2009	11.06.2009	11.06.2009

Wirkungsweise der Funktion ADD_MONTHS

Das liegt daran, dass beide Funktionen nach den gleichen Regeln arbeiten:

- Solange der Tag im errechneten Monat vorhanden ist, bleibt die Tageszahl bestehen.
- Ist hingegen der laufende Tag im neuen Monat nicht vorhanden, etwa aufgrund einer unterschiedlichen Tagesanzahl, so wird der Monatsletzte als Ergebnis angegeben.

```
SELECT
Auftragsdatum,

ADD_MONTHS(auftragsdatum, 1) AS "add_month",

Auftragsdatum + 1 MONTH  AS "MONTH"

FROM  Auftrag
where day(Auftragsdatum) in ('30', '31');
```

AUFTRAGSDATUM	add_month	MONTH
31.01.2011	28.02.2011	28.02.2011
31.05.2011	30.06.2011	30.06.2011

Verwendung des letzten Tages im Monat

Soweit so gut! Aufgrund dieser Regelung kann es passieren, dass beim Zurückrechnen des Datums nicht der ursprüngliche Wert ermittelt wird.

Ergänzung 18./2011

Schauen Sie sich das folgende Beispiel an. Angenommen, mit dem zuvor gezeigten SQL-Statement wurde anhand des Auftragsdatums als Soll-Liefertermin der 28.02.2011 ermittelt. Dann müsste man anhand des Liefertermins das ursprüngliche Auftragsdatum rekonstruieren können, indem man vom Soll-Liefertermin einen Monat subtrahiert:

```
SELECT
 Soll_Auslieferungsdatum,

 ADD_MONTHS(Soll_auslieferungsdatum, -1) AS "add_MONTH",

 Soll_Auslieferungsdatum -1 MONTH  AS "MONTH"

 FROM  Auftrag
 WHERE Soll_Auslieferungsdatum = '28.02.2011';
```

Und hier zeigt sich der Unterschied zwischen den Funktionen ADD_MONTHS und MONTH. Subtrahiert man einen Monat, weicht das errechnete Datum ggf. vom Ursprungsdatum ab, wenn Sie für die Subtraktion die Funktion MONTH verwenden.

```
SELECT
Soll_auslieferungsdatum,

ADD_MONTHS(Soll_auslieferungsdatum, -1) AS "add_month",

soll_auslieferungsdatum - 1 MONTH  AS "MONTH"

FROM  Auftrag
where soll_auslieferungsdatum = '28.02.2011';
```

SOLL_AUSLIEFERUNGSDATUM	add_month	MONTH
28.02.2011	31.01.2011	28.01.2011

Unterschied zwischen der Funktion MONTH und ADD_MONTHS

Oftmals ist genau dieses Ergebnis nicht erwünscht. Vielmehr soll der Monatsletzte als Ergebnis verwendet werden. Mit Release V5R4 deckt IBM jetzt diese Anforderung ab, indem die skalare Funktion ADD_MONTHS zur Verfügung gestellt wird.

7.6.6.6 Logische Funktionen, Konvertierungs- und Zeichenkettenfunktionen

In den vorangegangenen Kapiteln haben wir viel mit Datumsfunktionen gearbeitet. Nur, was nützen die vielen schönen Funktionen, wenn die Datumsfelder als Zeichenvariable oder als Dezimalwert gespeichert wurden? In diesem Kapitel möchte ich Ihnen zeigen, wie Sie Ihre Felder aufbereiten können, um anschließend problemlos die Datumsfunktionen nutzen zu können:

Ausgangsbasis für die folgenden Beispiele ist die eigens dafür eingerichtete Tabelle KONVERTIERUNG_DATUM. Die Tabelle hat vier Spalten, die folgendermaßen deklariert wurden:

```
CREATE TABLE    Konvertierung_Datum
(
Datum     DATE              NOT NULL ,
Datum_Char      CHAR(8)           NOT NULL DEFAULT ' ' ,
Datum_Dec    DECIMAL(8, 0)  NOT NULL DEFAULT 0 ,
Datum_C_DEC  DECIMAL(7, 0)  NOT NULL DEFAULT 0
) ;
```

Eigentlich ist es sehr einfach, Felder zu konvertieren. SQL stellt neben den Konvertierungsfunktionen CHAR, DEC, VARCHAR viele Konvertierungsfunktionen bereit. Die Liste der Skalarumsetzungsfunktionen ist lang und soll an dieser Stelle nicht vollständig aufgeführt werden. Hier hilft bei Bedarf das entsprechende SQL-Referenzhandbuch.

Wir wollen uns in einem ersten Schritt lediglich die Skalarfunktion DATE ansehen. In der „Hilfe“ des System i Navigators heißt es: „Die Funktion DATE benötigt als einziges Argument eine Zeitmarke, ein Datum, eine positive Zahl, die kleiner oder gleich 3.652.059 ist, oder eine gültige Zeichenfolgedarstellung eines Datums oder eine Zeichenfolge der Länge 7“. Als Rückgabewert sollen wir ein Datum erhalten.

```
SELECT datum, DATE(datum_char), DATE(datum_dec), DATE(datum_c_dec), DATE(datum_char_7)
FROM Konvertierung_datum;
```

DATUM	00002	00003	00004	00005
2011-05-12	+++++++++++++++	++++++++++...	3041-06-24	+++++++++++++++
2011-04-28	+++++++++++++++	++++++++++...	3041-04-01	+++++++++++++++
0001-01-01	+++++++++++++++	++++++++++...	+++++++++++++++	-

Die Konvertierungsfunktion DATE

Keine meiner Spalten wurde korrekt als Datum übersetzt. Warum?

7.6.6
Seite 26

Wenn das Argument der DATE-Funktion eine Ziffer ist, wie beispielsweise die Spalte 00004, wird das übergebene Argument als Anzahl Tage seit dem 01.01.0001 interpretiert. Wenn das Argument eine Zeichenkette ist, wird die Zeichenkette in ein Datum konvertiert, aber nur dann, wenn die Zeichenkette ein gültiges Datum repräsentiert. SQL kennt allerdings nur die folgenden Datumsformate:

Formatname	Format
ANSI-/ISO-Format	JJJJ-MM-TT
IBM USA Standardformat und Japanischer Industriestandard	JJJJ/MM/DD
IBM europäisches Standardformat	TT.MM.JJJJ
Unformatiertes julianisches Format	JJJJTTT
Formatiertes julianisches Format	JJ/TTT
Monat, Tag, Jahr	MM/TT/JJ
Tag, Monat, Jahr	TT/MM/JJ
Jahr, Monat, Tag	JJ/MM/TT

Unsere Datumswerte liegen in keinem dieser Formate vor. Wir müssen daher den Inhalt unserer Spalten erst einmal in ein für SQL verständliches Format bringen. Beginnen wollen wir mit der Spalte DATUM_CHAR. Zur Aufbereitung benötigen wir Zeichenkettenfunktionen oder besser gesagt die Funktion SUBSTRING oder SUBSTR. Die Funktion extrahiert einen bestimmten Abschnitt einer Zeichenfolge. Das erste Argument definiert die Quelle, das zweite Argument der Funktion bestimmt die Startposition und das dritte Argument enthält die Längenangabe.

```
SELECT Datum,   SUBSTR(Datum_CHAR, 7, 2)  || '.' ||
SUBSTRING(Datum_CHAR, 5, 2) || '.' || SUBSTR(Datum_CHAR,
   1, 4)
FROM Konvertierung_Datum;
```

Wir erhalten das folgende Ergebnis:

DATUM	00002
2011-05-12	12.05.2011
2011-04-28	28.04.2011
0001-01-01	-

DAS Ergebnis der Funktion SUBSTRING

Erst jetzt können wir die zweite Spalte mit der Funktion DATE in ein Datumsfeld konvertieren und natürlich beliebig aus dem Fundus der Datumsfunktionen schöpfen:

```
SELECT Datum,
DAYNAME(
     DATE(
               SUBSTR(Datum_CHAR, 7, 2)  || '.' || SUBSTRING
               (Datum_CHAR, 5, 2) || '.'
               || SUBSTR(Datum_CHAR, 1, 4)
               )
)

FROM Konvertierung_Datum;
```

Das Ergebnis stimmt:

DATUM	00002
2011-05-12	Donnerstag
2011-04-28	Donnerstag
0001-01-01	-
0001-01-01	+++++++++++++++

Ergebnis der Typkonvertierung

Sie können anhand des Ergebnisses aber feststellen, dass der Wert NULL in der dritten Tabellenzeile sowie ein nicht darstellbarer Datumswert in der vierten Tabellenzeile Probleme bereiten.

Die vierte Tabellenzeile enthält den Wert 30.02.2011. Dieser Wert ist kein gültiges Datum. Leider gibt es bislang keine SQL-Standardfunktion, mit der wir die Gültigkeit eines Datums prüfen können. Hierfür müssen wir eigene SQL-Funktionen (UDF) programmieren.

Die Spalte DATUM_CHAR in der dritten Tabellenzeile enthält Leerzeichen. Anstelle dieser Leerzeichen erhalten wir durch die DATE-Funktion und durch die Funktion DAYNAME den Wert NULL als Ergebnis. Doch dafür lässt sich schnell eine Lösung finden: Durch die logischen Funktionen IFNULL und COALESCE lassen sich NULL-Werte durch beliebige Zeichen ersetzen:

```
SELECT
Datum,
IFNULL( DAYNAME( DATE( SUBSTR(Datum_Char, 7, 2)  || '.' || SUBSTR(Datum_Char, 5, 2) || '.' ||
                       SUBSTR(Datum_Char, 1, 4) )), 'ohne Datum')
FROM Konvertierung_datum;
```

DATUM	00002
2011-05-12	Donnerstag
2011-04-28	Donnerstag
0001-01-01	ohne Datum

Die Funktion IFNULL

Als Nächstes konvertieren wir die numerische Spalte DATUM_DEC in ein gültiges Datumsformat. Die DATE-Funktion akzeptiert nur numerische Werte, die kleiner oder gleich dem Wert 3.652.059 sind. Unsere Zeilen enthalten die Werte 20.110.514. Die Zahl ist deutlich größer als erlaubt, so dass auch diese Werte nicht so einfach konvertiert werden können. Wir müssen die Spalte zunächst in ein Zeichenfeld umwandeln und können anschließend wie zuvor geschildert vorgehen:

```
SUBSTR(CHAR(Datum_Dec), 7, 2)  || '.' ||
   SUBSTR(CHAR(Datum_Dec), 5, 2) || '.' ||
SUBSTR(CHAR(Datum_Dec), 1, 4)
```

Die Funktion CHAR gibt das erste Argument in Form einer Zeichenfolge zurück. Als Argumente dürfen DATE- und TIME-Spalten, ganze Zahlen, gepackte und ungepackte Dezimalzahlen und Gleitkommazahlen verwendet werden. Führende Nullen gehen bei der Konvertierung verloren. Wenn Sie führende Nullen im Zeichenfeld darstellen müssen, dann sollten Sie die Daten mit der Funktion DIGITS konvertieren. Die folgende Abbildung soll Ihnen die Unterschiede zwischen den beiden Funktionen verdeutlichen:

```
SELECT CHAR(0110.99) , DIGITS(0110.99)
FROM Sysibm.Sysdummy1;
```

00001	00002
110.99	011099

Unterschiede zwischen CHAR und DIGITS

Lange Zeit war die Typkonvertierung der numerischen Spalten zwingend erforderlich, wenn anschließend Zeichenkettenfunktionen eingesetzt werden sollten. Heute – in Release i6.1 – kann die Konvertierung teilweise auch implizit durch die Zeichenkettenfunktion erfolgen:

```
SUBSTR(Datum_Dec, 7, 2)  || '.' ||
   SUBSTR(Datum_Dec, 5, 2) || '.' ||
SUBSTR(Datum_Dec, 1, 4)
```

Sie sehen, beide Ausdrücke führen zum gleichen Ergebnis:

DATUM	Mit Konvertierung	ohne Konvertierung
2011-05-12	12.05.2011	12.05.2011
2011-04-28	28.04.2011	28.04.2011
0001-01-01	. .0	. .0
0001-01-01	30.02.2011	30.02.2011

Ergebnis der impliziten Typkonvertierung

Ich empfehle Ihnen allerdings explizit, die Typkonvertierung durchzuführen, denn nicht alle Interfaces reagieren diesbezüglich identisch!

Anhand der Grafik können Sie schon erkennen, dass uns dieses Mal wieder die dritte Zeile Probleme bereitet. Die Spalte enthält den Originalwert 0. Als Ergebnis erhalten wir die Zeichenfolge: Leerzeichen Leerzeichen Punkt Leerzeichen Leerzeichen Punkt 0 Leerzeichen Leerzeichen Leerzeichen. Die Funktion IFNULL kann uns im ersten Schritt daher nicht helfen. Wir können das Problem aber mit der Funktion NULLIF lösen. Die Funktion NULLIF gibt den Wert NULL zurück, wenn die beiden Argumente der Funktion identisch sind, andernfalls wird der Wert des ersten Arguments zurückgegeben.

```
NULLIF(     SUBSTR(CHAR(Datum_Dec), 7, 2)  || '.' ||
               SUBSTR(Datum_Dec, 5, 2)
            || '.' ||     SUBSTR(Datum_Dec,
               1, 4) ,     '  .  .0   ')
```

7.6.6
Seite 30

Anschließend können wir dann wie gehabt die Funktion IFNULL verwenden, um den NULL-Wert durch ein beliebiges Datum zu ersetzen:

```
IFNULL(
NULLIF(     SUBSTR(CHAR(Datum_Dec), 7, 2)  || '.' ||
               SUBSTR(Datum_Dec, 5, 2)
            || '.' ||     SUBSTR(CHAR(Datum_Dec),
            1, 4) ,    '  .  .0   ')  ,  '01.01.0001')
```

Als Letztes konvertieren wir die Spalte Datum_C_DEC. Die Werte sind im Format CJJMMTT gespeichert. Dieses Mal sind die numerischen Werte kleiner als der Wert 3.652.059, so dass die DATE-Funktion ein Ergebnis liefert; allerdings ist das Ergebnis unsinnig, wie folgende Abbildung zeigt:

DATUM	00002
2011-05-12	3041-06-24
2011-04-28	3041-04-01
0001-01-01	++++++++++...
0001-01-01	3040-09-15

Wir erhalten unsinnige Ergebnisse

Damit wir mit den Inhalten der Spalte arbeiten können, müssen wir die Jahrhundertziffer nutzen, um eine gültige Jahreszahl zu erhalten. Für die Lösung gibt es verschiedene Ansätze:

- Wir könnten wieder die CASE-Funktion nutzen

```
CASE SUBSTR(DIGITS(Datum_C_Dec), 1, 1)

WHEN  '1'
THEN  SUBSTR(DIGITS(Datum_C_Dec), 6, 2) || '.' ||
            SUBSTR(DIGITS(Datum_C_Dec), 4, 2) ||
           '.20' || SUBSTR(DIGITS(Datum_C_Dec), 2, 2)

WHEN  '0'
THEN  SUBSTR(DIGITS(Datum_C_Dec), 6, 2) || '.' ||
            SUBSTR(DIGITS(Datum_C_Dec), 4, 2) ||
            '.19' ||  SUBSTR(DIGITS(Datum_C_Dec), 2, 2)
ELSE 'ungültiges Datum'
END
```

Oder wir könnten den Wert 19.000.000 zum Inhalt der Spalte addieren, damit wir die vierstellige Jahreszahl erhalten

```
SUBSTR(DIGITS(Datum_C_Dec ) , 6, 2) || '.' ||
SUBSTR(DIGITS(Datum_C_Dec)  , 4, 2) || '.' ||
SUBSTR(Datum_C_Dec + 19000000 , 1, 4)
```

Unabhängig davon, wie Ihre Datumswerte in der Datenbank gespeichert wurden, sobald wir Dezimal- oder Zeichenspalten konvertieren müssen, sind die Lösungen relativ aufwendig und ich kenne niemanden, der wirklich Lust hätte, für jede SQL-Abfrage diese Konvertierungen vorzunehmen. Der Schreibaufwand ist hoch und auch die Fehlerquellen sind vielfältig. Sie können aber problemlos eine VIEW erstellen, die die konvertierten Felder enthält:

7.6.6
Seite 32

```
CREATE VIEW   Konvertiertes_Datum (Datum1, Datum2, Datum3)
 AS
(
SELECT
 IFNULL(  DATE(SUBSTR(Datum_Char, 7, 2)  || '.' ||
               SUBSTR(Datum_Char, 5, 2)
               || '.' || SUBSTR(Datum_Char, 1, 4 )),
               '01.01.0001') AS "Char-Spalte JJJJMMTT",

IFNULL(  DATE
                  (NULLIF(
                      SUBSTR(Datum_Dec, 7, 2)  || '.' ||
                      SUBSTR(Datum_Dec, 5, 2)
                      || '.' || SUBSTR(Datum_Dec, 1, 4) ,
                   '  .  .0   ')
                    ),
 '01.01.0001') AS "Dezimale Spalte JJJJMMTT"  ,

IFNULL( DATE
                 (NULLIF
                    (
                    SUBSTR(DIGITS(Datum_C_Dec ) , 6, 2),
               '00' )

                    || '.' || SUBSTR(DIGITS(Datum_C_Dec)
                       , 4, 2) || '.' ||
                    SUBSTR(Datum_C_Dec + 19000000 , 1, 4)
                  ) ,
'01.01.0001') AS "Dezimale Spalte CJJMMTT"

 FROM Konvertierung_datum
);
```

Den Aufwand haben Sie natürlich ebenfalls – allerdings nur ein einziges Mal. Anschließend können Sie mit Ihren Daten problemlos arbeiten:

```
SELECT DAYNAME(Datum1),  LAST_DAY(Datum2),  Datum2 + 5 DAYS
FROM Konvertiertes_Datum
ORDER BY  1;
```

00001	00002	00003
Donnerstag	2011-04-30	2011-05-03
Montag	0001-01-31	0001-01-06
Montag	1997-06-30	1997-06-28
Samstag	2011-05-31	2011-05-17

Einsatz einer SQL-View

Weitere Informationen zum Einsatz von SQL-Views enthält das Kapitel 7.3.3.6. Das Problem ungültiger Datumswerte wird auf diesem Weg leider nicht behoben, aber wer sagt denn, dass Ihre Tabellen fehlerhafte Datumswerte enthalten?

7.6.6
Seite 34

7.6.6.7 Datenverschlüsselung

Als Datenverschlüsselung bezeichnet man den Vorgang, bei dem ein klar lesbarer Text, aber auch Ton- oder Bildaufzeichnungen mit Hilfe eines Verschlüsselungsverfahrens in eine „unleserliche“, das heißt nicht interpretierbare Zeichenfolge umgewandelt wird. Alle auf dem i-Server verwendeten Schlüsselungsverfahren sind symmetrische Verschlüsselungsalgorithmen und wie die meisten symmetrischen Verschlüsselungsverfahren handelt es sich um Blockverschlüsselungsalgorithmen, d. h. bevor die Daten verschlüsselt werden, werden sie zunächst in gleich große Blöcke unterteilt. Die Blöcke werden anschließend jeder einzeln verschlüsselt. Die Verschlüsselungsroutine setzt jedoch ganze Blöcke voraus, somit muss meistens der letzte Block mit Zufallsdaten aufgefüllt werden. Die jeweilige Blockgröße wird hierbei durch den Algorithmus vorgegeben. Derzeit werden drei Verfahren unterstützt:

Der **Data Encryption Standard (DES)** ist das älteste Verfahren, es wurde von IBM entwickelt und 1976 als US-Standard anerkannt. Beim DES handelt es sich um eine Blockchiffre, die auf Blöcken mit 64 Bits arbeitet. Aus den 64 Bit großen Blöcken des Klartextes erzeugt der symmetrische Algorithmus 64 Bit große Chiffriertexte. Der dazu verwendete Schlüssel ist ebenfalls 64 Bit lang, allerdings ist jedes achte Bit ein Paritätsbit, so dass nur 56 Bit des Schlüssels zur Ver- und Entschlüsselung benutzt werden. Die Zweifel an der Sicherheit von DES aufgrund der kurzen 56 Bit langen Schlüssel hörten jedoch nie auf und waren letztlich auch begründet. 1998 bestätigte die Electronic Frontier Foundation die Zweifel, indem sie eine Maschine zur Dechiffrierung von DES mittels Brute-Force-Verfahren baute. Als Reaktion darauf bot sich als eine vorübergehende Lösung 3DES (=Triple-DES, TDES) an. Die dreifache Verschlüsselung der Daten ist jedoch recht langsam und auch die verwendete Blocklänge von 64 Bit ist ein potenzielles Sicherheitsrisiko. Ein weiterer Kritikpunkt war, dass das Verschlüsselungsverfahren lange Zeit nicht vollständig veröffentlicht wurde und die Originalversion durch die National Security Agency (NSA) überarbeitet wurde, bevor DES zum Standard gemacht wurde. Deshalb wurden sogar Hintertüren der NSA vermutet, die allerdings nie gefunden wurden. So blieb es bei der Vermutung, dass die NSA bereits 1976 in der Lage war, den Code zu knacken. Aus diesen Gründen wurde DES auch in keinem anderen Land als offizieller Standard anerkannt. Und dennoch war DES lange Zeit der am häufigsten genutzte symmetrische Algorithmus zum Verschlüsseln von Daten.

Ergänzung 19/2011

RC2 ist ebenfalls eine 64 Bit Blockchiffre mit variabler Schlüssellänge, die von Ronald Rivest als möglicher Ersatz für DES im Jahr 1987 entwickelt wurde. RC steht für Rivest Cipher oder Ron's Code. Die Entwicklung von RC2 wurde von Lotus gesponsert, welche nach einer kundenspezifischen Chiffre suchten. Nachdem diese von der NSA evaluiert wurde, konnte sie als Teil der Software Lotus Notes außerhalb der Vereinigten Staaten exportiert werden. Die NSA schlug viele Änderungen am Algorithmus vor, die dann von Ronald Rivest eingearbeitet wurden. Nach weiteren Verhandlungen wurde die Blockchiffre für den Export freigegeben. Parallel mit RC4 fiel RC2 mit einer Schlüssellänge von 40-Bit nicht unter die amerikanischen Exportbeschränkungen für Kryptographie. Ursprünglich wurden die Details des Algorithmus als Eigentum der Firma RSA Security geheim gehalten. Doch am 29. Januar 1996 wurde der Quellcode von RC2 anonym im Forum scy.crypt eingetragen. Eine Aufdeckung des Quellcodes im ähnlichen Stil fand auch bei RC4 statt. Es ist bis heute unklar, ob der Verfasser Zugang zum Quellcode hatte, oder ob RC2 durch so genanntes Reverse Engineering aufgedeckt wurde. Desweiteren gilt das RC2-Verfahren als verwundbar gegenüber einem Angriff mit verwandtem Schlüssel. Eine Analyse bestätigte 1997 diesen Verdacht.

Es musste also ein neuer Verschlüsselungsstandard entwickelt werden, der nicht die Fehler seiner Vorgänger besitzt. Das US-amerikanische National Institute of Standards and Technology (NIST) hat Anfang 1997 zu einem offenen Wettbewerb aufgerufen, dessen Sieger als **Advanced Encryption Standard (AES)** festgelegt werden sollte. Dabei wurden folgende Kriterien aufgestellt, die von den Algorithmen zu erfüllen sind:

- AES muss ein symmetrischer Blockalgorithmus sein.
- AES muss mindestens 128 Bit lange Blöcke verwenden und Schlüssel von 128, 192 und 256 Bit Länge einsetzen können.
- AES soll gleichermaßen leicht in Hard- und Software zu implementieren sein.
- AES soll in Hard- wie Software eine überdurchschnittliche Performance haben.
- AES soll allen bekannten Methoden der Kryptanalyse (die Kunst, einen Geheimtext ohne Kenntnis des Schlüssels zu dechiffrieren) widerstehen können, insbesondere Power- und Timing-Attacken.
- Speziell für den Einsatz in Smartcards sollen geringe Ressourcen erforderlich sein (kurze Codelänge, niedriger Speicherbedarf).
- Der Algorithmus muss frei von patentrechtlichen Ansprüchen sein und darf von jedermann unentgeltlich genutzt werden.

Im Mai 2000 wurden die Analysen und öffentlichen Diskussionen abgeschlossen und im Oktober 2000 wurde der belgische Algorithmus Rijndael als neuer AES-Standard festgelegt. Grund hierfür waren insbesondere die Einfachheit und gute Performance. AES ist, wie gefordert ein Blockchiffre, dessen Blocklänge und Schlüssellänge unabhängig voneinander die Werte 128, 192 oder 256 Bit erhalten kann. Bei AES ist die Blockgröße somit variabel.

Alle genannten Verschlüsselungsverfahren sind seit Release i6.1 als SQL-Funktionen verfügbar. Allerdings wurden sie in unterschiedlichen Releases implementiert wie folgende Tabelle zeigt:

Funktion	Beschreibung
ENCRYPT_RC2 Release V5R3	Verschlüsselt eine **Zeichenfolge** mit dem RC2-Algorithmus. Unterstützte Datentypen CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, VARBINARY, BLOB, CLOB oder DBCLOB. Das Passwort kann eine Länge von 6 bis 127 Bytes haben. Das Passwort wird für die Entschlüsselung verwendet. Wird kein Passwort angegeben, wird der aktuelle Wert der Option SET ENCRYPTION PASSWORD verwendet. Passwörter sollten in den Applikationen nicht als Literale gespeichert werden. Vielmehr empfiehlt es sich, Hostvariablen oder die Option SET ENCRYPTION PASSWORD zu verwenden. Wenn Hinweise gespeichert werden, können diese mit der Funktion GETHINT abgerufen werden. Als Ergebnis liefert die Funktion eine BLOB-Spalte, wenn die zu verschlüsselnde Zeichenfolge den Datentyp LOB hat. Sie erhalten eine Ergebnisspalte vom Typ VARBINARY, wenn die Zeichenfolge BINARY oder VARBINARY ist, und in allen anderen Fällen VARCHAR FOR BIT DATA. Das für die Entschlüsselung zu verwendende Kennwort muss ein CHAR- oder VARCHAR-Wert sein, der das Kennwort darstellt, mit dem die Zeichenfolge verschlüsselt wird. Die Angabe eines Kennwortes ist optional. Wird das Kennwort nicht angegeben, werden die Daten mit dem Wert von ENCRYPTION PASSWORD verschlüsselt und entschlüsselt, der mit dem Befehl SET ENCRYPTION PASSWORD zugeordnet wurde.

7.6.6
Seite 38

ENCRYPT_TDES **Release V5R4**	Verschlüsselt eine Zeichenfolge mit dem 3DES-Algorithmus. Unterstützte Datentypen CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, VARBINARY, BLOB, CLOB oder DBCLOB. Das Passwort kann eine Länge von 6 bis 127 Bytes haben. Es dürfen keine CLOB-Variablen verwendet werden. Das Passwort wird für die Entschlüsselung verwendet. Wird kein Passwort angegeben, wird der aktuelle Wert der Option SET ENCRYPTION PASSWORD verwendet. Passwörter sollten in den Applikationen nicht als Literale gespeichert werden. Vielmehr empfiehlt es sich, Hostvariablen oder die Option SET ENCRYPTION PASSWORD zu verwenden. Wenn Hinweise gespeichert werden, können diese mit der Funktion GETHINT abgerufen werden. Als Ergebnis liefert die Funktion eine BLOB-Spalte, wenn die zu verschlüsselnde Zeichenfolge den Datentyp LOB hat. Die Zielspalte ist vom Typ VARBINARY, wenn die Quellzeichenfolge BINARY oder VARBINARY ist, und in allen anderen Fällen ist die Zielspalte vom Typ VARCHAR FOR BIT DATA. Das für die Entschlüsselung zu verwendende Kennwort muss ein CHAR- oder VARCHAR-Wert sein, der das Kennwort darstellt, mit dem die Zeichenfolge verschlüsselt wird. Die Angabe eines Kennwortes ist optional. Wird das Kennwort nicht angegeben, werden die Daten mit dem Wert von ENCRYPTION PASSWORD verschlüsselt und entschlüsselt, der mit dem Befehl SET ENCRYPTION PASSWORD zugeordnet wurde.

ENCRYPT_AES **Release i6.1**	Verschlüsselt eine Zeichenfolge mit dem AES-Algorithmus. Als Ausgangsformate können die Datentypen CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, VARBINARY, BLOB, CLOB oder DBCLOB verwendet werden. Das Passwort kann eine Länge von 6 bis 127 Bytes haben. Es dürfen keine CLOB-Variablen verwendet werden. Das Passwort wird für die Entschlüsselung verwendet. Wird kein Passwort angegeben, wird der aktuelle Wert der Option SET ENCRYPTION PASSWORD verwendet. Passwörter sollten in den Applikationen nicht als Literale gespeichert werden. Vielmehr empfiehlt es sich, Hostvariablen oder die Option SET ENCRYPTION PASSWORD zu verwenden. Wenn Hinweise gespeichert werden, können diese mit der Funktion GETHINT abgerufen werden. Als Ergebnis liefert die Funktion eine BLOB-Spalte, wenn die zu verschlüsselnde Zeichenfolge den Datentyp LOB hat. Die Zielspalte ist vom Typ VARBINARY, wenn die Quellzeichenfolge BINARY oder VARBINARY ist, und in allen anderen Fällen ist die Zielspalte vom Typ VARCHAR FOR BIT DATA. Das für die Entschlüsselung zu verwendende Kennwort muss ein CHAR- oder VARCHAR-Wert sein, der das Kennwort darstellt, mit dem die Zeichenfolge verschlüsselt wird. Die Angabe eines Kennwortes ist optional. Wird das Kennwort nicht angegeben, werden die Daten mit dem Wert von ENCRYPTION PASSWORD verschlüsselt und entschlüsselt, der mit dem Befehl SET ENCRYPTION PASSWORD zugeordnet wurde.
DECRYPT_....	Gibt den Wert zurück, der das Ergebnis der Entschlüsselung verschlüsselter Daten ist. Die Entschlüsselungsfunktion kann nur die Funktionen entschlüsseln, die mit der Funktion ENCRYPT_RC2 oder ENCRYPT_TDES verschlüsselt wurden.
DECRYPT_BIT	Wird verwendet für die Datentypen CHAR oder VARCHAR FOR BIT DATA. Der Rückgabewert ist ein Bit-Wert vom Typ VARCHAR FOR BIT DATA
DECRYPT_BINARY	Wird verwendet für die Datentypen BLOB, BINARY oder VARBINARY. Der Rückgabewert ist ein Binärwert vom Typ BLOB oder VARBINARY
DECRYPT_CHAR	Wird verwendet für die Datentypen CHAR, VARCHAR FOR BIT DATA, BINARY oder VARBINARY. Die Rückgabe liefert einen Wert vom Typ VARCHAR. Ist der verschlüsselte Wert vom Typ BLOB, so wird ein Wert vom Typ CLOB zurückgegeben.

DECRYPT_DB	Für die Datentypen CHAR, VARCHAR FOR BIT DATA, BINARY oder VARBINARY erfolgt die Rückgabe als Typ VARGRAPHIC. Der Datentyp BLOB wird als Typ DBCLOB zurück gegeben.
GETHINT	Gibt den Kennworthinweis zurück, wenn die durch Funktion ENCRYPT_RC2 oder ENCRYPT_TDES verschlüsselten Daten einen enthalten. Ein Kennworthinweis ist ein Ausdruck, der als Gedächtnisstütze für einen Dateneigner dient, falls dieser sein Kennwort vergisst. Das Ergebnis der Funktion ist VARCHAR(32), wenn ein Hinweis vorhanden ist, oder Null, wenn die Funktion ENCRYPT keinen Kennworthinweis enthält.

Schauen wir uns abschließend hierzu einige Beispiele an:

ITPSQL.MITARBEITER - 172.16.110.22(Epsserver)

Datei Bearbeiten Ansicht Zeilen Hilfe

	PERSONALNUMMER	NACHNAME	ABTEILUNG	BESCHAEFTIGT_ALS	ANZJAHR	GEHALT	VORGESETZTER
	10	Sanders	20	Clerk	7	18357.50	40
	30	Marenghi	38	Clerk	5	17506.75	80
	60	Habers	15	Mgr	10	20659.80	80
	70	Norgreen	20	Clerk		13504.60	150
	90	Plözke	42	Mgr	7	18352.80	
	125	Nemecek	42	Clerk	6	10505.90	

Die Tabelle „Mitarbeiter“

Ausgangsbasis ist dieses Mal die Tabelle „Mitarbeiter“. In dieser Tabelle wollen wir zukünftig den Namen und das Gehalt des Mitarbeiters verschlüsseln. Zunächst müssen wir uns um das Datenformat der Spalten kümmern. Die verschlüsselten Werte müssen als Binärwerte gespeichert werden. Daher sind für die Spalten „Nachname“ und „Gehalt“ nur die Datentypen BINARY, VARBINARY, CHAR FOR BIT DATA, VARCHAR FOR BIT DATA oder BLOB möglich.

Unsere Spalten sind jedoch bislang folgendermaßen definiert:

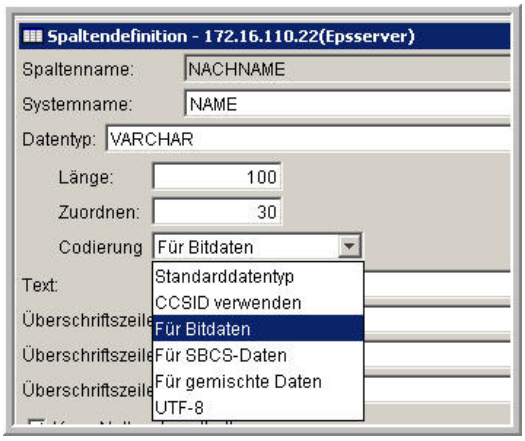
ITPSQL/MITARBEITER - 172.16.110.22(Epsserver)

Integritätsbedingungen über Fremdschlüssel | Prüfintegritätsbedingungen | Gespeicherte Ab

Tabelle | Spalten | Integritätsbedingungen über Schlüssel

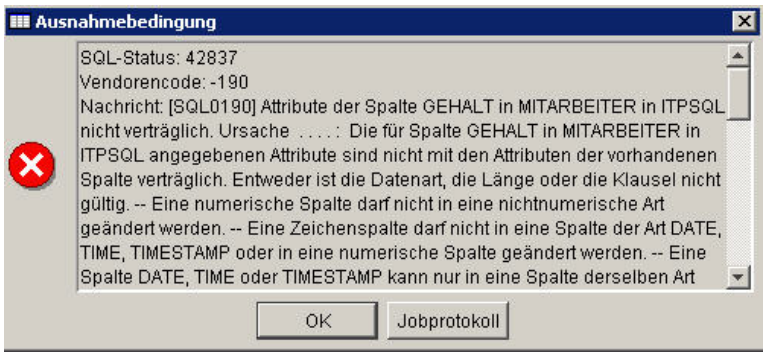
Spaltenname	Systemname	Datentyp	Län...	..	..	..	..	...	..	..	Zuordnen	Überschr...
PERSONALNUMMER	ID	SMALLINT		...		...						PERSONA...
NACHNAME	NAME	VARCHAR	100	...		...		6...			30	NACHNA...
ABTEILUNG	ABT	SMALLINT		...		...						ABTEILUN...
BESCHAEFTIGT_ALS	JOB	CHARACTER	10	...		...		2...				BESCHAE...
ANZJAHR	ANZJ	SMALLINT		...		...						ANZJAHR ...
GEHALT	GEHALT	DECIMAL	9,2	...		...						GEHALT ...

Die Spalte „Nachname“ lässt sich sehr einfach in den Datentyp VARCHAR FOR BIT DATA konvertieren:



Konvertierung der Spalte „Nachname“

Schwieriger wird es mit der Spalte „Gehalt“. Wenn Sie hier versuchen, den Datentyp auf BINARY oder CHARACTER FOR BIT DATA zu ändern, werden Sie folgende Fehlermeldung erhalten:



Datentypen nicht verträglich

Dabei ist es egal, ob Sie das grafische Interface oder die entsprechende SQL-Anweisung verwenden:

```
ALTER TABLE ITPSQL.Mitarbeiter
ALTER COLUMN Gehalt SET DATA TYPE CHAR(9) FOR BIT DATA
```

Eigentlich verwundert diese Fehlermeldung, da eine Typkonvertierung von DECIMAL zu CHARACTER erlaubt ist. Uns bleibt also nichts anderes übrig, als eine neue Spalte zu integrieren, die Daten zu kopieren, anschließend die alte Spalte zu löschen und abschließend die Originalspalte wiederherzustellen. Eine Umbenennung der kopierten Spalte kommt sicherlich nicht in Frage, da der Spaltenname nicht nachträglich verändert werden kann. Sollen also numerische Spalten nachträglich verschlüsselt werden, ist ein nicht unerheblicher Aufwand im Datenbankdesign erforderlich.

7.6.6
Seite 42

Jetzt müssen wir uns noch Gedanken über die Länge der Spalte machen. Die verschlüsselten Daten werden als Binärdaten gespeichert, beinhalten aber einen gewissen Overhead, da neben den eigentlichen Daten mindestens ein Passwort zur Verschlüsselung benötigt wird. Außerdem handelt es sich bei den Verfahren um Blockverschlüsselungsalgorithmen wie wir bereits wissen. Vor der Verschlüsselung werden dabei die Daten zunächst in gleich große Blöcke unterteilt. Die Blöcke werden anschließend jeder für sich verschlüsselt. Da die Verschlüsselungsroutinen ganze Blöcke voraussetzt, muss meistens der letzte Block mit Zufallsdaten aufgefüllt werden. Die jeweilige Blockgröße wird hierbei durch den Algorithmus vorgegeben. Schauen wir uns an, was das in unserem konkreten Beispiel bedeutet:

Als Passwort werde ich das Wort „Softwaremodernisierung“ verwenden. Das Passwort kann eine Länge zwischen 6 und 127 Byte haben. Grundsätzlich könnte zusätzlich ein Hinweis gespeichert werden, der helfen soll, sich an das Passwort zu erinnern. Solch ein Hinweiswort kann maximal 32 Byte lang sein. Darauf verzichte ich vorerst:

Für den Algorithmus TDES benötige ich für die Spalte Nachname:

Maximale Länge der Spalte z. B.	100 Byte
+ Passwort	127 Byte
+ Hinweis	0 Byte
+ Overhead für Verschlüsselung	24 Byte
	251Byte
+ auffüllen zum Block (8-Byte-Block)	5 Byte
Länge der verschlüsselten Daten	256 Byte

Würden wir BLOB-Spalten verschlüsseln oder einen Hinweis verwenden, erhöht sich die Byte-Anzahl entsprechend.

Ein ähnliches Ergebnis erhalten wir für den RC2-Standard:

Maximale Länge der Spalte z. B.	100 Byte
+ Passwort	127 Byte
+ Hinweis	0 Byte
+ Overhead für Verschlüsselung	16 Byte
	243Byte
+ auffüllen zum Block (8-Byte-Block)	5 Byte
Länge der verschlüsselten Daten	248 Byte

Und zu guter Letzt schauen wir uns die Berechnung auch noch für das AES-Verfahren an:

Maximale Länge der Spalte z. B.	100 Byte	9
+ Passwort	127 Byte	127
+ Hinweis	0 Byte	32
+ Overhead für Verschlüsselung	24 Byte	64
	251 Byte	192
+ auffüllen zum Block (16-Byte-Block)	5 Byte	0
Länge der verschlüsselten Daten	256 Byte	240

Für die Spalte „Gehalt“ muss ich entsprechende Berechnungen durchführen.

Für die Verschlüsselung der Daten verwende ich die Funktion ENCRYPT_AES, der drei Parameter übergeben werden:

```
ENCRYPT_AES (Quelle, Passwort, Hinweis)
```

Im ersten Parameter wird der zu verschlüsselnde Wert angegeben. Im zweiten Parameter muss das Passwort, über das die spätere Entschlüsselung zugelassen wird, hinterlegt werden. Bei dem Passwort muss es sich um einen alphanumerischen String mit einer Länge zwischen 6 und 127 Zeichen handeln. Im dritten Parameter kann ein bis zu 32 Zeichen langer alphanumerischer Wert angegeben werden, der dem Anwender hilft, sich an sein Passwort zu erinnern, sollte er es vergessen haben.

Das folgende Beispiel zeigt, wie die skalare Funktion ENCRYPT_AES verwendet werden kann:

```
INSERT INTO Itpsql.Mitarbeiter
VALUES(12,
ENCRYPT_AES('ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUV
WXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUV', 'Soft-
waremodernisierung'),
20, 'Mgr', 1, 150,
ENCRYPT_AES('123456,78', 'Softwaremodernisierungsbuch', 'ITP-
Verlag-2-Buch'));
```

7.6.6
Seite 44

Etwas schwieriger wird es, wenn Sie die bislang unverschlüsselten Werte der Tabelle nachträglich verschlüsseln wollen:

```
update Mitarbeiter A set
Gehalt = (SELECT ENCRYPT_AES(CHAR(Gehalt, 9))
         FROM Mitarbeiter B
         WHERE A.Id = B.Id),
Nachname = (SELECT ENCRYPT_AES(CHAR(Nachname, 100))
           FROM Mitarbeiter B
           WHERE A.id = B.id);
```

Aufgrund der Typkonvertierung der Spalten „Nachname“ und „Gehalt“ müssen Sie für das UPDATE die Daten zuerst als Character-Daten konvertieren. Tun Sie es nicht, werden Sie Probleme mit den Längenangaben der Spalten bekommen.

Schauen Sie sich das UPDATE-Beispiel noch einmal an. Sie sehen, dass ich weder einen Hinweis noch ein Passwort setze. Der Hinweis muss ohnehin nicht zwingend übergeben werden und auch auf die Nennung des Passwortes kann verzichtet werden, wenn Sie ein globales Passwort mit der Anweisung SET ENCRYPTION PASSWORD setzen.

Die SQL-Anweisung SET ENCRYPTION PASSWORD hinterlegt ein globales Passwort, das immer dann verwendet wird, wenn die Funktion ENCRYPT_AES selbst kein Passwort enthält. Schauen Sie sich das folgende RPG-Beispiel an. Einmalig wird am Programmanfang das Passwort gesetzt und fortan für alle Ver- und Entschlüsselungsanweisungen verwendet:

```
/FREE
EXEC SQL SET ENCRYPTION PASSWORD = :PASSWORT;

EXEC SQL
INSERT INTO Mitarbeiter
VALUES(12,

Encrypt_aes('ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUV
WXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUV'), 20,
'Mgr', 1, 150, encrypt_aes('123456,78');

IF SQLCODE < *ZEROS;
….;
….;
ENDIF;
```

Zusätzlich kann so auch der Hinweis global hinterlegt werden:

```
/FREE
EXEC SQL SET ENCRYPTION PASSWORD = :Passwort WITH HINT
:Hnweis;
```

Wenn Sie sich entschließen, Daten einiger Tabellen zu verschlüsseln, sollten Sie auch mit globalen Passwörtern arbeiten, denn was nutzt ein Passwort, das als Klartext in den SQL-Anweisungen steht?

Dieses Verfahren hilft nicht, wenn Sie sich Remote mit einer Datenbank verbinden. Das Passwort wird immer als lesbarer Text gesendet. Hier brauchen Sie daher weitere Verschlüsselungen wie bspw. IPSEC oder SSL.

Schauen wir uns jetzt das Ergebnis unserer Arbeit an:

PERSONALNUMMER	NACHNAME	AB...	BES...	A...	VO...	GEHALT
10	(□Z□□□Nõ¾/Ø&?Ú!□¾/Ø&?Ú!□X¿B¯c(Z'½¾□\□Ô;¶Ô□□...	20	Clerk	7	40	(G□□□□Ns¾/Ø&?Ú!□¾/Ø&?Ú!□ê¬ìÔÚ□□¤þVÙ□> □□□
30	(k> □□□Nõ¾/Ø&?Ú!□¾/Ø&?Ú!□þ>Ñ₩¼¯□³y □Ýz´Ð□j□...	38	Clerk	5	80	(Ô□□□□Ns¾/Ø&?Ú!□¾/Ø&?Ú!□&õøùe□□Úº?□¤Yh Í
60	(□W□□□Nõ¾/Ø&?Ú!□¾/Ø&?Ú!□3¦¾ì□□□M¦Ó¹ÐÈÅ!)3¾...	15	Mgr	10	80	(ú□□□□Ns¾/Ø&?Ú!□¾/Ø&?Ú!□_□ì?6□á□²s□□P#□'
70	(å¾□□□Nõ¾/Ø&?Ú!□¾/Ø&?Ú!□□□□Î□8L«□ü□k□ ®□...	20	Clerk	-	150	(¼ñ□□□Ns¾/Ø&?Ú!□¾/Ø&?Ú!□Õ□þã±ÐL õm□□8µ3Oå
90	(□□□□□Nõ¾/Ø&?Ú!□¾/Ø&?Ú!□"□{□Ôgi□½YÅ□ù□ì*È...	42	Mgr	7	-	(j?□□□Ns¾/Ø&?Ú!□¾/Ø&?Ú!□LU□ÚÚ9□-¯3]A#¢
125	(ºÅ□□□Nõ¾/Ø&?Ú!□¾/Ø&?Ú!□TP³□·½Ð□üT□!□?□□P,...	42	Clerk	6	-	(³□□□□Ns¾/Ø&?Ú!□¾/Ø&?Ú!□□¼Cê#p*□I□`A□Æ-
185	(Ïi□□□Nõ¾/Ø&?Ú!□¾/Ø&?Ú!□□z081U□×¨Xd□A7NÙåáDäÎ...	51	Mgr	6	-	(I□□□□Ns¾/Ø&?Ú!□¾/Ø&?Ú!□]]æfwî}¦â3Ù0i□□;

Die verschlüsselten Daten

Um die Daten wieder in Klarschrift anzuzeigen, benötigen Sie fortan eine der folgenden Funktionen:

DECRYPT_BIT: Entschlüsseln und Ausgeben als Bit-Daten,
DECRYPT_CHAR: Entschlüsseln und Ausgeben als Single-Byte alphanumerische Werte,
DECRYPT_BINARY: Entschlüsseln und Ausgeben als Binär-Werte,
DECRYPT_DB: Entschlüsseln und Ausgeben als Double Byte alphanumerische Werte.

Diese skalaren Funktionen stehen bereits seit Release V5R3 zur Verfügung und werden unabhängig vom verwendeten Verschlüsselungsalgorithmus zum Entschlüsseln der Daten verwendet.

```
SELECT Id, DECRYPT_BIT(Gehalt) as BitDaten ,
DECRYPT_CHAR(Gehalt) as Zeichendaten,
DECRYPT_BINARY(Gehalt) as Binaerwerte
FROM ItpSql.Mitarbeiter ;
```

Ergänzung 19/2011

7.6.6
Seite 46

ID	BITDATEN	ZEICHENDATEN	BINAERWERTE
125	10505,90	10505,90	F1F0F5F0F56BF9F040
185	21150,00	21150,00	F2F1F1F5F06BF0F040
160	21000,00	21000,00	F2F1F0F0F06BF0F040
175	12258,50	12258,50	F1F2F2F5F86BF5F040
255	12009,75	12009,75	F1F2F0F0F96BF7F540
195	14252,75	14252,75	F1F4F2F5F26BF7F540
205	11508,60	11508,60	F1F1F5F0F86BF6F040

Daten müssen lesbar gemacht werden

Nun wollen wir aber auch noch herausfinden, was es mit dem Hinweiswort auf sich hat. Gehen wir davon aus, dass wir unser Passwort vergessen haben. Die Funktion GETHINT liefert uns für diesen Fall den Kennworthinweis:

```
SELECT ID, GETHINT(Gehalt) FROM Mitarbeiter
```

ID	00002
30	-
24	ITP_BUCH-2
23	ITP_BUCH-2

Wirkung der Funktion GETHINT

7.6.6.8 ROW_NUMBER(), RANK() und DENSE_RANK() – Release V5R4

Mit den Funktionen ROW_NUMBER, RANK() und DENSE_RANK() können Sie eine Liste von Werten aus Ihrem Datenbestand mit einer laufenden Nummerierung oder in einer Rangfolge darstellen. Die Funktionen ROW_NUMBER und RANK() gehören zu den Online Analytical Processing Functions (OLAP) und können daher nie als Argumente einer Aggregatfunktion verwendet werden. Aus diesem Grund könnten Sie Klauseln wie SUM(RANK()……) nicht nutzen. Und es gibt noch weitere Einschränkungen für OLAP-Funktionen.

Die OLAP-Spezifikationen sind nicht erlaubt in Abfragen

- die korrelierenden SubSelects enthalten,
- die die CCSID konvertieren,
- in Kombination mit den Funktionen LOWER, TRANSLATE, UPPER oder SUBSTRING,
- die mit verteilten Tabellen arbeiten,
- in Kombination mit benutzerdefinierten Table-Funktionen (UDTF),
- die Tabellen verarbeiten, die mit einem Read Trigger verbunden sind.

Eingesetzt werden die Funktionen vorrangig in der Spaltenauswahl oder in einer ORDER BY-Klausel. Schauen wir uns die Wirkungsweise an einem Beispiel an:

Über die Funktion ROW_NUMBER() kann das Ergebnis einer SQL-Abfrage mit einem laufenden Zähler versehen werden. Die Funktion wird ohne Parameter aufgerufen und darf nur in der SELECT-Liste, also als neue Spalte, oder in den ORDER-BY-Anweisungen angegeben werden. Die Verwendung dieser Funktion in den WHERE-Bedingungen ist nicht zulässig. Das Ergebnis aus dieser Funktion wird immer als numerischer Wert mit dem Datentyp BIGINT ausgegeben und kann niemals NULL sein.

Das folgende Beispiel zeigt die einfachste Verwendung der Funktion ROW_NUMBER() in einem Select-Statement.

```
SELECT
ROW_NUMBER() OVER(ORDER BY Kundennummer),
Kundennummer,
SUM(Auftragswert)
FROM Auftrag
GROUP BY Kundennummer;
```

Zunächst ermitteln wir die Auftragssumme je Kunde anhand unserer Tabelle AUFTRAG und nummerieren die Tabellenzeilen aufsteigend nach der Kundennummer. Sinnvoller wäre es vermutlich, die Zeilen absteigend nach dem Auftragswert zu nummerieren. Natürlich ist auch das problemlos möglich:

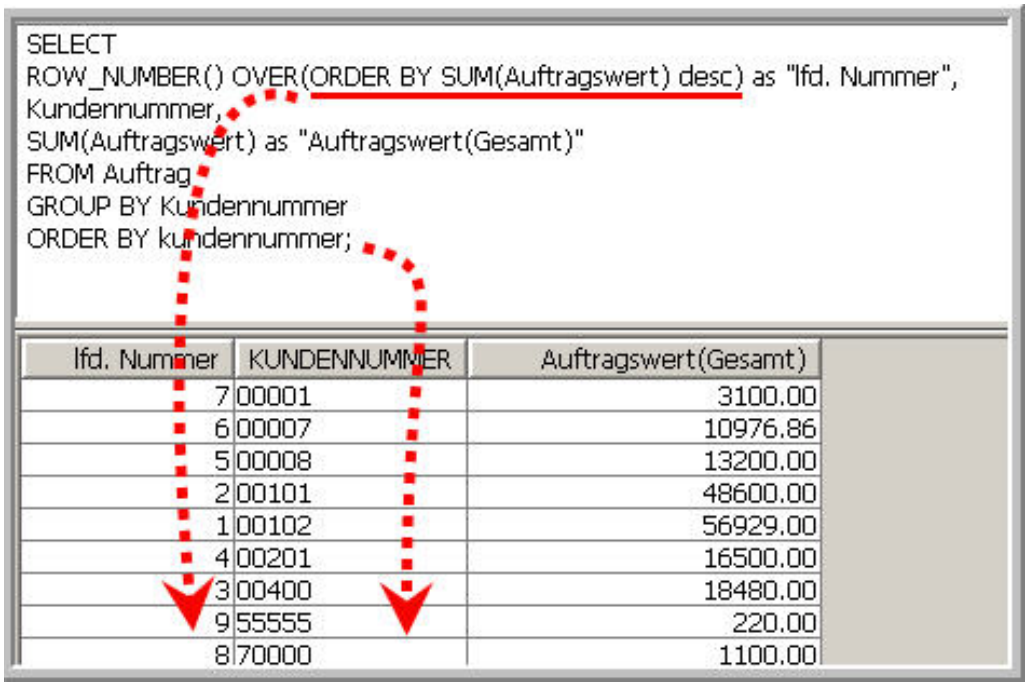
```
SELECT
ROW_NUMBER() OVER(ORDER BY SUM(Auftragswert) DESC),
Kundennummer,
SUM(Auftragswert)
FROM Auftrag
GROUP BY Kundennummer;
```

Und so sieht das Ergebnis aus:

lfd. Nummer	KUNDENNUMMER	Auftragswert(Gesamt)
1	00102	56929.00
2	00101	48600.00
3	00400	18480.00
4	00201	16500.00
5	00008	13200.00
6	00007	10976.86
7	00001	3100.00
8	70000	1100.00
9	55555	220.00

Wirkung der Funktion ROW_NUMBER

Die Nummerierung erfolgt in der Reihenfolge der ausgegebenen Zeilen, d. h., die erste Zeile erhält die laufende Nummer 1, die nächste Zeile die Nummer 2 usw. Nur wenn im SQL-Statement selbst keine ORDER BY-Anweisung verwendet wird, erfolgt die Sortierung der Daten abhängig vom ausgewählten Zugriffspfad der Funktion ROW_NUMBER(). Entscheiden Sie sich hingegen für eine eigenständige ORDER BY-Klausel im SQL-Statement, so bestimmt die ORDER BY-Klausel die Sortierung des Result Sets, und die ORDER BY-Klausel der ROW_NUMBER-Funktion bestimmt die Datenreihenfolge für die einzufügende Nummerierung.



lfd. Nummer	KUNDENNUMMER	Auftragswert(Gesamt)
7	00001	3100.00
6	00007	10976.86
5	00008	13200.00
2	00101	48600.00
1	00102	56929.00
4	00201	16500.00
3	00400	18480.00
9	55555	220.00
8	70000	1100.00

Zugriffspfad der Nummerierung

Die Funktion ROW_NUMBER() liefert eine laufende Nr., d.h., auch dann, wenn die Ergebniszeilen identisch sind, wird eine unterschiedliche Nummer ausgegeben.

lfd. Nummer	KUNDENNUMMER	Auftragswert(Gesamt)
1	00102	56629.00
2	00101	47050.00
3	00400	18480.00
4	00201	16500.00
5	00008	13276.00
6	00007	10976.86
7	00109	10976.00
8	00001	3100.00
9	70000	1100.00
10	55555	1100.00

Identische Werte erhalten unterschiedliche Nummerierungen

Vielfach ist es jedoch notwendig, eine Rangfolge zu bilden, d.h., sofern die wesentlichen Informationen für mehrere Zeilen identisch sind, muss die gleiche Nummer vergeben werden. Hier können die Funktionen RANK() und DENSE_RANK() eingesetzt werden. Nehmen wir das eben verwendete Beispiel wieder als Ausgangsbasis.

7.6.6
Seite 50

Wie auch bei der Verwendung der Funktion ROW_NUMBER() ist das Ergebnis aus den Funktionen RANK() und DENSE_RANK() ein numerischer Wert, der im Format BIGINT ausgegeben wird. Beide Funktionen werden ohne Parameter aufgerufen und können nur in der Spaltenliste oder der ORDER BY-Klausel eines Select-Statements genutzt werden.

```
SELECT
RANK() OVER(ORDER BY SUM(Auftragswert) DESC) AS "lfd. Nummer",
Kundennummer,
SUM(Auftragswert) AS "Auftragswert(Gesamt)"
FROM Auftrag
GROUP BY Kundennummer
```

Die Funktion ähnelt der Funktion ROW_NUMBER(). Wieder wird durch die Klausel OVER(ORDER BY …) die Spalte benannt, über die die laufende Nummer gebildet werden soll. Die auf- und absteigende Sortierung kann wie in einer ORDER BY-Klausel über die Schlüsselwörter ASC und DESC bestimmt werden.

Die folgende Abbildung zeigt die Unterschiede in der Wirkung:

Nummer	KUNDENNUMMER	Auftragswert(Gesamt)
1	00102	56629.00
2	00101	47050.00
3	00400	18480.00
4	00201	16500.00
5	00008	13276.00
6	00007	10976.86
6	00109	10976.86
8	00001	3100.00
9	70000	1100.00
9	55555	1100.00

Wirkung der Funktion RANK()

Anders als bei der ROW_NUMBER-Funktion erhalten jetzt identische Werte identische Nummerierungen. Allerdings ergeben sich bei Verwendung der skalaren Funktion RANK() lückenhafte Nummerierungen. Sie sehen in der oberen Abbildung, dass die Nummer 6 zweimal genutzt wurde und daher die Nummer 7 in der Nummerierung fehlt. Manchmal ist es jedoch erforderlich, solche Lücken zu vermeiden. In diesen Fällen verwenden Sie einfach die Funktion DENSE_RANK(). Dabei ist es egal, ob Sie sich für die Schreibweise DENSE_RANK() oder DENSERANK() entscheiden.

lfd. Nummer	KUNDENNUMMER	Auftragswert(Gesamt)
1	00102	56629.00
2	00101	47050.00
3	00400	18480.00
4	00201	16500.00
5	00008	13276.00
6	00007	10976.86
6	00109	10976.86
7	00001	3100.00
8	70000	1100.00
8	55555	1100.00

Wirkung der Funktion DENSE_RANK()

Solange in den Spalten, über die sortiert werden soll, keine NULL-Werte enthalten sind, ist die Reihenfolge klar. Die Frage ist nun, wo werden NULL-Werte, die bekanntlich außerhalb des gültigen Bereichs liegen, eingeordnet? Am Anfang oder am Ende? Man kann es sich aussuchen! Für jede einzelne Spalte, die in der OVER(ORDER BY ...)-Klausel aufgelistet wird, kann man festlegen, wo die NULL-Werte eingeordnet werden sollen. Bei der Angabe von NULLS FIRST werden die Null-Werte vor dem gültigen Bereich eingeordnet und bei NULLS LAST am Ende. Dabei spielt es keine Rolle, ob auf- oder absteigend sortiert wird. Der Standardwert ist NULLS FIRST.

```
SELECT
DENSE_RANK() OVER(ORDER BY SUM(Auftragswert) DESC NULLS LAST )
AS "lfd. Nummer",
Kundennummer,
SUM(Auftragswert) AS "Auftragswert(Gesamt)"
FROM Auftrag
GROUP BY Kundennummer;
```

7.6.6
Seite 52

Bislang haben wir immer das gesamte Ergebnis nummeriert. Vielleicht ist es aber erforderlich, die Nummerierung innerhalb einer Gruppe zu bilden? Um das zu erreichen, kann man innerhalb der OVER(ORDER BY...)-Anweisung eine PARTITION BY-Klausel einfügen. Schauen wir uns an, was gemeint ist, indem wir die Aufträge je Kunde nummerieren:

```
SELECT
DENSE_RANK()
OVER(PARTITION BY Kundennummer ORDER BY Auftragswert ASC) ,
Kundennummer,
SUM(Auftragswert) AS "Auftragswert(Gesamt)"
FROM Auftrag
GROUP BY Kundennummer, Auftragswert
```

Jetzt wird das Ranking für jeden Kunden erstellt:

lfd. Nummer	KUNDENNUMMER	Auftragswert(Gesamt)
1	00001	0.00
2	00001	3100.00
1	00007	150.00
2	00007	10826.86
1	00008	12000.00
2	00008	1276.00
1	00101	16500.00
2	00101	12400.00
3	00101	18150.00

Nummerierung innerhalb einer Gruppe

7.6.6.9 Mehrdimensionale Gruppierungen – Release i6.1

Wer je mit Query/400 gearbeitet hat, weiß, wie einfach Zwischensummen nach einem Gruppenwechsel gebildet werden können. Beim Wechsel auf SQL mussten viele feststellen, dass eine vergleichbare Funktionalität in SQL nicht vorhanden war und häufig war das der Grund dafür, weiter mit QUERY/400 zu arbeiten, anstatt mühevoll mit mehreren Sub-Selects und diversen UNION-Anweisungen die Zwischensummen „zu Fuß“ zu bilden.

Mit Release i6.1 erweitert IBM endlich die GROUP BY-Klausel. Jetzt können auch mit SQL beliebige Zwischensummen gebildet werden.

Lassen Sie uns die Neuerungen wieder an einem Beispiel erklären. Wir benötigen eine Auswertung der Kundenumsätze der Jahre 2009 bis 2011.

Umsatzjahr	KUNDENNUMMER	Auftragswert/Jahr	
2009	00007	9992.60	
2009	00008	12076.00	
2009	00101	26850.00	
2009	00102	28251.00	
2009	00109	6426.00	
2009	00201	15000.00	
2009	00400	16800.00	
2009	55555	1060.00	
2009	70000	900.00	Zwischenergebnis
2010	00101	4800.00	
2010	00102	2339.00	
2010	00400	1680.00	Zwischenergebnis
			Gesamtwert:

Die Aufgabenstellung

7.6.6
Seite 54

Vor Release 6.1 konnten die geforderten Zwischensummen nur mit Hilfe verschiedener Sub-Selects gebildet werden, die über UNION-Anweisungen verbunden wurden:

```
SELECT CHAR(YEAR(Auftragsdatum)),
Kundennummer, SUM(Auftragswert)
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2009' AND '31.12.2009'
GROUP BY CHAR(YEAR(Auftragsdatum)), Kundennummer

UNION ALL
-- ZwischenSUMme 2009
SELECT CHAR(' ') , 'Gesamt 2009', SUM(Auftragswert)
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2009' AND '31.12.2009'

UNION ALL
SELECT CHAR(YEAR(Auftragsdatum)) ,
Kundennummer, SUM(auftragswert) AS "Auftragswert/Jahr"
FROM Auftrag
WHERE Auftragsdatum BETWEE '01.01.2010' AND '31.12.2010'
GROUP BY CHAR(YEAR(Auftragsdatum)), Kundennummer

UNION ALL
-- ZwischenSUMme 2010
SELECT CHAR(' ') , 'Gesamt 2010', SUM(auftragswert) AS
"Auftragswert/Jahr"
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2010' AND '31.12.2010'

UNION ALL
SELECT CHAR(YEAR(Auftragsdatum)) AS "Umsatzjahr",
Kundennummer, SUM(auftragswert) AS "Auftragswert/Jahr"
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2011' AND '31.12.2011'
GROUP BY CHAR(YEAR(Auftragsdatum)), Kundennummer

UNION ALL
-- ZwischenSUMme 2011
SELECT CHAR(' ') , 'Gesamt 2011', SUM(auftragswert) AS
"Auftragswert/Jahr"
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2011' AND '31.12.2011'
```

```
UNION ALL
-- GesamtSUMme

SELECT CHAR(' ') , 'Gesamt 2009 - 2011', SUM(auftragswert) AS
"Auftragswert/Jahr"
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2009' AND '31.12.2011';
```

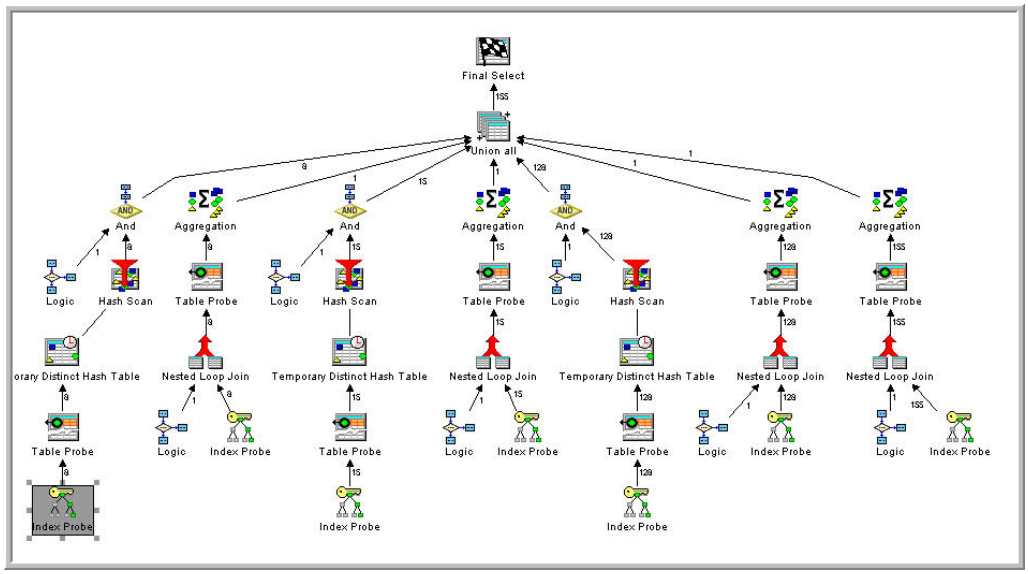
Häufig musste die Darstellung mit Tricks aufbereitet werden. Ich muss beispielsweise die erste Spalte als CHAR-Spalte konvertieren, um den entsprechenden Text darstellen zu können:

Umsatzjahr	KUNDENNUMMER	Auftragswert/Jahr
2009	00400	16800.00
2009	00007	9992.60
2009	55555	1060.00
2009	00201	15000.00
	Gesamt 2009	117355.60
2010	00400	1680.00
2010	70000	100.00
2010	00102	2339.00
2010	00101	4800.00
	Gesamt 2010	8919.00
2011	00201	1500.00
2011	00008	1200.00
2011	06000	-
2011	55555	40.00
2011	00001	3100.00
2011	00109	4550.86
2011	00007	984.26
2011	70000	100.00
2011	00102	26039.00
2011	00101	15400.00
	Gesamt 2011	52914.12
	Gesamt 2009 - 2011	179188.72

Zwischensummen und Gesamtsummen bis V6R1

Sieht man von dem hohen Aufwand einmal ab, ist das Ergebnis ganz passabel. Doch was ist, wenn ich die Zwischensummen je Kunde bilden möchte? Der Aufwand wäre nicht mehr zu bewältigen. Außerdem würden Sie an die Grenzen der UNION-Anweisung stoßen. Sie können nur 31 UNION-Anweisungen in einem SQL-Statement codieren, und Sie haben doch sicherlich mehr als 30 Kunden?

Derartige Abfragen können daher sehr schnell sehr komplex werden und aufgrund der vielen Sub-Selects zwingen Sie SQL dazu, die gleiche Tabelle mehrfach – nämlich für jedes Sub-Select zu lesen. Ihre Fleißarbeit wird also auch noch mit schlechter Performance belohnt. Sie glauben mir nicht? Schauen Sie sich doch das zugehörige Visual Explain-Diagramm an:



Jedes SUB-Select wird separat gelesen

Um Zwischensummen zu vereinfachen und v. a. die Performance zu verbessern, können ab Release i6.1 die Funktionen ROLLUP, CUBE und GROUPING SETS innerhalb der GROUP BY-Anweisung verwendet werden.

Die Funktion ROLLUP bildet Zwischensummen in hierarchischer Reihenfolge:

```
SELECT YEAR(Auftragsdatum) AS "Umsatzjahr",
Kundennummer, SUM(auftragswert) AS "Auftragswert/Jahr"
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2009' AND '31.12.2011'
GROUP BY ROLLUP( YEAR(Auftragsdatum), Kundennummer)
ORDER BY YEAR(Auftragsdatum), Kundennummer;
```

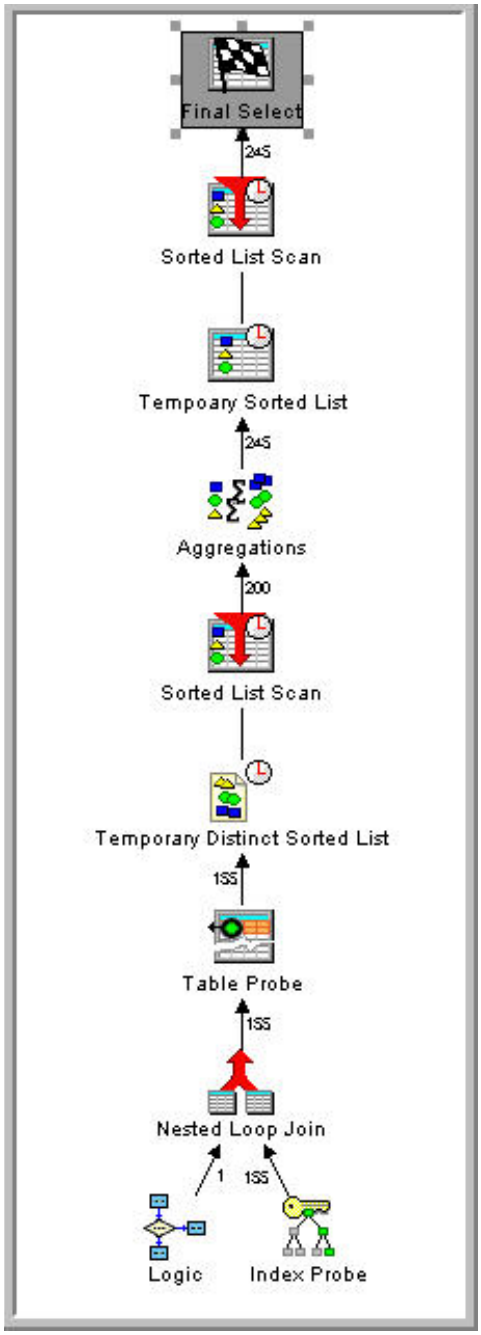
Sie sehen an der Abfrage, dass die Funktion ROLLUP innerhalb der GROUP BY-Klausel verwendet wird. Deshalb erhalten wir auch bei dem Beispiel Zwischenergebnisse für den Jahresumsatz. Am Ende wird das Gesamtergebnis ausgegeben. Die Abfrage liefert somit das gleiche Ergebnis wie das ursprüngliche SQL-Statement.

Umsatzjahr	KUNDENNUMMER	Auftragswert/Jahr
2009	00109	6426.00
2009	00201	15000.00
2009	00400	16800.00
2009	55555	1060.00
2009	70000	900.00
2009	-	117355.60
2010	00101	4800.00
2010	00102	2339.00
2010	00400	1680.00
2010	70000	100.00
2010	-	8919.00
2011	00001	3100.00
2011	00007	984.26
2011	00008	1200.00
2011	00101	15400.00
2011	00102	26039.00
2011	00109	4550.86
2011	00201	1500.00
2011	06000	-
2011	55555	40.00
2011	70000	100.00
2011	-	52914.12
-	-	179188.72

Wirkungsweise der Funktion ROLLUP

Es entfällt aber nicht nur viel lästige Schreibarbeit, sondern auch die umständliche Verarbeitung des SQL-Statements gehört der Vergangenheit an, wie nachfolgendes Visual Explain-Diagramm zeigt:

7.6.6
Seite 58



Verarbeitung der Funktion ROLLUP

Betrachtet man die Visual Explain-Auswertung, stellt man fest, dass auf die Tabelle nur ein einziges Mal zugegriffen wird. Die ausgewählten Zeilen werden zunächst in einer temporären, eindeutig sortierten Liste gespeichert, so dass dadurch die Kundenumsätze verdichtet werden. Eine zweite temporäre Liste hilft anschließend die Zwischen- und Gesamtsummen zu bilden. Vergleichen Sie die beiden gezeigten Visual Explain-Diagramme, können Sie sich sicherlich vorstellen, dass die Plattenzugriffe in der zweiten Abfrage deutlich reduziert werden, so dass die Gesamtlaufzeit davon natürlich entsprechend profitieren wird.

Für derartige Abfragen gibt es „Spielvarianten“. So können Sie anstelle der Funktion ROLLUP auch die Funktion CUBE verwenden.

```
SELECT YEAR(Auftragsdatum) AS "Umsatzjahr",
Kundennummer, SUM(auftragswert) AS "Auftragswert/Jahr"
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2009' AND '31.12.2011'
GROUP BY CUBE( YEAR(Auftragsdatum), Kundennummer)
ORDER BY YEAR(Auftragsdatum), Kundennummer;
```

Anders als bei der Funktion ROLLUP bildet die Funktion CUBE für jede genannte Spalte Zwischensummen. Wir erhalten also die Umsätze aufgelaufen für ein Jahr, aber auch die Umsätze aufgelaufen für jeden Kunden. Und natürlich wird auch hier abschließend die Gesamtsumme dargestellt:

Umsatzjahr	KUNDENNUMMER	Auftragswert/Jahr
2009	00400	16800.00
2009	55555	1060.00
2009	70000	900.00
2009	-	117355.60
2010	00101	4800.00
2010	00102	2339.00
2010	00400	1680.00
2010	70000	100.00
2010	-	8919.00
2011	00001	3100.00
2011	00007	984.26
2011	00008	1200.00
2011	00101	15400.00
2011	00102	26039.00
2011	00109	4550.86
2011	00201	1500.00
2011	06000	-
2011	55555	40.00
2011	70000	100.00
2011	-	52914.12
-	00001	3100.00
-	00007	10976.86
-	00008	13276.00
-	00101	47050.00
-	00102	56629.00
-	00109	10976.86
-	00201	16500.00
-	00400	18480.00
-	06000	-
-	55555	1100.00
-	70000	1100.00
-	-	179188.72

Wirkungsweise der Funktion CUBE

Betrachtet man auch für die Funktion CUBE das entsprechende Visual Explain-Diagramm, so wird man feststellen, dass auch jetzt – trotz mehrfacher Summenbildung – die Tabelle lediglich einmal gelesen wird. Ich möchte allerdings an dieser Stelle auf die Darstellung des Diagramms verzichten.

Bei den Funktionen ROLLUP und CUBE haben Sie keinen Einfluss auf die Zwischensummen. Es gibt jedoch Situationen, in denen Sie bestimmen wollen, welche Zwischensummen gebildet werden sollen und welche nicht. In diesen Fällen nutzen Sie einfach die Funktion GROUPING SETS.

Bei der Funktion GROUPING SETS können Sie die Spalten, für die Zwischensummen erforderlich sind, direkt in der Funktion benennen.

```
SELECT YEAR(Auftragsdatum) AS "Umsatzjahr",
Kundennummer, SUM(auftragswert) AS "Auftragswert/Jahr"
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2009' AND '31.12.2011'
AND ( Kundennummer = '00101' OR Kundennummer = '00102')
GROUP BY GROUPING SETS
(
(Kundennummer) ,
(YEAR(Auftragsdatum), Kundennummer ) ,
()
)
ORDER BY Kundennummer;
```

An dem Beispiel können Sie sehen, dass die Spalten, anhand derer Zwischensummen gebildet werden sollen, in Klammern eingeschlossen werden. Jede geforderte Gruppierung wiederum muss durch Komma getrennt werden. Soll abschließend die Gesamtsumme ausgegeben werden, wird einfach eine leere runde Klammer () angegeben.

Ich fordere mit meiner Abfrage somit Gruppensummen je Kunde an und zusätzlich die aufgelaufenen Kundenumsätze je Kalenderjahr. Abschließend möchte ich den Gesamtumsatz sehen.

SELECT YEAR(Auftragsdatum) AS "Umsatzjahr", ...

Umsatzjahr	KUNDENNUMMER	Auftragswert/Jahr
-	00101	47050.00
2011	00101	15400.00
2010	00101	4800.00
2009	00101	26850.00
-	00102	56629.00
2011	00102	26039.00
2010	00102	2339.00
2009	00102	28251.00
-	-	103679.00

ORDER BY Kundennummer

SELECT YEAR(Auftragsdatum) AS "Umsat...

Umsatzjahr	KUNDENNUMMER	Auftragswert/Jahr
2009	00102	28251.00
2009	00101	26850.00
2010	00102	2339.00
2010	00101	4800.00
2011	00102	26039.00
2011	00101	15400.00
-	00101	47050.00
-	00102	56629.00
-	-	103679.00

ORDER BY YEAR(Auftragsdatum)

Wirkungsweise der Funktion GROUPING SETS

Wie gefordert, sehe ich die gewünschten Gruppensummen. Beachten sollten Sie hierbei aber auch die ORDER BY-Klausel Ihres SQL-Statements – nicht die Funktion GROUPING SETS bestimmt die Darstellung der Ergebnisse, sondern die ORDER BY-Klausel in Ihrem SQL-Statement.

Betrachtet man die neuen Funktionen, so stellt man fest, dass die Funktionen ROLLUP() und CUBE() Vereinfachungen der GROUPING SET()-Schreibweise sind. Im Hinblick auf die Performance macht es keinen Unterschied, ob Sie mit GROUPING SETS() oder mit ROLLUP() bzw. CUBE() arbeiten. Alle Funktionen sind gleichermaßen performant.

Allerdings treten manchmal „Fehler“ auf, wenn Sie die GROUPING SETS „überfrachten“. Schauen Sie was passiert, wenn ich mein Beispiel erweitere:

```
SELECT YEAR(Auftragsdatum) AS "Umsatzjahr",
Kundennummer, SUM(auftragswert) AS "Auftragswert/Jahr"
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2009' AND '31.12.2011'
AND ( Kundennummer = '00101' OR Kundennummer = '00102')
GROUP BY GROUPING SETS
( (Kundennummer) ,
(YEAR(Auftragsdatum)),
(YEAR(Auftragsdatum), Kundennummer ) ,
()
)
ORDER BY Kundennummer, YEAR(Auftragsdatum);
```

7.6.6
Seite 62

Ich habe die Funktion GROUPING SETS lediglich um die Zwischensumme YEAR(Auftragsdatum) erweitert und erhalte daraufhin das folgende Ergebnis:

SELECT YEAR(Auftragsdatum) AS "Umsatzjahr",...

Umsatzjahr	KUNDENNUMMER	Auftragswert/Jahr
2009	00101	26850.00
2010	00101	4800.00
2011	00101	15400.00
-	00101	47050.00
2009	00102	28251.00
2010	00102	2339.00
2011	00102	26039.00
-	00102	56629.00
-	- Gesamtwert	103679.00
-	- Summe für 2009	55101.00
-	- Summe für 2010	7139.00
-	- Summe für 2011	41439.00

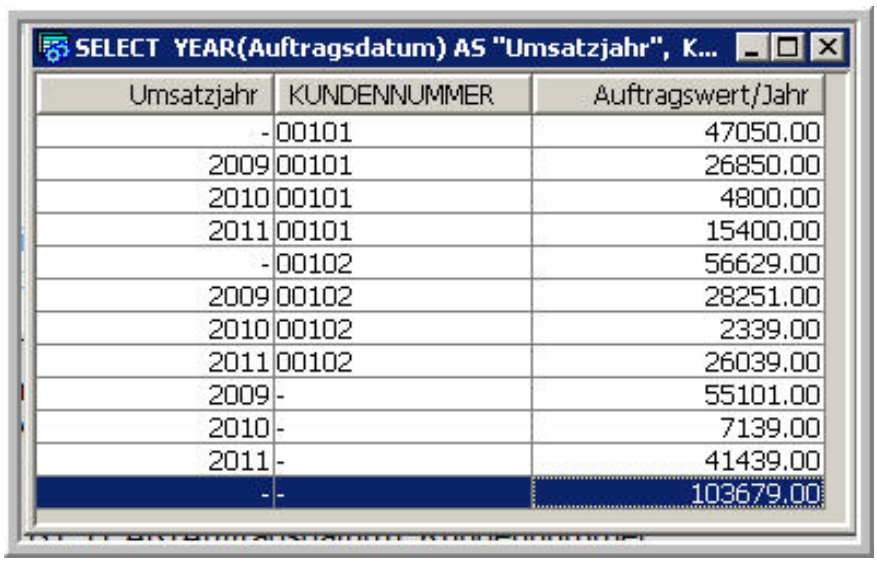
Wenn man zu viel will!

Zwar werden die Summen korrekt gebildet; doch kann man anhand des Ergebnisses teilweise nicht mehr feststellen, um welche Informationen es sich handelt.

Für diese Situationen müssen Sie die Funktionen GROUPING SETS und ROLLUP bzw. CUBE einfach kombinieren:

```
SELECT YEAR(Auftragsdatum) AS "Umsatzjahr",
Kundennummer, SUM(auftragswert) AS "Auftragswert/Jahr"
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2009' AND '31.12.2011'
GROUP BY CUBE( YEAR(Auftragsdatum), Kundennummer)
ORDER BY YEAR(Auftragsdatum), Kundennummer;

SELECT YEAR(Auftragsdatum) AS "Umsatzjahr",
Kundennummer, SUM(auftragswert) AS "Auftragswert/Jahr"
FROM Auftrag
WHERE Auftragsdatum BETWEEN '01.01.2009' AND '31.12.2011'
AND ( Kundennummer = '00101' OR Kundennummer = '00102')

GROUP BY GROUPING SETS
( (Kundennummer) ,
ROLLUP(YEAR(Auftragsdatum), Kundennummer )
)
ORDER BY Kundennummer;
```

Die Funktion GROUPING SETS bestimmt, dass für die Kundennummer aufgelaufene Ergebnisse ermittelt werden. Und die Funktion ROLLUP sorgt dafür, dass einerseits die Jahresumsätze je Kunde, andererseits die Summen je Auftragsjahr korrekt ausgewiesen werden.



Umsatzjahr	KUNDENNUMMER	Auftragswert/Jahr
-	00101	47050.00
2009	00101	26850.00
2010	00101	4800.00
2011	00101	15400.00
-	00102	56629.00
2009	00102	28251.00
2010	00102	2339.00
2011	00102	26039.00
2009	-	55101.00
2010	-	7139.00
2011	-	41439.00
-	-	103679.00

Kombination von GROUPING SETS und der Funktion ROLLUP

Ich denke, die Beispiele zeigen deutlich, dass die neuen Funktionen in vielen Auswertungen sinnvoll eingesetzt werden können und helfen werden, schnellere und einfachere Lösungen zu erstellen. Doch die Funktionen sind auch sehr mächtig, insbesondere dann, wenn Sie sie kombinieren.

7.6.6
Seite 64

7.6.6.10 Skalare Funktionen mit regulären Ausdrücken

Seit Release i7.1 TR9 gibt es eine neue Variante skalarer Funktionen: Diese Funktionen beinhalten als Parameter einen regulären Ausdruck, der nach den gleichen Regeln aufgebaut wird wie für das Prädikat REGEXP_LIKE. Lesen Sie das Kapitel 7.6.1.2.1, wenn Sie mit dem Aufbau der regulären Ausdrücke nicht vertraut sind.

In diesem Kapitel soll es darum gehen, Telefonnummern zu korrigieren. Bisher wurden in der Kundentabelle alle Telefonnummern mit einfacher Vorwahl anstatt mit Ländervorwahl erfasst. Diese Eingabe wollen wir jetzt korrigieren, das heißt, aus der Rufnummer 0421555589 soll die Rufnummer +49 421555589 werden.

Dazu können wir die Funktion REGEXP_REPLACE verwenden. Sie ersetzt die gefundenen Vorkommen der Vergleichsmuster mit der Ersatzzeichenkette.

```
SELECT
CUSTOMER_TELEPHONE,
TRIM(REGEXP_REPLACE(CUSTOMER_TELEPHONE, '^([0])', '+49 '))
AS "KORRIGIERTE TELEFONNUMMER"
FROM CUSTOMER;
```

Der erste Parameter bestimmt die Tabellenspalte. Im zweiten Parameter finden wir die gesuchte Zeichenkette – in unserem Beispiel die führende Null. Der dritte Parameter legt die Ersatzzeichenkette fest.

CUSTOMER_TELE...	korrigierte Telefonnummer
04432 6465	+49 4432 6465
+57 34334-23	+57 34334-23
+43 555300	+43 555300
+43 55 34-1	+43 55 34-1
02223455	+49 2223455
+34 554 413-44	+34 554 413-44
09823 880	+49 9823 880
05075555	+49 5075555
0802390	+49 802390
05111053	+49 5111053
0654999	+49 654999

Ergebnis der Telefonnummernkorrektur – Teil 1

7.6.6
Seite 66

Schaut man sich die Abbildung an, stellt man schnell fest, dass die Korrektur richtig ist. Es fällt aber auch auf, dass innerhalb der Telefonnummern noch Leerzeichen vorhanden sind, die aber unerwünscht sind. Schauen wir mal, ob wir auch dieses Problem mit der Funktion lösen können:

```
SELECT CUSTOMER_TELEPHONE, REGEXP_REPLACE(TRIM(CUSTOMER_TELEPHONE), '\S', '', 1, 2)
AS "KORRIGIERTE NUMMER"
 FROM CUSTOMER ;
```

Für diese Anforderung benötigen wir auch den dritten und vierten Parameter der Funktion. Mit dem dritten Parameter bestimmen wir die Startposition für die Suche. Der vierte Parameter sorgt für das Vorkommen. Wir suchen also nach einem Leerzeichen (\s) und entfernen dieses Leerzeichen (‘‘). Dabei soll ab dem ersten Zeichen innerhalb der Spalte gesucht werden, aber erst ab dem zweiten Vorkommen ersetzt werden. Das Ergebnis ist wieder in Ordnung:

CUSTOMER_TELE...	korrigierte Nummer
04432 6465	04432 6465
+57 34334-23	+57 34334-23
+43 555300	+43 555300
+43 55 34-1	+43 5534-1
02223455	02223455
+34 554 413-44	+34 554413-44
09823 880	09823 880
050 5 55	050 555
0802390	0802390
05111053	05111053
0654999	0654999

Ergebnis der Telefonnummernkorrektur – Teil 2

Falls Sie sich fragen, ob auch beide Korrekturen gleichzeitig durchgeführt werden können, lautet die Antwort: Ja. Auch die skalaren Funktionen mit regulären Ausdrücken können wir verschachteln. In unserem Beispiel sieht die Lösung dann wie folgt aus:

```
SELECT CUSTOMER_TELEPHONE, TRIM(REGEXP_REPLACE(
(REGEXP_REPLACE(TRIM(CUSTOMER_TELEPHONE), '\S', '', 1, 2, 'I')
), '^([0])', '+49 ')) AS „KORRIGIERTE NUMMER“
FROM CUSTOMER ;
```

CUSTOMER_TELE...	korrigierte Nummer
04432 6465	+49 4432 6465
+57 34334-23	+57 34334-23
+43 555300	+43 555300
+43 55 34-1	+43 5534-1
02223455	+49 2223455
+34 554 413-44	+34 554413-44
09823 880	+49 9823 880
050 5 55	+49 50 555
0802390	+49 802390
05111053	+49 5111053
0654999	+49 654999

Das Gesamtergebnis mit Verschachtelung

Das Ergebnis ist sicherlich korrekt. Nur ist fraglich, ob dieser SQL-Befehl wirklich verständlich ist. Oder ob er auch noch in einem Jahr nachvollziehbar wäre – und Kollegen diesen Befehl ebenfalls in angemessener Zeit verstehen könnten?

7.6.6
Seite 68

Wenn ich die Daten korrigieren müsste, würde ich immer eine Lösung in Einzelschritten vorziehen, und das sähe dann folgendermaßen aus:

```
SET TRANSACTION ISOLATION LEVEL CS;

ggf. Journaling sporadisch starten

CL: STRJRNPF FILE(EPSW09/CUSTOMER) JRN(EPSW09/QSQJRN) ;

-- Korrektur führende Null in +49

UPDATE CUSTOMER A
SET CUSTOMER_TELEPHONE =

(
SELECT TRIM(REGEXP_REPLACE(CUSTOMER_TELEPHONE, '^([0])', '+49 '))
FROM CUSTOMER B
WHERE A.CUSTOMER_NUMBER = B.CUSTOMER_NUMBER
);

-- Leerzeichen entfernen, aber nicht nach Ländervorwahl

UPDATE CUSTOMER A
SET CUSTOMER_TELEPHONE =

(
SELECT REGEXP_REPLACE(TRIM(CUSTOMER_TELEPHONE), '\S', '', 1, 2, 'I')
FROM CUSTOMER B
WHERE A.CUSTOMER_NUMBER = B.CUSTOMER_NUMBER
);

SELECT * FROM CUSTOMER;

COMMIT;
-- Journaling wieder beenden, falls gewünscht

CL: ENDJRNPF FILE(EPSW09/CUSTOMER) JRN(EPSW09/QSQJRN) ;
```

Es gibt weitere skalare Funktionen, die ebenfalls mit Suchmustern arbeiten:

Die Funktion REGEXP_COUNT ermittelt die Anzahl der Vorkommen des gesuchten Musters in einer Zeichenkette. Wir können mithilfe der Funktion beispielsweise ermitteln, wie oft das Pluszeichen – gefolgt von zwei Ziffern und einem Leerzeichen – in einer Zeichenspalte vorkommen kann:

```
SELECT CUSTOMER_TELEPHONE,
REGEXP_COUNT(CUSTOMER_TELEPHONE, '^(\+[0-9]{2})[\S]{1}')
FROM CUSTOMER;
```

Die skalare Funktion REGEXP_INSTR liefert die Startposition des Suchmusters oder die Position nach dem Ende des Suchmusters. In folgendem Beispiel wird die Position eines Minuszeichens – ausgehend vom Anfang der Zeichenkette – ermittelt.

```
SELECT
CUSTOMER_TELEPHONE,
REGEXP_INSTR(CUSTOMER_TELEPHONE, '\-', 1, 2, 0)
FROM CUSTOMER;
```

Sicherlich vermuten Sie schon, dass es auch eine neue SUBSTR-Funktion gibt, die mit Such-Mustern arbeitet. Die Funktion heißt REGEXP_SUBSTR. Mit dem nachfolgenden Statement schneiden wir das Länderkennzeichen der Telefonnummer aus:

```
SELECT CUSTOMER_TELEPHONE,
REGEXP_SUBSTR(CUSTOMER_TELEPHONE, '^(\+[1-9]{2,}[\s])', 1, 1)
FROM CUSTOMER;
```

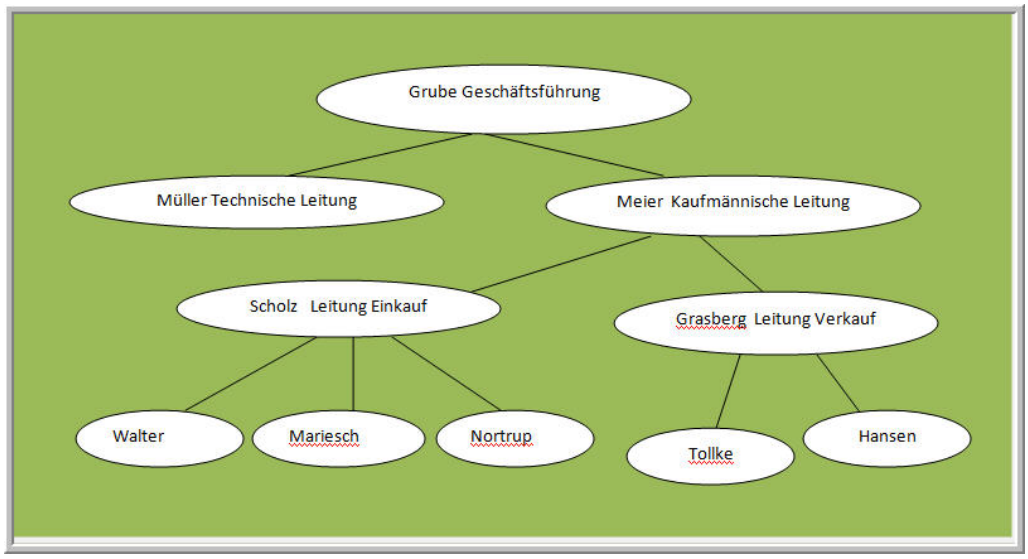
Der dritte Parameter der Funktion bestimmt wieder die Startposition und der vierte Parameter legt fest, welches Vorkommen ausgeschnitten werden soll.

7.6.6
Seite 70

7.6.8 Rekursive SQL-Aufrufe

„Als Rekursion (auch Rekurrenz oder Rekursivität), von lateinisch recurrere = zurücklaufen, bezeichnet man den Aufruf oder die Definition einer Funktion durch sich selbst. Die gegenseitige Rekursion bildet sich durch den gegenseitigen Verweis zweier oder mehrerer Funktionen auf einander. "... soweit Wikipedia. Wissen Sie jetzt wie Rekursionen wirken und wie sie zur Lösungsfindung eingesetzt werden können?

Tatsächlich sind rekursive Lösungsansätze nicht alltäglich, aber es gibt sie in Form von Stücklisten, Chargenrückverfolgungen oder Organigrammen. Grundsätzlich liegt einer rekursiven Lösung immer eine Baumstruktur zu Grunde:

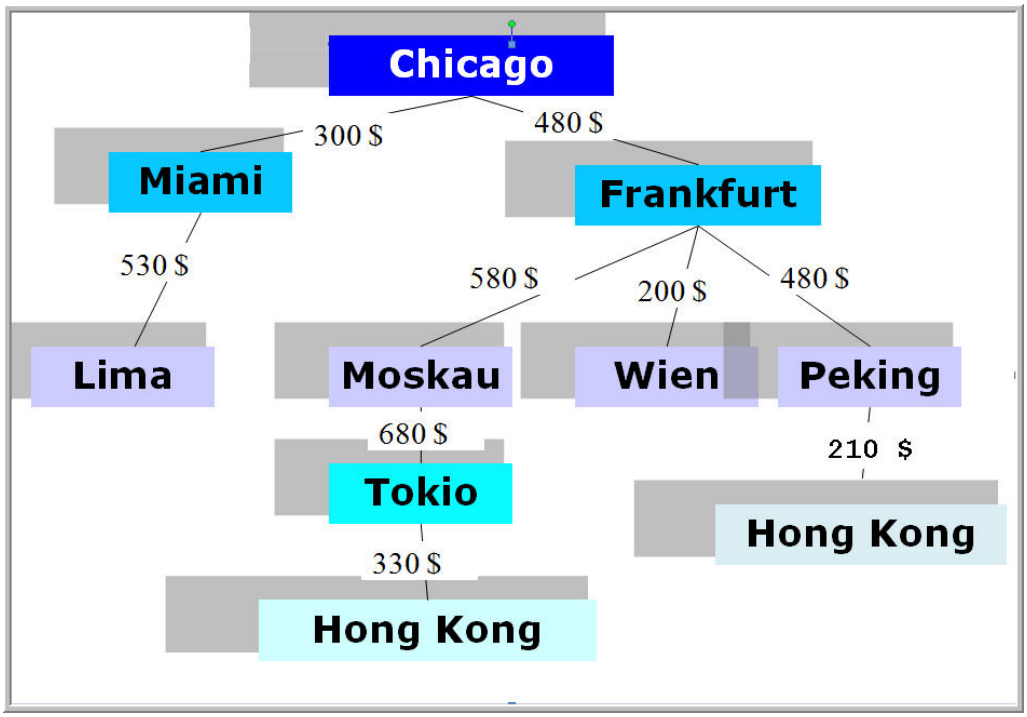


Organigramm

Ausgehend von einem Startpunkt sollen mit einer rekursiven Lösung die weiteren Ebenen aufgelöst werden. Hier als Beispiel: „Welche Mitarbeiter unterstehen der kaufmännischen Leitung?"

7.6.8
Seite 2

Schauen Sie sich die folgende Abbildung an. Sie zeigt alle Flugverbindungen ausgehend von Chicago.



Flugverbindungen von Chicago

Eine Frage könnte z. B. lauten „Welche Flugverbindungen stehen von Chicago nach Hong Kong zur Verfügung und was kosten sie?“

Bleiben wir fürs Erste bei diesem Beispiel. Als Lösungsansatz müssen wir rekursive CTEs verwenden (RCTE). Die Flugverbindungsdaten finden wir in der Tabelle FLIGHTS:

ITPSQL.FLIGHTS - 172.16.110.22(Epsserver)

Datei Bearbeiten Ansicht Zeilen Hilfe

	DEPARTURE	ARRIVAL	CARRIER	FLIGHT_NUMBER	PRICE
	Peking	Hong Kong	Asia Air	453	210
	New York	Paris	Atlantic	234	400
	Chicago	Miami	NA Air	2334	300
	New York	London	Atlantic	5473	350
	London	Athen	Mediterranean	247	340
	Athen	Nicosia	Mediterranean	2356	280
	Paris	Madrid	Euro Air	3256	380
	Paris	Kairo	Euro Air	63	480
	Chicago	Frankfurt	Atlantic	37	480

Die Tabelle FLIGHTS

Wir wollen in einem ersten Schritt alle Flüge ausgehend von Chicago ermitteln. Rekursive Abfragen benötigen immer drei Phasen und somit eigentlich immer drei Abfragen:

- Initialisierende Abfrage
- Iterative Abfrage
- Haupt-Abfrage

Die initialisierende Abfrage dient dazu, einen Ausgangspunkt – also einen Startpunkt – für die Iteration festzulegen. Die initialisierende Abfrage muss immer an erster Stelle innerhalb der RCTE codiert werden und wird nur ein einziges Mal ausgeführt.

```
WITH recursive Verbindungen (Abflughafen, Zielflughafen,
Anzahl_Fluege, Kosten)
AS
(
SELECT departure, arrival, 1, Price
FROM flights WHERE departure = 'Chicago'
```

Sie sehen, dass ich das Schlüsselwort RECURSIVE verwendet habe. Sie können es verwenden, um die Rekursion zu kennzeichnen. Einfluss auf die Ausführung der Abfrage hat es jedoch nicht.

Im nächsten Schritt erfolgt eine Verknüpfung der Basistabellen mit sich selbst, also die eigentliche Rekursion. Erst durch diese Iteration kommt es zur rekursiven Abfrage. Die Verbindung zwischen der Originaltabelle FLIGHTS und der CTE VERBINDUNGEN erfolgt immer über eine Parent-Child-Beziehung. In unserem Fall wird der Abflughafen (Departure) mit dem Zielflughafen (Arrival) verknüpft, d. h. der Abflughafen kann als „Elternteil“ und die untergeordneten Zielflughäfen als „Kinder“ angesehen werden.

```
UNION ALL
SELECT Ver.Abflughafen, f.arrival, ver.Anzahl_Fluege + 1 ,
Kosten + Price
FROM Verbindungen Ver, Flights F
WHERE Ver.Zielflughafen = F.departure
)
```

7.6.8
Seite 4

Die Rekursion wird solange ausgeführt, bis die unterste Ebene erreicht wird, d. h. die Verarbeitung endet, wenn dem Zielflughafen kein Abflughafen mehr zugeordnet werden kann. Sind alle Flugverbindungen aufgelöst, kann die ausführende Abfrage folgen:

```
SELECT Abflughafen, Zielflughafen, Anzahl_Fluege, Kosten
FROM Verbindungen
WHERE Zielflughafen = ‚Hong Kong‘;
```

In unserem ersten Beispiel werden der Ausgangsflughafen „Chicago“, der Zielflughafen, die Anzahl der notwendigen Flüge und die damit verbundenen Kosten angezeigt. Als Bedingung können wir jetzt noch hinzufügen, dass uns lediglich die Flüge nach Hong Kong interessieren.

```
WITH Verbindungen (Abflughafen, Zielflughafen, Anzahl_Fluege, Kosten)
AS
(

SELECT departure,  arrival, 1, Price
FROM flights  WHERE departure = 'Chicago'

UNION ALL

SELECT Ver.Abflughafen,  f.arrival, ver.Anzahl_Fluege + 1 , Kosten + Price
FROM Verbindungen Ver, Flights F
WHERE Ver.Zielflughafen = F.departure
)

SELECT Abflughafen, Zielflughafen, Anzahl_Fluege, Kosten
FROM Verbindungen
WHERE Zielflughafen = 'Hong Kong';
```

ABFLUGHAFEN	ZIELFLUGHAFEN	ANZAHL_FLUEGE	KOSTEN
Chicago	Hong Kong	3	1170
Chicago	Hong Kong	4	2070

Das Ergebnis der Rekursion

Rekursive Common-Table-Expressions unterliegen einigen Restriktionen, die Sie beachten müssen:

- Die CTE benötigt zwingend die Spaltennamen nach dem CTE-Identifier.
- Die initialisierende und die iterative Abfrage müssen mit der Anweisung SELECT beginnen. Die Anweisung SELECT DISTINCT ist dabei nicht erlaubt.
- Die beiden Abfragen müssen über den UNION ALL Operator verknüpft werden; UNION ist nicht gestattet.

- Abfragen, die Teil der Rekursion sind, dürfen keine Aggregatfunktionen, keine GROUP BY oder HAVING-Klausel enthalten und dürfen zudem mit Ausnahme der initialisierenden Abfrage keine ORDER BY-Klausel nutzen.
- Die Klausel FULL OUTER JOIN ist in den iterativen Abfragen nicht erlaubt. Die Klausel LEFT OUTER JOIN hingegen darf nur nicht genutzt werden, wenn die CTE der rechte JOIN-Operand ist. Analog gilt: Ist die CTE der linke Operand, darf die Klausel RIGHT OUTER JOIN nicht verwendet werden.

Nun gehen wir einen Schritt weiter und hinterlegen unsere Abfrage als SQL-View. Dadurch ändert sich der Aufbau unseres SQL-Statements ein wenig.

```
CREATE View Flugverbindungen(Abflughafen, Zielflughafen,
Anzahl_Fluege, Kosten)
AS


(
SELECT departure, arrival, 1, Price
FROM flights WHERE departure = 'Chicago'
UNION ALL


SELECT Ver.Abflughafen, f.arrival, ver.Anzahl_Fluege + 1 ,
Kosten + Price
FROM Flugverbindungen Ver, Flights F
WHERE Ver.Zielflughafen = F.departure
)
```

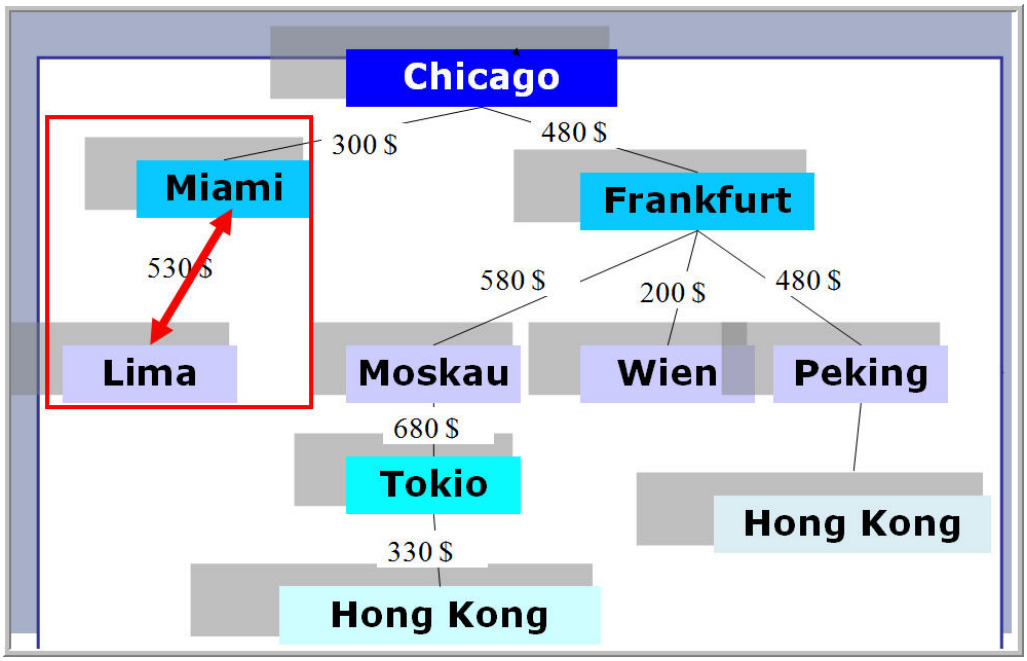
Innerhalb einer SQL-View können wir nicht mit CTEs arbeiten. Vielmehr ersetzt die View die CTE des ersten Beispiels. Die View FLUGVERBINDUNGEN enthält zudem auch nur die initialisierende Abfrage und die Iteration. Die Hauptabfrage hat keinen direkten Bezug mehr zur Rekursion.

```
SELECT *
FROM Flugverbindungen
WHERE Zielflughafen = 'Hong Kong';
```

7.6.8.1 Wechselseitige Rekursionen

Was passiert aber, wenn wechselseitige Rekursionen nicht ausgeschlossen werden können?

Schauen Sie sich die folgende Grafik an:



Problem der wechselseitigen Rekursion

Jetzt ist ein Rückflug von Lima nach Miami integriert. Unsere Abfrage würde in einer Endlosschleife laufen:

ABFLUGHAFEN	ZIELFLUGHAFEN	ANZAHL_FLUEGE	KOSTEN
Chicago	Hong Kong	3	1170
Chicago	Lima	4	1890
Chicago	Hong Kong	4	2070
Chicago	Miami	5	2420
Chicago	Lima	6	2950
Chicago	Miami	7	3480
Chicago	Lima	8	4010
Chicago	Miami	9	4540
Chicago	Lima	10	5070

Endlosschleifen

Wir müssen also verhindern, dass ein Abflug- und Zielflughafen mehrfach vorkommt.

```
WITH Verbindungen (Abflughafen, Zielflughafen, Anzahl_Fluege,
Kosten)
AS
(
SELECT departure, arrival, 1, Price
FROM flights WHERE departure = 'Chicago'
UNION ALL
SELECT Ver.Abflughafen, f.arrival, ver.Anzahl_Fluege + 1 ,
Kosten + Price
FROM Verbindungen Ver, Flights F
WHERE Ver.Zielflughafen = F.departure
)
CYCLE Zielflughafen SET Stop TO '1' DEFAULT '0'
SELECT Abflughafen, Zielflughafen, Anzahl_Fluege, Kosten
FROM Verbindungen
WHERE Zielflughafen = 'Hong Kong';
```

Die Klausel CYCLE in einer CTE überwacht das Mehrfachvorkommen einer Parent-Child-Beziehung. Wenn der Abflughafen und der Zielflughafen im Beispiel aufgrund einer wechselseitigen Rekursion mehrfach in einem Zweig selektiert werden, wird ein Flag gesetzt. In meinem Beispiel ist dies die Variable STOP. Sie können jeden anderen Variablennamen verwenden und müssen die Variable auch nicht definieren. Sie wird automatisch mit dem Datentyp CHAR(1) deklariert. Sie erhält den Standardwert 0 und ihr wird der Wert 1 zugewiesen, wenn eine wechselseitige Rekursion auftritt. Dies geschieht alles automatisch. Ein Eingreifen Ihrerseits ist nicht gefordert.

ABFLUGHAFEN	ZIELFLUGHAFEN	ANZAHL_FLUEGE	KOSTEN	STOP
Chicago	Frankfurt	1	480	0
Chicago	Hong Kong	3	1170	0
Chicago	Hong Kong	4	2070	0
Chicago	Lima	2	830	0
Chicago	Miami	1	300	0
Chicago	Miami	3	1360	1
Chicago	Moskau	2	1060	0
Chicago	Peking	2	960	0
Chicago	Tokio	3	1740	0
Chicago	Wien	2	680	0

Das Flag wird automatisch gesetzt

Sie können in der oberen Grafik erkennen, dass die Zeile, die zum Abbruch der Verarbeitung in dem entsprechenden Zweig geführt hat, im Result Set enthalten ist. Sie müssten dies in Ihrer Hauptabfrage daher ggf. berücksichtigen.

7.6.8
Seite 8

In einer rekursiven View können Sie nun aber die CYCLE-Klausel leider nicht nutzen. Wir umgehen dieses Problem daher, indem wir die View auf der CTE aufbauen:

```
CREATE View Flugverbindungen(Abflughafen, Zielflughafen,
Anzahl_Fluege, Kosten, Stop)

AS
```

Zuerst erstellen wir die View. Jetzt kodieren wir kein einfaches SELECT-Statement, sondern verwenden unsere CTE:

```
With CTE (Abflughafen, Zielflughafen, Anzahl_Fluege, Kosten)

AS

(
SELECT departure, arrival, 1, Price
FROM flights WHERE departure = 'Chicago'

UNION ALL

SELECT CTE.Abflughafen, f.arrival, CTE.Anzahl_Fluege + 1 ,
Kosten + Price
FROM CTE, Flights F
WHERE CTE.Zielflughafen = F.departure
)

CYCLE Zielflughafen SET Stop TO '1' DEFAULT '0'
```

Sie dürfen hierbei nicht die Hauptabfrage der CTE auslassen und müssen daran denken, dass sich jede Selektion in der Hauptabfrage natürlich auf das spätere Result Set der View auswirkt.

```
SELECT Abflughafen, Zielflughafen, Anzahl_Fluege, Kosten,
Stop
FROM CTE;
```

Ist die View auf diesem Weg erstellt, können Sie wieder problemlos mit den Daten arbeiten:

```
Select Zielflughafen, Kosten
from Flugverbindungen
where Stop != '1';
```

ZIELFLUGHAFEN	KOSTEN
Miami	300,00
Frankfurt	480,00
Lima	830,00
Moskau	1060,00
Peking	960,00
Wien	680,00
xxxxx	920,00
Tokio	1740,00
Hong Kong	1170,00
Hong Kong	2070,00

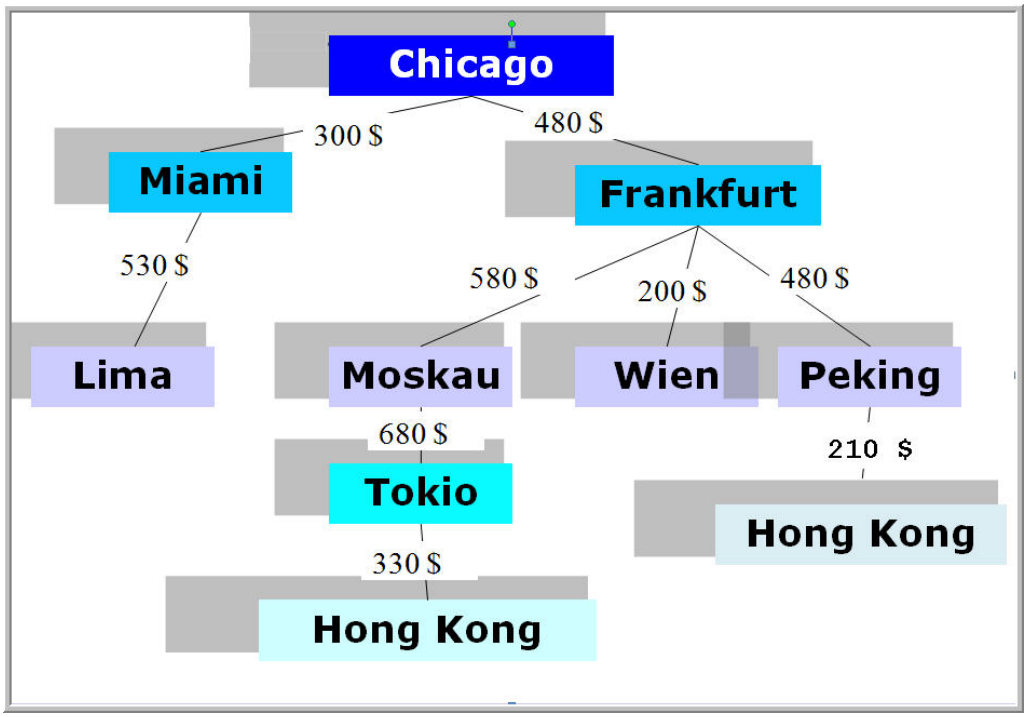
Das Ergebnis

7.6.8.2 Rekursive SQL-Aufrufe – CONNECT BY

Vor einiger Zeit haben wir uns rekursive SQL-Lösungen angeschaut. Aber die DB2 for i unterstützt zwei unterschiedliche Syntaxen: Wir können rekursive Common Table Expressions (RCTE) – wie in den vorangegangenen Kapiteln gezeigt – verwenden oder arbeiten mit der Klausel CONNECT BY. Jede Methode hat ihre spezifischen Vor- und Nachteile.

Die Klausel CONNECT BY ist viel einfacher zu verstehen, bietet aber weniger Möglichkeiten, den Datenabruf zu steuern. So können Sie mit der Klausel CONNECT BY beispielsweise nicht die Sortierreihenfolge der Parent-Child-Beziehung steuern. Standardmäßig erhalten Sie bei der CONNECT BY-Klausel immer die Reihenfolge DEPTH FIRST. Dazu aber mehr in den Beispielen.

Benutzen wir auch dieses Mal wieder unsere Tabelle FLIGHTS, um die CONNECT BY-Klausel kennenzulernen:



Flugverbindung von Chicago

Wie zuvor, stellen wir die Frage: „Welche Flugverbindungen stehen von Chicago nach Hong Kong zur Verfügung und was kosten sie?“

Die Flugverbindungsdaten finden wir wie gehabt in der Tabelle FLIGHTS:

ITPSQL.FLIGHTS - 172.16.110.22(Epsserver)

Datei Bearbeiten Ansicht Zeilen Hilfe

DEPARTURE	ARRIVAL	CARRIER	FLIGHT_NUMBER	PRICE
Peking	Hong Kong	Asia Air	453	210
New York	Paris	Atlantic	234	400
Chicago	Miami	NA Air	2334	300
New York	London	Atlantic	5473	350
London	Athen	Mediterranean	247	340
Athen	Nicosia	Mediterranean	2356	280
Paris	Madrid	Euro Air	3256	380
Paris	Kairo	Euro Air	63	480
Chicago	Frankfurt	Atlantic	37	480

Die Tabelle FLIGHTS

Wir wollen in einem ersten Schritt alle Flüge ausgehend von Chicago ermitteln. Anders als bei RCTEs benötigen wir mit der CONNECT BY-Klausel keine drei Phasen:

```
SELECT CONNECT_BY_ROOT Departure AS Ausgangspunkt,

Departure AS Abflug , Arrival AS Ankunft , LEVEL AS Anzahl

FROM Flights

START WITH departure = 'Chicago'
-- WHERE departure = 'Chicago'

CONNECT BY PRIOR arrival = departure
-- FROM Verbindungen ver, Flights F
-- WHERE ver.Zielflughafen = F.departure
```

Lassen Sie uns nun das Statement analysieren. Mit der Klausel START WITH bestimmen wir den Ausgangspunkt. Dies entspricht unserer WHERE-Klausel in der initiierenden RCTE. Wir zuvor, nutzen wir wieder „CHICAGO“ als Ausgangspunkt.

Anschließend codieren wir die Iteration, indem wir die Klausel CONNECT BY PRIOR verwenden. Durch das Schlüsselwort PRIOR geben Sie bekannt, dass in den zuvor selektierten Sätzen der Ankunftsort als Verknüpfungsfeld für die Iteration verwendet werden soll. Letztendlich codieren Sie an dieser Stelle eine INNER JOIN-Bedingung. Normalerweise dürfte dafür eine einzelne Spalte ausreichend sein; Sie können aber auch über mehrere Spalten die Verknüpfung herstellen. Die CONNECT BY PRIOR-Klausel entspricht unserer WHERE-Bedingung in der Iteration der RCTE.

7.6.8
Seite 12

Die Hauptabfrage verbirgt sich hinter der Klausel CONNECT_BY_ROOT. Hier legen Sie die Spalten für das Result Set fest. Der Ausdruck LEVEL ist ein feststehender Begriff, wenn Sie die CONNECT BY-Syntax verwenden. Diese Pseudospalte repräsentiert die Rekursionstiefe, in unserem Beispiel also die Anzahl der Flüge. Schauen wir uns jetzt das Ergebnis an:

AUSGANGSPUNKT	ABFLUG	ANKUNFT	ANZAHL
Chicago	Chicago	Miami	1
Chicago	Miami	Lima	2
Chicago	Chicago	Frankfurt	1
Chicago	Frankfurt	Vienna	2
Chicago	Frankfurt	Beijing	2
Chicago	Frankfurt	Moscow	2
Chicago	Moscow	Tokyo	3
Chicago	Tokyo	Hong Kong	4

Das Ergebnis

Wenn Sie die beiden Syntaxen miteinander vergleichen, können Sie auf den ersten Blick sehen, dass das CONNECT BY-Statement sehr viel „kompakter“ ist und daher vermutlich für viele auch einfacher umzusetzen:

```
-- Rekursive CTEWITH recursive Verbindungen (Abflughafen,
Zielflughafen, Anzahl_Fluege)
AS

(
SELECT departure, arrival, 1
FROM flights
WHERE departure = 'Chicago'

UNION ALL

SELECT Ver.Abflughafen, f.arrival, ver.Anzahl_Fluege + 1
FROM Verbindungen Ver, Flights F
WHERE Ver.Zielflughafen = F.departure
)
SELECT *
FROM Verbindungen;
```

```
Oder zum direkten Vergleich die Klausel CONNECT BY

SELECT CONNECT_BY_ROOT Departure AS Ausgangspunkt,

Departure AS Abflug , Arrival AS Ankunft , LEVEL AS Anzahl

FROM flights

START WITH departure = 'Chicago'

CONNECT BY PRIOR arrival = departure
```

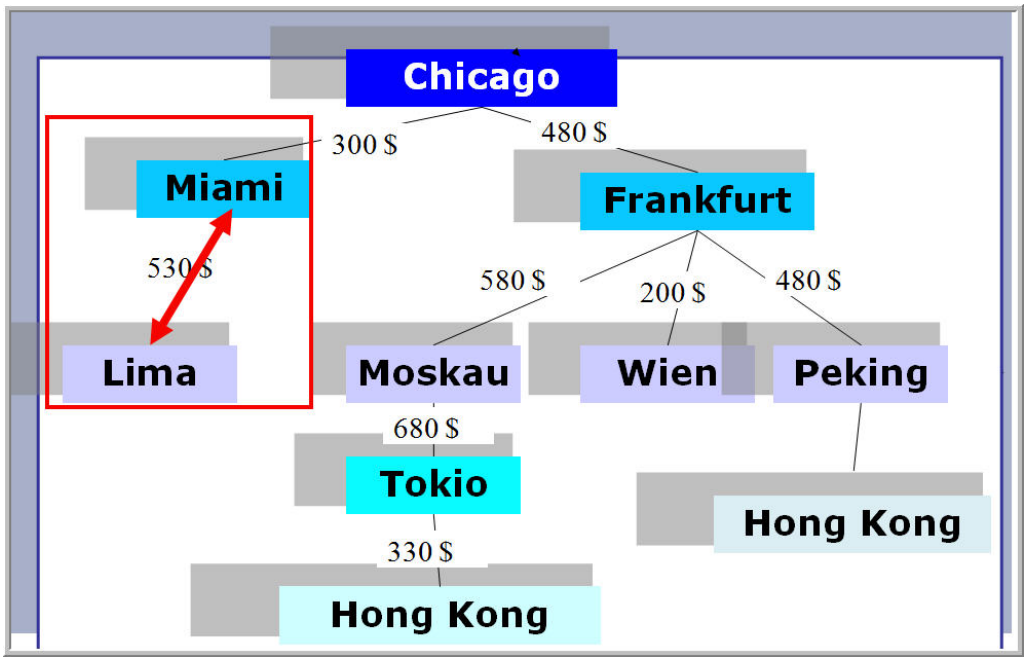
Natürlich können wir auch die Klausel CONNECT BY innerhalb einer View verwenden:

```
CREATE OR REPLACE View
Flugverbindungen(Abflughafen, Zielflughafen, Anzahl_Fluege)
AS

(
SELECT CONNECT_BY_ROOT Departure as Abflug, Departure, Arri-
val, LEVEL

FROM flights

START WITH departure = 'Chicago'

CONNECT BY PRIOR arrival = departure
) ;
```

Diesbezüglich lässt sich kein Unterschied zwischen den beiden Syntaxen feststellen.

7.6.8
Seite 14

Wie verhält sich die CONNECT BY-Klausel aber in Bezug auf wechselseitige Rekursion? Nutzen wir auch hier wieder das bereits verwendete Beispiel aus dem Kapitel 7.6.8.1:



Problem der wechselseitigen Rekursion

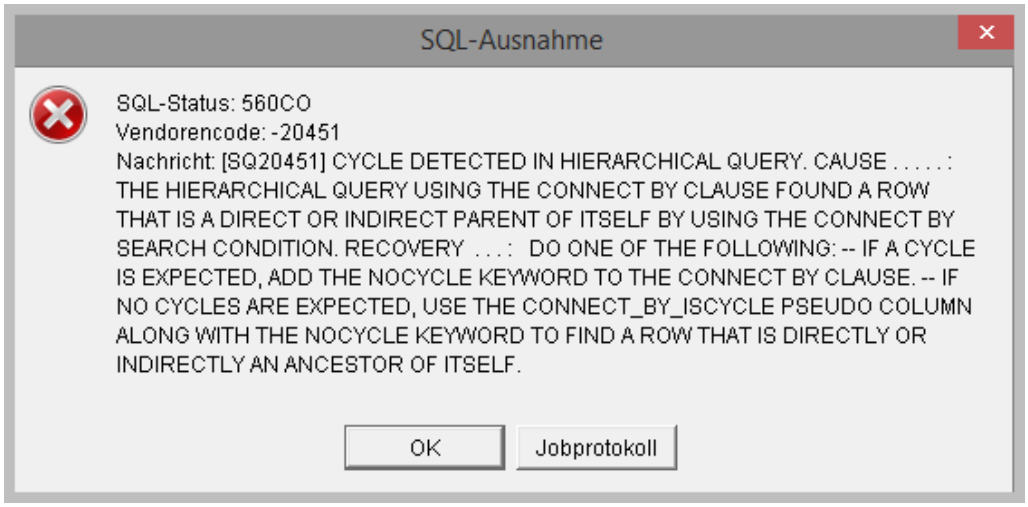
Wir konnten im Kapitel 7.6.8.1 sehen, dass unsere RCTE Mehrfachausgaben im Result Set zeitigt und wir eine Endlosschleife bekommen. Sie müssen allerdings keine Angst haben, die Schleife beendet sich nach einiger Zeit selbstständig! Schauen Sie sich das Ergebnis noch einmal an:

ABFLUGHAFEN	ZIELFLUGHAFEN	ANZAHL_FLUE...
Chicago	Miami	1
Chicago	Frankfurt	1
Chicago	Lima	2
Chicago	Moscow	2
Chicago	Beijing	2
Chicago	Vienna	2
Chicago	Miami	3
Chicago	Tokyo	3
Chicago	Lima	4
Chicago	Hong Kong	4
Chicago	Miami	5
Chicago	Lima	6
Chicago	Miami	7
Chicago	Lima	8
Chicago	Miami	9
Chicago	Lima	10
Chicago	Miami	11

Endlosschleifen

Und wie verhält sich die CONNECT BY-Klausel?

Wenn in einem CONNECT BY-Statement Endlosschleifen festgestellt werden, wird das Statement sofort automatisch mit dem SQLSTATUS SQ20451 abgebrochen:



Die Fehlermeldung

Sie müssen sich also niemals vor Endlosschleifen fürchten. Falls Sie aber trotz wiederkehrender Ergebnisse die Ergebnismenge benötigen, müssen Sie das Schlüsselwort NOCYCLE zusammen mit der Pseudospalte CONNECT_BY_ISCYCLE verwenden.

```
SELECT CONNECT_BY_ROOT
Departure AS Ausgangspunkt, Departure as Abflug , Arrival as
Ankunft,
LEVEL AS Anz_Fluege ,

CONNECT_BY_ISCYCLE as Zirkel

 FROM flights
 START WITH departure = 'Chicago'

 CONNECT BY NOCYCLE PRIOR arrival = departure;
```

7.6.8
Seite 16

Anhand der Pseudospalte „Zirkel“ lassen sich wechselseitige Rekursionen anschließend leicht identifizieren und ausschließen:

AUSGANGSPUNKT	ABFLUG	ANKUNFT	ANZ_FLUEGE	ZIRKEL
Chicago	Chicago	Miami	1	0
Chicago	Miami	Lima	2	0
Chicago	Lima	Miami	3	1
Chicago	Chicago	Frankfurt	1	0
Chicago	Frankfurt	Vienna	2	0
Chicago	Frankfurt	Beijing	2	0
Chicago	Frankfurt	Moscow	2	0
Chicago	Moscow	Tokyo	3	0
Chicago	Tokyo	Hong Kong	4	0

Wechselseitige Rekursionen ausschließen

Sie sehen, ähnlich wie bei einer RCTE nutzen wir eine Klausel, um die wechselseitigen Rekursionen abzufangen. Und im Gegensatz zur RCTE können Sie die CYCLE-Klausel auch direkt in einer View verwenden:

```
CREATE or REPLACE VIEW Flugverbindungen(Ausgangspunkt, Abflug, Ankunft,
Anzahl_Fluege, Zirkel)

AS

(

SELECT CONNECT_BY_ROOT Departure AS Ausgangspunkt, Departure as Abflug ,
Arrival as Ankunft, LEVEL AS Anz_Fluege ,
CONNECT_BY_ISCYCLE as Zirkel       -- Pseudospalte
FROM flights
START WITH departure = 'Chicago'
CONNECT BY NOCYCLE PRIOR arrival = departure -- Klausel NOCYCLE
);
```

Sie haben sicherlich schon festgestellt, dass Pseudospalten für die CONNECT BY-Syntax eine erhebliche Rolle spielen. So benötigen Sie beispielsweise auch eine Pseudospalte, wenn Sie im Result Set Spalten miteinander verketten wollen. Schauen Sie sich die nachfolgende Abbildung an:

AUSGANGSPUNKT	ABFLUG	ANKUNFT	FLUGVERBINDUNGEN	ANZ_FLUEGE	ZIRKEL
Chicago	Chicago	Miami	: Miami	1	0
Chicago	Miami	Lima	: Miami : Lima	2	0
Chicago	Lima	Miami	: Miami : Lima : Miami	3	1
Chicago	Chicago	Frankfurt	: Frankfurt	1	0
Chicago	Frankfurt	Vienna	: Frankfurt : Vienna	2	0
Chicago	Frankfurt	Beijing	: Frankfurt : Beijing	2	0
Chicago	Frankfurt	Moscow	: Frankfurt : Moscow	2	0
Chicago	Moscow	Tokyo	: Frankfurt : Moscow : Tokyo	3	0
Chicago	Tokyo	Hong Kong	: Frankfurt : Moscow : Tokyo : Hong Kong	4	0

Spalten verketten

Die Flugverbindungen werden in einer eigenen Spalte verkettet. Dies gelingt nur mittels der Pseudospalte SYS_CONNECT_BY_PATH:

```
SELECT CONNECT_BY_ROOT Departure AS Ausgangspunkt,
Departure as Abflug , Arrival as Ankunft,
SYS_CONNECT_BY_PATH(TRIM(Arrival), ' : ') as Flugverbindungen,
LEVEL AS Anz_Fluege ,

 CONNECT_BY_ISCYCLE as Zirkel

 FROM flights
 START WITH departure = 'Chicago'
 CONNECT BY NOCYCLE PRIOR arrival = departure ;
```

Auch wenn Sie alle Zeilen selektieren wollen, die keine Child-Sätze enthalten, müssen Sie eine Pseudospalte zu Hilfe nehmen.

```
SELECT CONNECT_BY_ROOT Departure AS Ausgangspunkt,
Departure as Abflug , Arrival as Ankunft,
SYS_CONNECT_BY_PATH(TRIM(Arrival), ' : ') as Flugverbindungen,
CONNECT_BY_ISLEAF ANZ_CHILD,
LEVEL AS Anz_Fluege ,
 CONNECT_BY_ISCYCLE as Zirkel      -- Pseudospalte
 FROM flights
 START WITH departure = 'Chicago'
 CONNECT BY NOCYCLE PRIOR arrival = departure;
```

Schauen wir uns das Ergebnis an:

AUSGAN...	ABFLUG	ANKUNFT	FLUGVERBINDUNGEN	LEAF	ANZ_FLUEGE	ZIRKEL
Chicago ...	Chicago ...	Miami ...	: Miami	0	1	0
Chicago ...	Miami ...	Lima ...	: Miami : Lima	0	2	0
Chicago ...	Lima ...	Miami ...	: Miami : Lima : Miami	1	3	1
Chicago ...	Chicago ...	Frankfurt ...	: Frankfurt	0	1	0
Chicago ...	Frankfurt ...	Vienna ...	: Frankfurt : Vienna	1	2	0
Chicago ...	Frankfurt ...	Beijing ...	: Frankfurt : Beijing	1	2	0
Chicago ...	Frankfurt ...	Moscow ...	: Frankfurt : Moscow	0	2	0
Chicago ...	Moscow ...	Tokyo ...	: Frankfurt : Moscow : Tokyo	0	3	0
Chicago ...	Tokyo ...	Hong Kong ...	: Frankfurt : Moscow : Tokyo : Hong Kong	1	4	0

Ergebnis der Pseudospalte ISLEAF

7.6.8
Seite 18

Mit Hilfe der Pseudospalte CONNECT_BY_ISLEAF können Sie feststellen, welche Flughäfen Endstationen sind. Sie steuern darüber keineswegs die Rekursionstiefe. Allerdings können Sie diese Spalte auch direkt in einer WHERE-Bedingung verwenden.

```
SELECT Arrival as Ankunft, LEVEL AS Anz_Fluege
FROM flights

WHERE CONNECT_BY_ISLEAF     = 0

START WITH departure = 'Chicago'
CONNECT BY NOCYCLE PRIOR arrival = departure;
```

Für die Pseudospalte CONNECT_BY_ISLEAF finden wir in einer RCTE keinen unmittelbaren Vergleich. Können wir daraus schließen, dass die CONNECT BY-Syntax die bessere Wahl ist? Keinesfalls, denn es gibt auch bei dieser Syntax Einschränkungen: Beispielsweise können Sie die Sortierfolge nicht steuern. Für die CONNECT_BY-Syntax gibt es zum Beispiel keine Entsprechung für die folgenden Klauseln:

```
SEARCH BREADTH FIRST BY spalte1, spalte2… SET Search-Spalte
```

```
SEARCH DEPTH FIRST BY spalte1, spalte2… SET Search-Spalte
```

Sie erhalten Ihr Result Set stets nach dem Prinzip SEARCH DEPTH FIRST. Für jeden Parent-Satz werden die zugehörigen Child-Sätze ausgegeben.

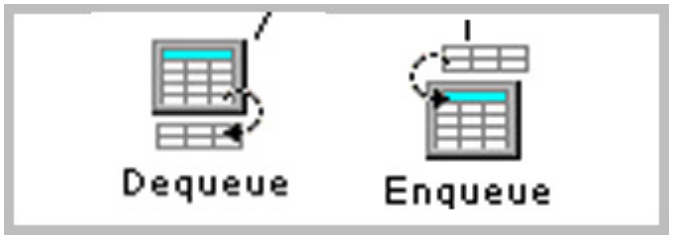
Und noch etwas müssen Sie bedenken, wenn Sie sich für eine Syntax entscheiden: Es wird Ihnen nicht gelingen, während der Rekursion Summen zu bilden. So ist es zum Beispiel nicht möglich, mit der CONNECT BY-Syntax die Flugpreise zu kumulieren, um die Gesamtkosten zu ermitteln.

7.6.8.3 Optimierung rekursiver SQL-Aufrufe

Rekursive SQL-Abfragen können sehr komplex sein und gegebenenfalls lange Laufzeiten haben. Umso wichtiger ist es, diese Abfragen individuell zu optimieren und einige Regeln bei ihrer Erstellung zu berücksichtigen.

Schauen wir uns zunächst einmal an, wie der SQL-Optimizer rekursive Lösungen implementiert.

Die initialisierenden Zeilen werden in eine Queue gestellt (DEQUEUE) und sukzessive von dort wieder ausgelesen (ENQUEUE). Ist die Queue leer, wird die Abfrage beendet. Die nachfolgenden Symbole repräsentieren das neue Verfahren:



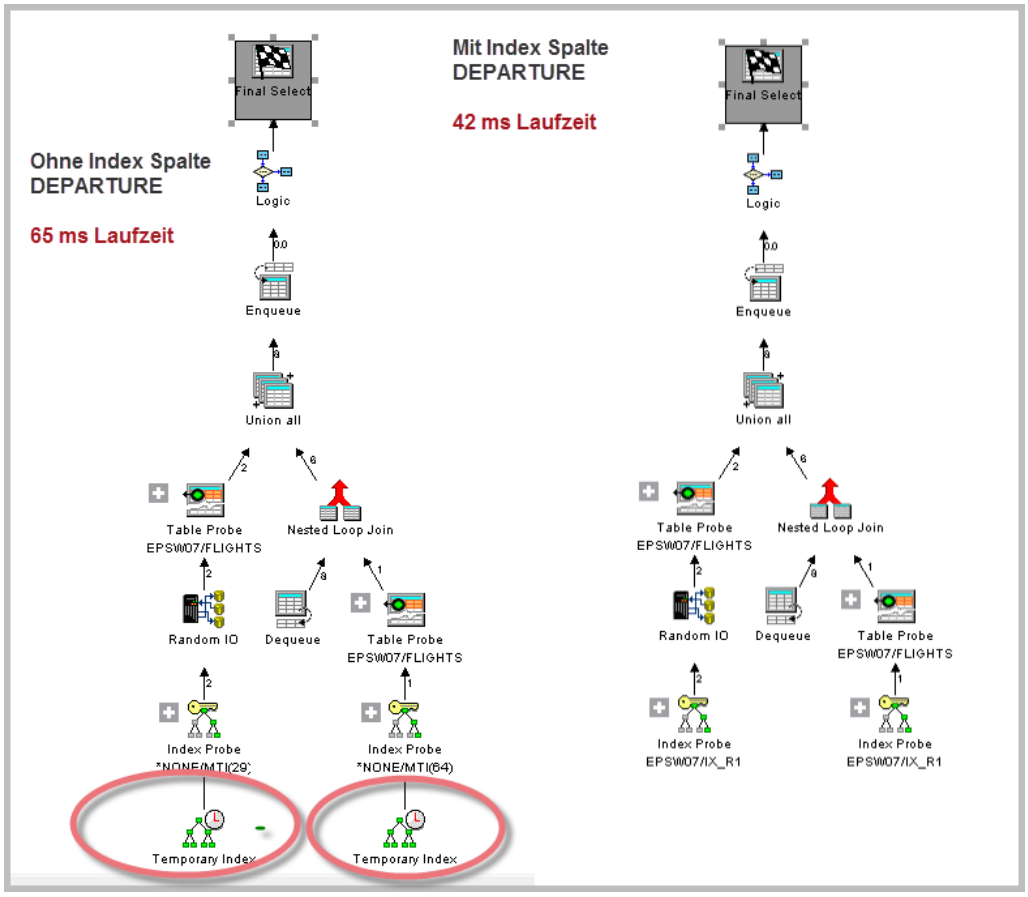
Symbole rekursiver Verarbeitung

Welche Indizes sind für diesen Datenabruf erforderlich? Die Verknüpfungsspalten müssen als Index vorliegen. Was ist damit gemeint? In meinem Beispiel verwende ich die Spalte DEPARTURE der Tabelle FLIGHTS sowohl in der initialisierenden Abfrage als auch für die rekursive Verknüpfung:

```
WITH recursive destinations (original, arrival, flight_count )
as
(
SELECT departure, arrival, 1
FROM flights
WHERE departure = 'Chicago'

UNION ALL

SELECT d.original, f.arrival, d.flight_count + 1
FROM destinations d, flights F
WHERE d.arrival = f.departure
)


SELECT original, arrival as "Zielflughafen", flight_count as "Anzahl Flüge"
FROM destinations;
```

7.6.8
Seite 20

Folglich benötigt die Tabelle FLIGHTS einen Index auf der Spalte DEPARTURE:



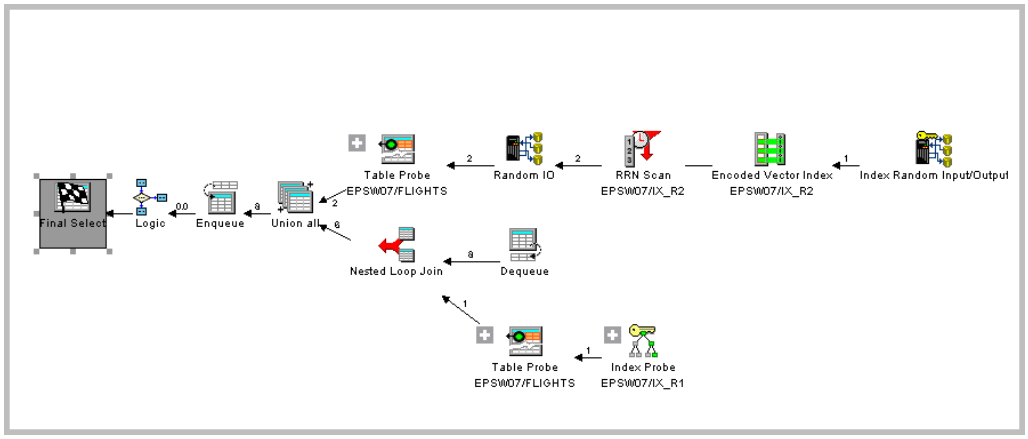
Optimierte rekursive Abfragen

Durch den Index reduziert sich die Laufzeit auf meinem Server um ca. 30 Prozent. Muss der temporäre Index auch noch aufgebaut werden – er liegt also nicht im Planungscache – benötigt die Abfrage mit dem temporären Index sogar 92 ms. Ein Index auf die Spalte ARRIVAL ist im Übrigen unnütz, da diese Spalte nicht aus der Tabelle FLIGHTS stammt, sondern aus der CTE.

Alternativ könnte es sinnvoll sein, anstatt eines Binärindizes auch einen Encoded Vector Index (EVI) zu erstellen, da es sich bei dem Abflughafen um eine Spalte mit überschaubarer Kardinalität handelt und die Tabelle sicherlich nicht täglich um weitere Flugverbindungen erweitert wird. Schauen wir was passiert:

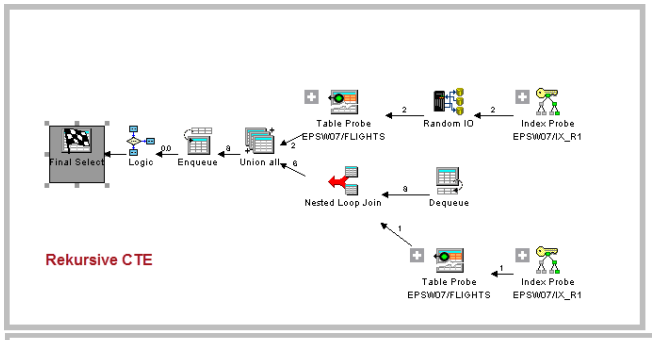
```
CREATE ENCODED VECTOR INDEX IX_R2 ON Flights(Departure);
```

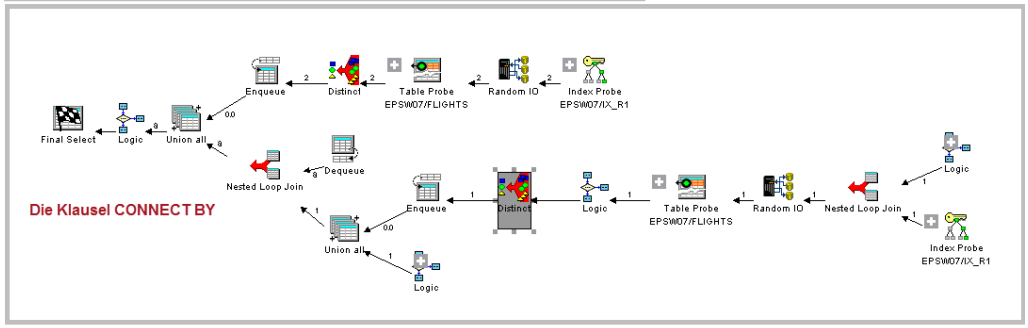
Unser Ausführungsplan ändert sich jetzt, da der Optimizer sich tatsächlich entscheidet, den EVI zu verwenden:



Auch EVIs können in rekursiven Abfragen eine Rolle spielen

Kommen wir in diesem Zusammenhang noch einmal auf die unterschiedlichen Syntaxen zurück. In den vorangegangenen Kapiteln haben Sie gesehen, dass Sie anstatt der rekursiven CTEs auch mit der CONNECT BY-Klausel arbeiten können. Aus der Erfahrung heraus weiß ich, dass viele Programmierer diese Syntax bevorzugen, da Sie kompakter und vielleicht auch einfacher zu verstehen ist. Aber auch in diesem Fall bestätigt sich die Aussage, dass nicht alles, was für den Programmierer schneller geht, auch der bessere Weg für die Datenbank ist. Schauen Sie sich das Ausführungsdiagramm der CONNECT BY-Syntax einmal an, dann werden Sie feststellen, dass die Anweisung sehr viel mehr Overhead erzeugt, trotz identischem Result Set:





Laufzeitvergleich CONNECT BY versus RCTEs

7.6.8
Seite 22

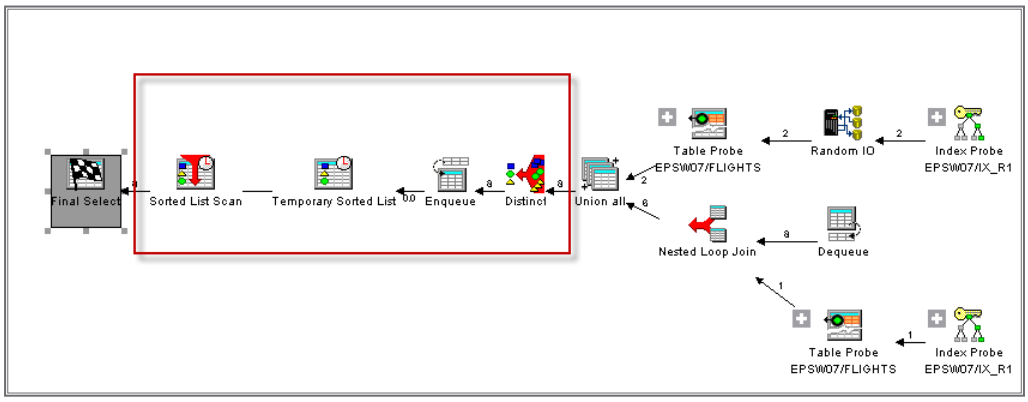
Zwei Gründe gibt es für dieses Phänomen:
Erstens habe ich Ihnen erläutert, dass die CONNECT BY-Syntax stets automatisch wechselseitige Rekursionen überprüft.

Zweitens unterscheiden sich die Sortierfolgen. Wenn Sie die CONNECT BY-Syntax verwenden, wird stets der Parent-Satz gefolgt von den Child-Sätzen und allen weiteren abhängigen Sätzen ausgegeben bevor der nächste Parent-Satz folgt. Anders verhält es sich bei rekursiven CTEs, dort erhalten Sie in der Regel zunächst die Parent-Zeilen und erst danach die Child-Sätze, gefolgt von den „Enkeln“ und „Urenkeln“.

Was folgern wir daraus?
Wenn Ihre rekursiven CONNECT-BY-Abfragen zu lange dauern, sollten Sie sich auf jeden Fall für rekursive CTEs entscheiden. Außerdem sollten Sie keinesfalls vorsorglich die CYCLE-Klausel in Ihrer CTE verwenden. Diese Klausel nutzen Sie nur, wenn Sie wissen, dass sich Endlosschleifen ergeben können und/oder wenn Sie die betroffenen Zeilen analysieren wollen.

Da rekursive SQL-Abfragen Queues nutzen, um die JOIN-Verknüpfungen zu realisieren, wird das Prinzip FIRST IN FIRST OUT praktiziert, solange Sie keine Sortierfolge explizit erzwingen. In der Regel erhalten Sie die Daten dadurch in der Sortierfolge BREADTH FIRST. BREADTH FIRST heißt, Sie erhalten zunächst alle direkten Child-Sätze eines Parent bevor die nächsten Sätze (= Enkelkinder) ausgegeben werden. Garantiert ist diese Sortierfolge allerdings nicht!

Eine garantierte Sortierfolge erhalten Sie nur, wenn Sie die Klauseln BREADTH FIRST oder DEPTH FIRST verwenden. Allerdings muss dadurch der Ausführungsplan deutlich erweitert werden:



Explizite Sortierfolgen in rekursiven Abfragen

Sie können in der vorangegangenen Abbildung sehen, dass die Sortierfolge immer über eine temporär sortierte Liste implementiert wird, und das kostet zusätzliche CPU und Arbeitsspeicher. Verzichten Sie daher auf die „Sortier“-Klauseln, wenn das Result Set nicht zwingend in einer bestimmten Reihenfolge ausgegeben werden muss.

Abschließend noch eine allgemeine Anmerkung zur Datenselektion. In gewöhnlichen CTEs versuchen wir stets bereits in den CTEs oder Views Zeilenselektionen unterzubringen, um in der Hauptabfrage die Datenlast zu reduzieren. Daraus könnte man ableiten, dass auch in rekursiven CTEs die Datenselektionen Bestandteil der rekursiven Abfrage sein sollten – also innerhalb der UNION ALL-Abfrage genutzt werden sollten. Bei rekursiven Abfragen müssen Sie mit den lokalen Selektionskriterien allerdings sehr vorsichtig sein, da sie gegebenenfalls die Ergebnismenge fehlerhaft beeinflussen.

Schauen Sie sich das nächste Beispiel an, dann werden Sie schnell verstehen, was gemeint ist: Gehen wir davon aus, dass Sie alle Flüge ab Chicago selektieren wollen, aber nicht die Verbindungen nach Frankfurt benötigen. Frankfurt als „Endstation“ soll ausgeschlossen werden.

```
WITH recursive Verbindungen
(Abflughafen, Zielflughafen, Anzahl_Fluege, Kosten, Flugroute)
AS

(
SELECT Departure, Arrival, 1, Price,
CAST(TRIM(Departure) || ' : ' || TRIM(Arrival) AS VARCHAR(2000))
FROM Flights
WHERE Departure = 'Chicago'

UNION ALL

SELECT Abflughafen, Arrival, Anzahl_Fluege + 1 , Price + Kosten,
CAST(TRIM(Flugroute) || ' : ' || TRIM(Arrival) AS VARCHAR(2000))
FROM Verbindungen, Flights F
WHERE Zielflughafen = Departure and Zielflughafen <> 'Frankfurt'
)
SELECT Abflughafen, Zielflughafen, Anzahl_Fluege, Kosten, Flugroute
FROM Verbindungen;
```

Die lokale Sektion wurde in die Rekursion integriert. Als Ergebnis erhalten Sie anschließend das folgende Result Set:

ABFLUGHAFEN	ZIELFLUGHAFEN	ANZAHL_FLUE...	KOSTEN	FLUGROUTE
Chicago	Miami	1	300	Chicago : Miami
Chicago	Frankfurt	1	480	Chicago : Frankfurt
Chicago	Lima	2	830	Chicago : Miami : Lima

Lokale Selektionskriterien

7.6.8
Seite 24

Frankfurt als Zielflughafen wurde nicht ausgeschlossen, aber alle Flüge über Frankfurt wurden nicht mehr selektiert.

Das Result Set sieht völlig anders aus, wenn die lokale Selektion in der Hauptabfrage verwendet wird:

ABFLUGHAFEN	ZIELFLUGHAFEN	ANZAHL_FLUE...	KOSTEN	FLUGROUTE
Chicago	Miami	1	300	Chicago : Miami
Chicago	Lima	2	830	Chicago : Miami : Lima
Chicago	Moscow	2	1060	Chicago : Frankfurt : Moscow
Chicago	Beijing	2	960	Chicago : Frankfurt : Beijing
Chicago	Vienna	2	680	Chicago : Frankfurt : Vienna
Chicago	Tokyo	3	1740	Chicago : Frankfurt : Moscow : Tokyo
Chicago	Hong Kong	4	2070	Chicago : Frankfurt : Moscow : Tokyo : Hong Kong

Selektionskriterien in der Hauptabfrage

Vielleicht sollten Sie es sich zur Regel machen, dass bei rekursiven Abfragen die lokalen Selektionen stets in der Hauptabfrage stattfinden – auch wenn Sie in diesen Fällen in Kauf nehmen müssen, dass das Result Set sequenziell gelesen wird.

7.6.10 Datensätze einfügen – Die Anweisung INSERT

Neue Datensätze werden in SQL mittels der Anweisung INSERT in eine Tabelle eingefügt. Dabei können wir Felder mit Inhalten füllen. Wichtig ist, dass die Angaben der Feldwerte den Felddefinitionen entsprechen. Das bedeutet, dass wir zum Beispiel in ein Zeichenfeld (Character) nur alphanumerische Inhalte einfügen und in ein Feld vom Typ Integer bestimmte Zahlenwerte angeben können.

Schauen wir uns einmal die Syntax der SQL-Anweisung INSERT an.

```
INSERT INTO tabellename ( Nr, NachName, VorName, GebJahr )
VALUES ( 1, 'Winzig', 'Willi', 1954 );
```

Wie man an dem Syntaxbeispiel sehen kann, besteht die INSERT-Anweisung genau genommen aus „INSERT INTO“. Dieser Anweisung folgt der einfache oder qualifizierte Name der Tabelle, in die wir den neuen Inhalt einfügen wollen.

Anschließend wird in Klammern die Liste der Felder bzw. Spalten eingetragen, deren Inhalt in der Tabelle eingefügt werden soll. In unserem Beispiel sind dies die Spalten Nr, NachName, VorName und GebJahr.

Zuletzt folgt nun die Werteliste der Felder. Dabei ist darauf zu achten, dass die Reihenfolge der Spaltenwerte den zuvor angegebenen Spaltennamen entspricht.

Beachten Sie die zuvor beschriebene Abhängigkeit der Feldarten und der zuzuweisenden Inhalte. Alphanumerische Daten sind, wie in unserem Beispiel anhand der Spalten NachName und VorName zu erkennen, bevorzugt in Hochkommata anzugeben. Zahlenwerte sind grundsätzlich ohne Hochkommazeichen zu verwenden.

Lassen Sie uns das Hinzufügen neuer Datensätze in eine Tabelle an unserer Beispieltabelle „ARTIKEL“ einmal genauer ansehen.

Ergänzung 16/2011

Um einen neuen Datensatz in die Tabelle einzufügen, verwenden wir zunächst als erstes Beispiel die folgende SQL INSERT-Anweisung:

```
INSERT INTO SQLDEMO.ARTIKEL (FIRMA)
  VALUES (('200'))
```

Anhand der zuvor gegebenen Informationen wissen wir, dass mit dieser Anweisung Folgendes ausgeführt wird:

1. INSERT INTO SQLDEMO.ARTIKEL
   Mit diesem Teil der Anweisung teilen wir SQL mit, dass wir in die Tabelle ARTIKEL in dem Schema SQLDEMO neue Datensätze hinzufügen wollen. Der Name der Tabelle muss der INSERT-INTO-Anweisung unmittelbar folgen. Außerdem muss die angegebene Tabelle in dem Kontext auch existieren. Sollten Sie eine nicht existierende Tabelle angeben, dann schlägt die Anweisung fehl.

2. Teilbereich Spalten (in unserem Beispiel „Firma“)
   Bei diesem Teilbereich der INSERT-Anweisung handelt es sich um die Angabe der einzufügenden Spaltenwerte. In diesem Beispiel wird folglich nur der Inhalt der Spalte „Firm“ in die Tabelle eingefügt. Die übrigen, hier nicht angegebenen Spalten werden mit den Standardwerten initialisiert und leer gelassen.

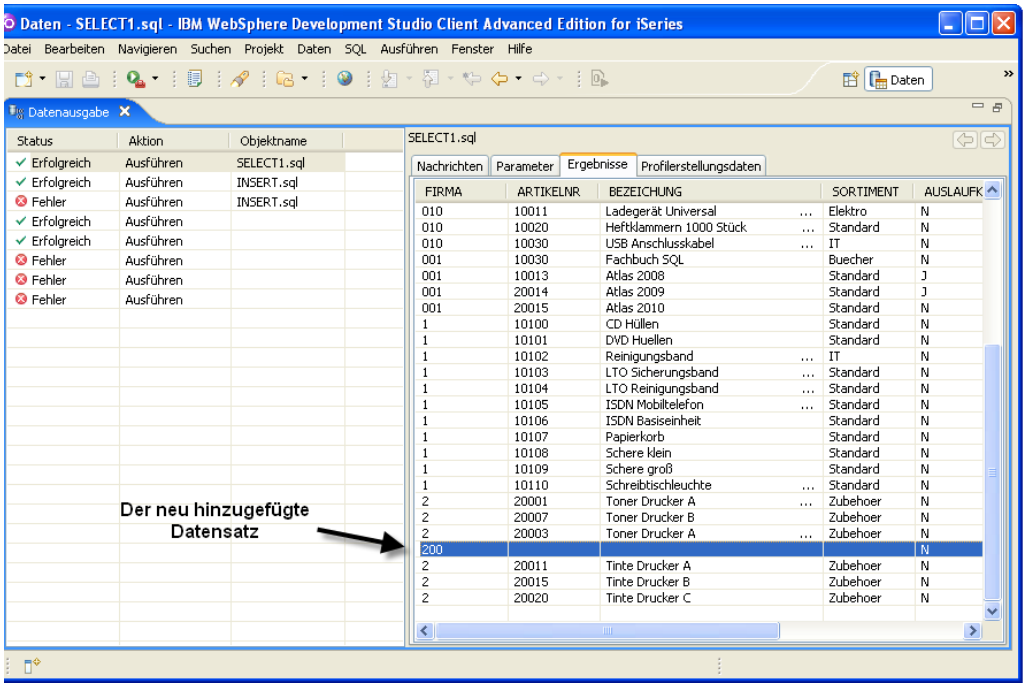
3. VALUES
   Mit dem Bereich „VALUE“ ordnen wir den zuvor aufgelisteten Spalten die einzufügenden Werte zu. In unserem Beispiel ist dies der Wert „200“. Da es sich um einen Alphawert handelt, wird dieser bevorzugt in Hochkommazeichen eingegeben. Die Werte sind dabei in Klammern zu fassen.

Schauen wir uns nun das Ergebnis der Ausführung der ersten INSERT-INTO-Anweisung einmal an.

Dazu verwenden wir die bereits bekannte SELECT-Anweisung in folgender Form:

```
SELECT *
  FROM SQLDEMO.ARTIKEL
```

Das Ergebnis finden Sie in der folgenden Abbildung.

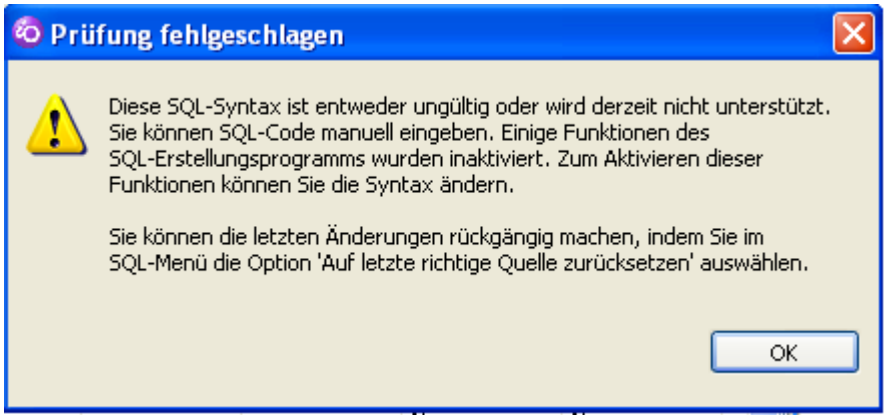


Ergebnis der INSERT-INTO-Anweisung

Durch die INSERT-INTO-Anweisung hat SQL in die Tabelle ARTIKEL einen neuen Eintrag – oder genauer: einen neuen Datensatz – der Tabelle hinzugefügt. Da wir lediglich die Spalte „Firma“ angegeben und mit einem Wert versehen haben, finden wir in dem neuen Datensatz auch nur diesen einen Spaltenwert. Alle übrigen Spalten sind mit dem für den jeweiligen Spaltentyp gebräuchlichen Standardwert initialisiert und damit leer.

Anmerkung

Wenn die SQL-Eingabe Fehler enthält, dann werden diese je nach verwendetem Werkzeug unterschiedlich ausgegeben. In der folgenden Abbildung sehen Sie einen Fehlerhinweis aus WDSC, der auf eine ungültige Syntax schließen lässt. Leider fehlen hier Details dazu, welcher Bereich der SQL-Anweisung fehlerhaft ist und nicht den SQL-Standards entspricht. Gute SQL-Eingabewerkzeuge beinhalten einen direkten Hinweis auf die fehlerhafte Stelle innerhalb der SQL-Anweisungen. Das macht besonders dann Sinn, wenn man mit wirklich komplexen und umfassenden SQL-Anweisungen arbeitet.



Syntaxfehler

Bei der Verwendung von INSERT INTO gibt es einige Regeln zu beachten, die ich Ihnen hier an dieser Stelle aufzeigen möchte. Hält man sich an diese Regeln, dann werden die neuen Datensätze erfolgreich in die Tabelle eingestellt. Anderenfalls wird ein Fehler ausgegeben. Ein Beispiel für eine solche Fehlermeldung sehen Sie in der vorhergehenden Abbildung.

Mit INSERT INTO wird ein neuer Datensatz in eine Tabelle eingefügt. Verfügt die Tabelle über einen eindeutigen Schlüssel, dann muss der neue Datensatz diese Eindeutigkeit wahren. Diese Eindeutigkeit bezieht sich nicht nur auf den eindeutigen Schlüssel (auch Primärschlüssel genannt), sondern auf alle Spalten, deren Werte als eindeutig anzugeben sind.

Die Spaltenwerte müssen den Spaltentypen entsprechen. Zahlenwerte müssen folglich mit Zahlen gefüllt werden; Datumspalten mit Datumswerten etc. Spalten können in SQL so definiert sein, dass diese Nullwerte gestatten. Wenn eine Spalte so definiert wurde, dass keine NULL-Werte zulässig sind, dann muss für diese Spalte auch ein Wert angegeben werden. Folglich sind für alle Felder, die keine NULL-Werte zulassen, Inhalte anzugeben – und sei es nur ein Unterlassungsleerwert.

Wenn eine dieser Bedingungen nicht erfüllt wird, schlägt die Ausführung der SQL INSERT-INTO-Anweisung fehl. Das hat zur Folge, dass der neue Datensatz nicht in die Tabelle gestellt wird. Damit sind folglich keine Datenfragmente durch eine fehlerhaft angegebene SQL INSERT-INTO-Anweisung möglich.

Jetzt ist es an der Zeit, uns ein wenig mit einer Besonderheit bei der Befüllung von alphanumerischen Spalten zu beschäftigen. Wie Sie an dem vorhergehenden Beispiel gesehen haben, wurde die Spalte „Firma“ mit dem Wert „200“ gefüllt. Die Spalte Firma ist in der Tabelle als 3-stelliges Charakterfeld definiert. Dieses kann natürlich neben reinen Zeichen auch reine Zifferfolgen aufnehmen, wie es in dem Beispiel der Fall ist.

Nun stellt sich die Frage, ob man diesen Wert auch anders hätte eingeben können. Wäre es möglich, den Wert 200 als Ziffernfolge ohne Hochkommazeichen anzugeben?

Machen wir die Probe und ändern die INSERT-INFO-Anweisung ein klein wenig ab, indem wir die Hochkommazeichen, die den Wert 200 einfassen, weglassen.

Damit sieht die INSERT-INFO-Anweisung nun folgendermaßen aus:

```
INSERT INTO SQLDEMO.ARTIKEL (FIRMA)
  VALUES ((200))
```

Führen wir die Anweisung aus, dann sehen wir, dass auch diese Form der Werteingabe von SQL akzeptiert und ausgeführt wird.

Das Ergebnis schaut genauso aus, wie im vorherigen Beispiel gezeigt:

SELECT1.sql

Nachrichten | Parameter | Ergebnisse | Profilerstellungsdaten

FIRMA	ARTIKELNR	BEZEICHUNG	SORTIMENT	AUSLAUFKZ	SPERRKZ	E
001	20015	Atlas 2010 ...	Standard	N	N	K
1	10100	CD Hüllen ...	Standard	N	N	K
1	10101	DVD Huelle...	Standard	N	N	K
1	10102	Reinigungs...	IT	N	N	K
1	10103	LTO Sicher...	Standard	N	N	K
1	10104	LTO Reinig...	Standard	N	N	K
1	10105	ISDN Mobilt...	Standard	N	N	K
1	10106	ISDN Basis...	Standard	N	N	K
1	10107	Papierkorb ...	Standard	N	N	K
1	10108	Schere klei...	Standard	N	N	K
1	10109	Schere gro...	Standard	N	N	K
1	10110	Schreibtisc...	Standard	N	N	K
2	20001	Toner Druc...	Zubehoer	N	N	K
2	20007	Toner Druc...	Zubehoer	N	N	K
2	20003	Toner Druc...	Zubehoer	N	N	K
200				N	N	
2	20011	Tinte Druck...	Zubehoer	N	N	K
2	20015	Tinte Druck...	Zubehoer	N	N	K
2	20020	Tinte Druck...	Zubehoer	N	N	K
200				N	N	
201				N	N	
200				N	N	
ABC				N	N	
ABC				N	N	
200				N	N	

Neuer Datensatz

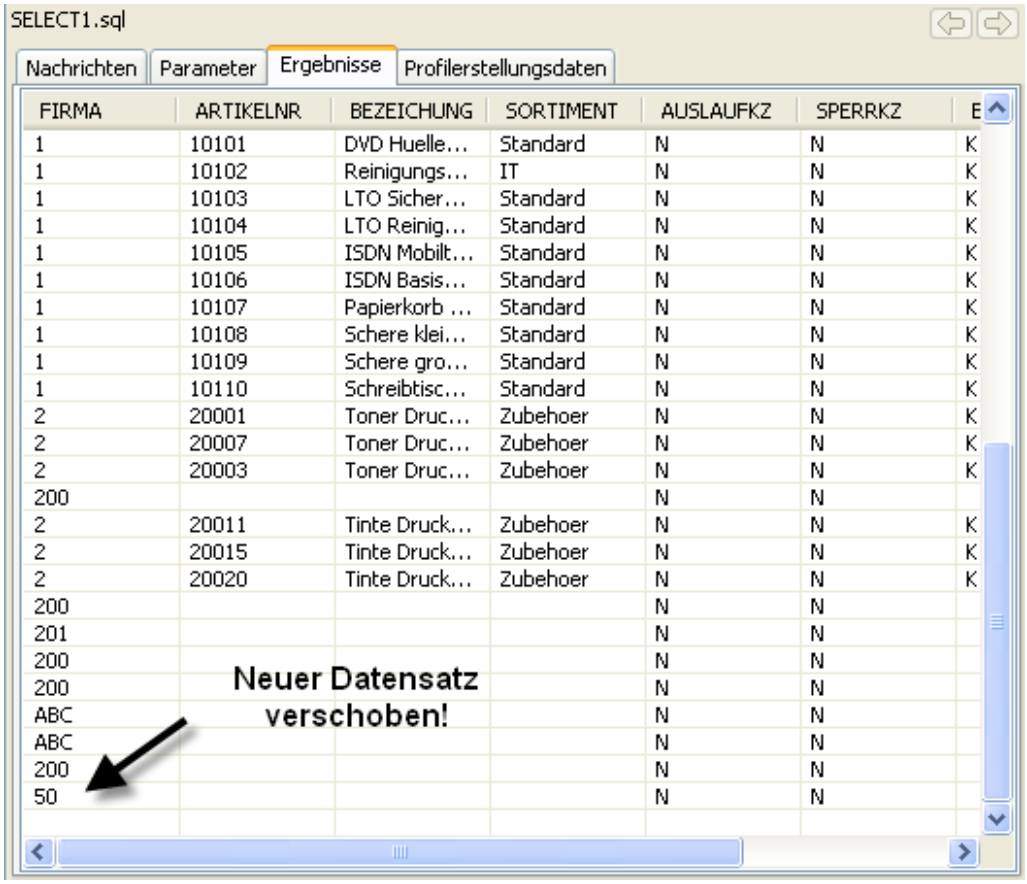
Wie Sie sehen können, besteht kein Unterschied zur Eingabe des rein numerischen Feldwerts in ein Zeichenfeld – unabhängig davon, ob er mit oder ohne Hochkommazeichen eingegeben wurde.

Jetzt kommt aber die Einschränkung: In unseren Beispielen haben wir mit einem Wert gearbeitet, der exakt der Spaltenlänge entspricht. Was passiert nun, wenn wir einen kürzeren Spaltenwert angeben?

Auch dies wollen wir anhand von zwei einfachen Beispielen prüfen. Dabei werden wir als Spaltenwert für die Firma jeweils 50 eingeben.

```
INSERT INTO SQLDEMO.ARTIKEL (FIRMA)
  VALUES ((50))
```

Schauen wir uns das Ergebnis direkt an:



FIRMA	ARTIKELNR	BEZEICHUNG	SORTIMENT	AUSLAUFKZ	SPERRKZ	E
1	10101	DVD Huelle...	Standard	N	N	K
1	10102	Reinigungs...	IT	N	N	K
1	10103	LTO Sicher...	Standard	N	N	K
1	10104	LTO Reinig...	Standard	N	N	K
1	10105	ISDN Mobilt...	Standard	N	N	K
1	10106	ISDN Basis...	Standard	N	N	K
1	10107	Papierkorb ...	Standard	N	N	K
1	10108	Schere klei...	Standard	N	N	K
1	10109	Schere gro...	Standard	N	N	K
1	10110	Schreibtisc...	Standard	N	N	K
2	20001	Toner Druc...	Zubehoer	N	N	K
2	20007	Toner Druc...	Zubehoer	N	N	K
2	20003	Toner Druc...	Zubehoer	N	N	K
200				N	N	
2	20011	Tinte Druck...	Zubehoer	N	N	K
2	20015	Tinte Druck...	Zubehoer	N	N	K
2	20020	Tinte Druck...	Zubehoer	N	N	K
200				N	N	
201				N	N	
200				N	N	
200				N	N	
ABC				N	N	
ABC				N	N	
200				N	N	
50				N	N	

Neuer Datensatz Firma 50

Wie Sie sehen, ist der Datensatz in die Tabelle eingefügt worden. Beachten Sie aber, dass der Wert linksbündig eingestellt ist. Durch das Einfügen als Ziffern hat SQL den neuen Spalteninhalt zeichenfeldtypisch linksbündig eingestellt.

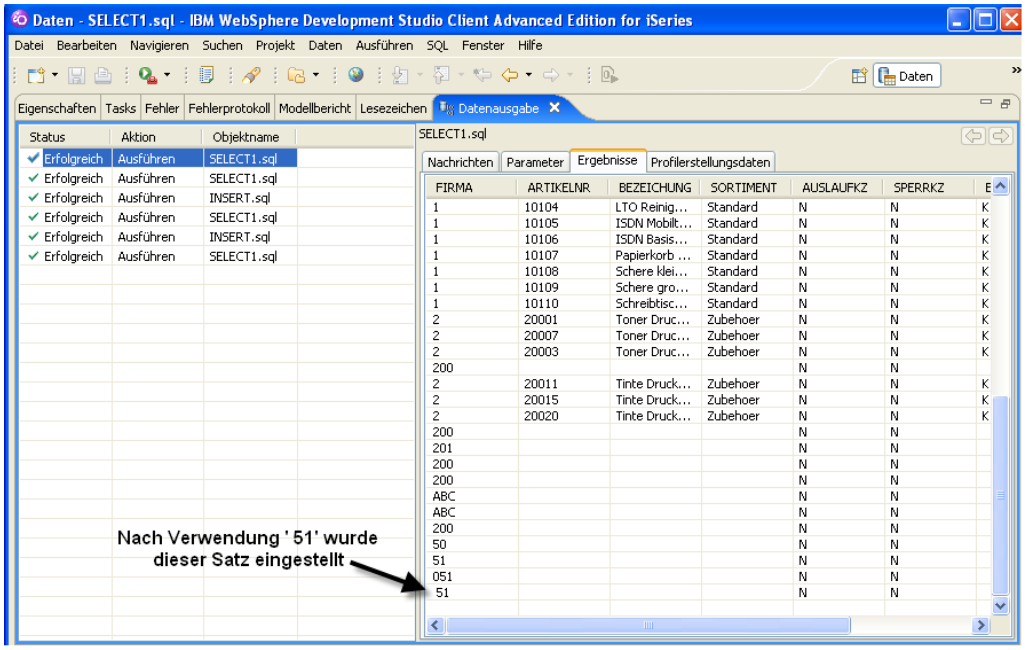
Denselben Effekt haben wir, wenn wir den 2-stelligen Feldwert in Hochkomma angeben.

```
INSERT INTO SQLDEMO.ARTIKEL (FIRMA)
  VALUES (('51'))
```

Soll der Wert nicht linksbündig, sondern rechtsbündig in dem Charakterfeld eingestellt werden, dann muss den SQL-Regeln gefolgt und die Eingabe in Hochkommazeichen eingeschlossen werden. Folglich wäre die korrekte Syntax für die Eingabe eines rechtsbündig eingestellten Zeichenwerts, der kürzer ist als die maximale Spaltenlänge:

```
INSERT INTO SQLDEMO.ARTIKEL (FIRMA)
  VALUES (('051'))
```

Führen wir diese Anweisung einmal aus und schauen uns das Ergebnis an:



Korrekt eingefügter Datensatz

Anmerkung

Wie bereits zuvor beschrieben, bietet es sich beim Arbeiten mit Zeichenspalten (Typ Charakter) an, deren Werte in Hochkommazeichen einzufügen und damit neue Spaltenwerte zu vergeben. Die Verwendung der Hochkommazeichen ist bei reinen Zeichenfolgen notwendig. Werden in ein Charakterfeld numerische Werte (wie zum Beispiel die Firmennummer 200) eingefügt, dann können die Hochkommazeichen zwar auch entfallen – es kann aber unter Umständen zu Ergebnissen kommen, die so nicht gewünscht sind. Um Verwechslungen und vor allem auch ungewollte Verschiebungen zu vermeiden, sollte man sich angewöhnen, alphanumerische Werte stets in Hochkommata gefasst einzugeben.

Nun haben wir einzelne neue Datensätze in die Tabelle eingefügt. Dabei haben wir uns lediglich mit einer Spalte bzw. deren Inhalt beschäftigt. Damit wurden neue Datensätze generiert und in einer quasi beliebigen Reihenfolge in die Tabelle eingestellt. Wie Sie anhand der Beispiele gesehen haben, wurden die neuen Datensätze an die bereits bestehenden angefügt. Auf die Reihenfolge der eingefügten Datensätze haben wir keinen Einfluss. Vielmehr obliegt es uns später, bei einem Zugriff auf die gewünschten Informationen die optimalen Angaben so zu machen, dass das System und SQL in der Lage sind, diese Informationen auf dem schnellstmöglichen Weg zu beschaffen. Grundsätzlich gilt aber, dass beim Einstellen von neuen Datensätzen in eine nicht geschlüsselte Tabelle die Reihenfolge nicht beeinflusst werden kann.

Abschließend noch ein Hinweis.

Die Angabe der einzelnen Spaltenwerte kann auch in der folgenden Form geschehen:

```
INSERT INTO SQLDEMO.ARTIKEL (FIRMA, ARTIKELNR, BEZEICHNUNG,
SORTIMENT,
  AUSLAUFKZ, SPERRKZ, BESCHAFFUNG, ANLAGEDATUM)
  VALUES ('500', DEFAULT, DEFAULT, DEFAULT, DEFAULT, DEFAULT,
    DEFAULT, DEFAULT)
```

Das Ergebnis dieser Anweisung liefert genau denselben Effekt wie die Anweisungen zuvor. Mit den Sonderwerten „DEFAULT“ weisen wir das System an, den für den jeweiligen Spaltentyp vorgegebenen Standardinitialisierungswert zu verwenden.

7.6.10.1 INSERT INTO – Spaltenwerte

Anhand der zuvor gezeigten Beispiele haben wir zwar neue Datensätze generiert, diese sind aber unvollständig – enthalten sie doch nur einen einzigen Spaltenwert – die Firmennummer. Letztlich kann das in der Praxis zwar auch mal vorkommen, es ist aber wahrscheinlicher, dass nicht nur ein Spaltenwert in einem Datensatz mit einem Wert versehen wird, sondern mehrere Spaltenwerte.

Die Beispiele zuvor sollten erst einmal dem Veranschaulichen der Besonderheiten bei der Verwendung von Zeichenfeldern dienen. Lassen Sie uns nun einen Blick auf die Füllung mehrerer Spalten innerhalb eines Datensatzes werfen.

Dazu erweitern wir die INSERT-INTO-Anweisung entsprechend. Die nachfolgende Beispielanweisung dient der Anlage eines neuen Datensatzes mit mehreren Spaltenwerten. Auch hier ist darauf zu achten, dass die angegebenen Spaltennamen auch tatsächlich in der Tabelle existieren. Ist dies nicht der Fall, wird die Ausführung der SQL-Anweisung mit einem Fehler abgebrochen werden, ohne dass der neue Datensatz in die Tabelle eingestellt wird.

Die Spaltennamen werden in diesem Beispiel in Form einer Feldnamenliste angegeben. Dabei handelt es sich um die Auflistung der Spaltennamen der Tabelle. Diese können, wie wir später noch sehen werden, in einer beliebigen Reihenfolge definiert werden.

Um zum Beispiel jede Spalte der Tabelle ARTIKEL mit einem Wert zu füllen, kann die folgende Anweisung verwendet werden.

```
INSERT INTO SQLDEMO.ARTIKEL (FIRMA, ARTIKELNR, BEZEICHNUNG,
SORTIMENT,
  AUSLAUFKZ, SPERRKZ, BESCHAFFUNG, ANLAGEDATUM)
  VALUES ('500', 50001, 'Schutzhuelle A5', 'Standard', 'N',
  'N',
    'K', DEFAULT)
```

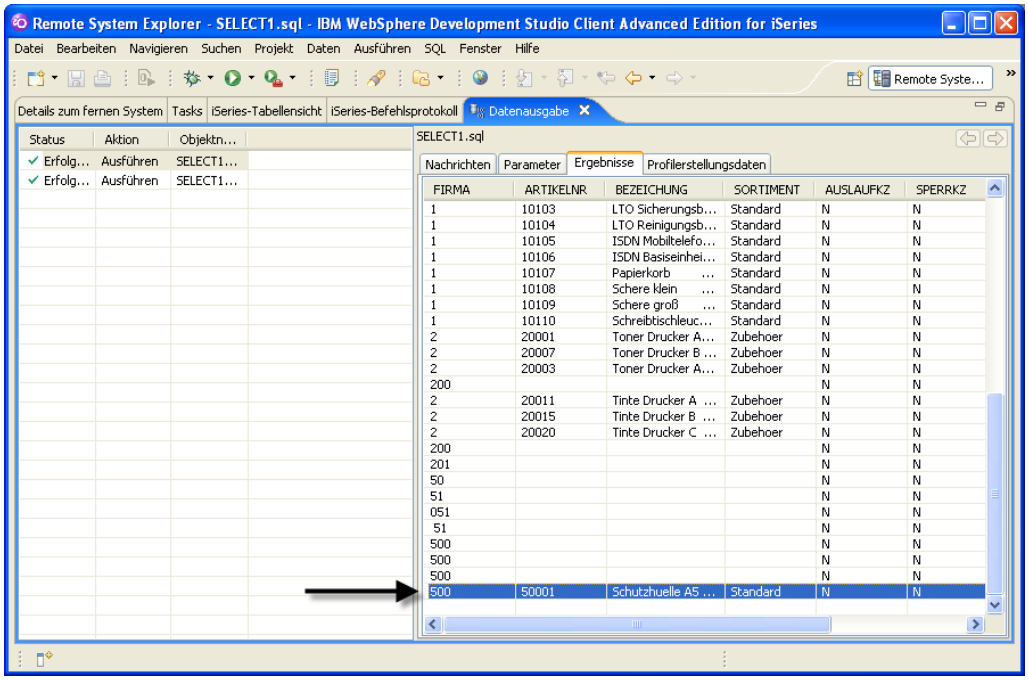
Bei dieser Anweisung wurden alle Spaltennamen angegeben.

Anmerkung

Die Reihenfolge der Felder muss bei der Eingabe nicht der Reihenfolge der Spalten innerhalb der Tabelle entsprechen. Über den Namen des Feldes bzw. der Spalte wird die betreffende Spalte eindeutig für SQL gekennzeichnet. Damit ist die Reihenfolge der einzufügenden Spaltenwerte durch die Namen der Spalten definiert. Dabei können Spalten, die nicht mit der INSERT-Anweisung eingefügt werden sollen, einfach weggelassen werden. Diese werden dann nicht bzw. mit dem Standardinitialisierungswert gefüllt. Wird die Reihenfolge der Spalten abweichend zu deren Reihenfolge innerhalb der Tabelle angegeben, dann muss die Reihenfolge der Werte zwingend denen der Spalten entsprechen!

In dem Abschnitt „VALUES“ ist der jeweilige Spaltenwert angegeben. Achten Sie dabei auf die korrekte Verwendung der Syntax für die unterschiedlichen Spaltentypen sowie auf die Reihenfolge, wenn Sie zuvor bei der Auflistung der zu ändernden Spalten von der Reihenfolge innerhalb der Tabelle abweichen bzw. einzelne Spalten in dem INSERT-Konstrukt nicht enthalten sind.

Das Ergebnis stellt sich dann wie folgt dar:



Vollständiger Datensatz

Ein Spaltenwert fehlt nun noch in unserem Beispiel, und zwar die Spalte „Anlagedatum“.

Bei der Füllung von Datumsfeldern ist besondere Vorsicht geboten. Selbiges gilt übrigens auch für Zeitangaben oder Zeitmarken.

Die Eingabe eines Datumswerts erfolgt ähnlich wie die eines Zeichenwerts in Hochkommazeichen.

Ein Beispiel für die Eingabe kann so ausschauen:

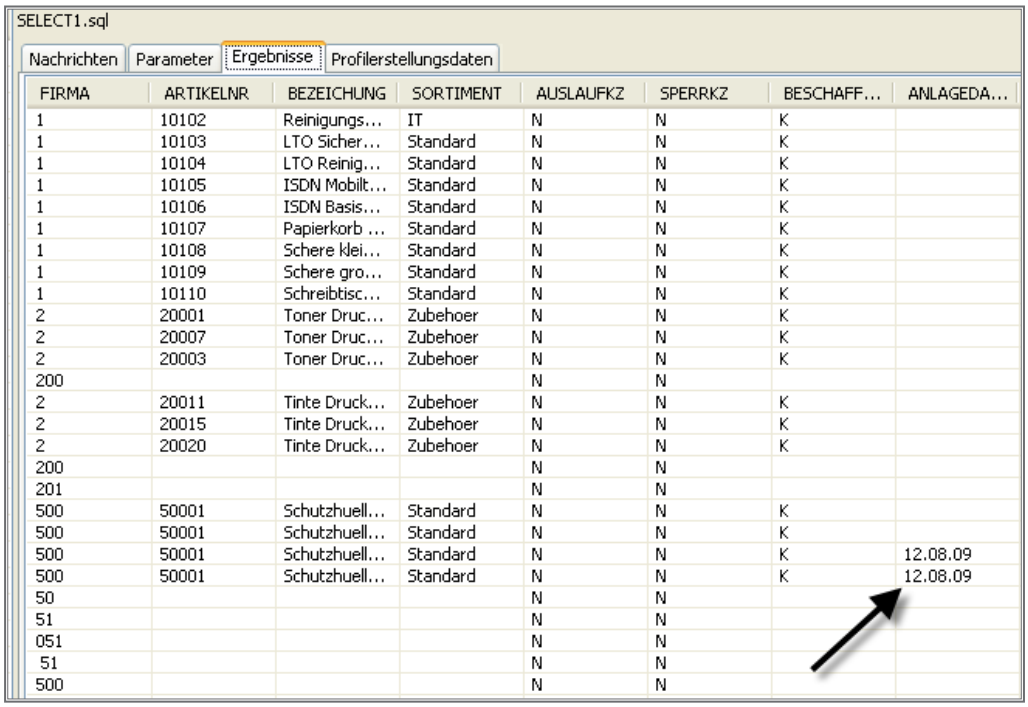
```
INSERT INTO SQLDEMO.ARTIKEL (FIRMA, ARTIKELNR, BEZEICHNUNG,
SORTIMENT,
  AUSLAUFKZ, SPERRKZ, BESCHAFFUNG, ANLAGEDATUM)
  VALUES ('500', 50001, 'Schutzhuelle A5', 'Standard', 'N',
  'N',
    'K', '12.08.09')
```

SQL setzt das Datum automatisch in das gewünschte Datumsformat um.

Bei der Verwendung von Datumsfeldern kann bei Bedarf auch mit Sonderwerten gearbeitet werden. Auf diese Weise lässt sich zum Beispiel das aktuelle Tagesdatum einstellen. Dazu verwendet man den Sonderwert „CURRENT DATE“, wie im folgenden Beispiel zu sehen ist:

```
INSERT INTO SQLDEMO.ARTIKEL (FIRMA, ARTIKELNR, BEZEICHNUNG,
SORTIMENT,
  AUSLAUFKZ, SPERRKZ, BESCHAFFUNG, ANLAGEDATUM)
  VALUES ('500', 50001, 'Schutzhuelle A4', 'Standard', 'N',
  'N',
    'K', CURRENT DATE)
```

Das Ergebnis dieser Anweisung:


SELECT1.sql

Nachrichten | Parameter | Ergebnisse | Profilerstellungsdaten

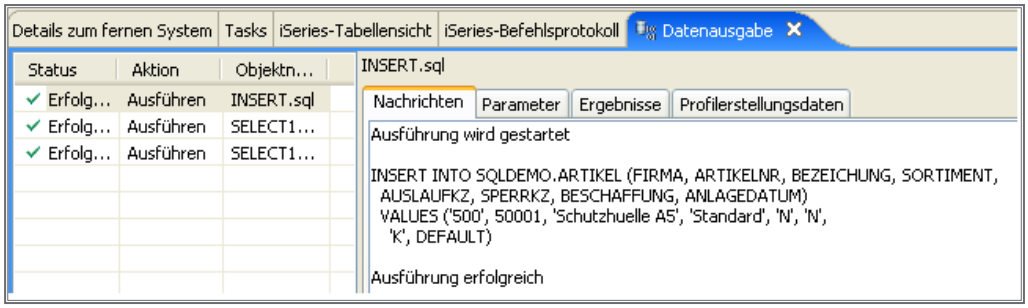
FIRMA	ARTIKELNR	BEZEICHUNG	SORTIMENT	AUSLAUFKZ	SPERRKZ	BESCHAFF...	ANLAGEDA...
1	10102	Reinigungs...	IT	N	N	K	
1	10103	LTO Sicher...	Standard	N	N	K	
1	10104	LTO Reinig...	Standard	N	N	K	
1	10105	ISDN Mobilt...	Standard	N	N	K	
1	10106	ISDN Basis...	Standard	N	N	K	
1	10107	Papierkorb ...	Standard	N	N	K	
1	10108	Schere klei...	Standard	N	N	K	
1	10109	Schere gro...	Standard	N	N	K	
1	10110	Schreibtisc...	Standard	N	N	K	
2	20001	Toner Druc...	Zubehoer	N	N	K	
2	20007	Toner Druc...	Zubehoer	N	N	K	
2	20003	Toner Druc...	Zubehoer	N	N	K	
200				N	N		
2	20011	Tinte Druck...	Zubehoer	N	N	K	
2	20015	Tinte Druck...	Zubehoer	N	N	K	
2	20020	Tinte Druck...	Zubehoer	N	N	K	
200				N	N		
201				N	N		
500	50001	Schutzhuell...	Standard	N	N	K	
500	50001	Schutzhuell...	Standard	N	N	K	
500	50001	Schutzhuell...	Standard	N	N	K	12.08.09
500	50001	Schutzhuell...	Standard	N	N	K	12.08.09
50				N	N		
51				N	N		
051				N	N		
51				N	N		
500				N	N		

Datumswerte

Tipp

Grundsätzlich sollte man darauf achten, dass je nach verwendetem SQL-Werkzeug die Rückgabe möglicher Fehler bei der Ausführung des SQL-Befehls mehr oder weniger komfortabel geschieht. Es ist eigentlich stets zu empfehlen, dass man nach der Erstellung einer neuen SQL-Anweisung deren Erfolg auch überprüft. Dies kann mittels der Ergebnisanzeige des SQL-Tools oder durch einen SELECT auf den neu eingefügten Datensatz geschehen.

Die folgende Abbildung zeigt exemplarisch die Erfolgsmeldung nach der INSERT-INTO-Ausführung in WDSC. Andere Editoren verfügen über ähnliche Auskunftssysteme, mit denen der Erfolg/Misserfolg der Ausführung schnell geprüft werden kann.



Ergebnisprotokoll der Befehlsausführung

Beim Einsatz dieser Prüfungen ist jedoch zu beachten, dass dabei lediglich SQL-Syntax geprüft wird. Selbstverständlich sind inhaltliche Plausibilitätsprüfungen nicht enthalten.

Anmerkung

INSERT INTO dient dem Einfügen eines einzelnen Datensatzes in eine Tabelle. Wie man anhand des vorhergehenden Beispiels gesehen hat, können dabei unterschiedliche Formen der Angabe von INSERT INTO genutzt werden. Dabei lassen sich komplett leere Datensätze, Datensätze mit Füllung einer oder mehrerer Spalten und noch viele Datensätze mehr anlegen. Wichtig ist, dass die Form der Spalten und deren Inhalte gewahrt werden, um später die Korrektheit der Daten gewährleisten zu können.

INSERT INTO kann, wie wir gesehen haben, nativ über ein SQL-Interface ausgeführt werden. Dabei sind dann die Erfolgskontrollen direkt möglich. Alternativ zum Einsatz der Dialogeingabe kann INSERT INTO natürlich auch in einem Programm automatisch ausgeführt werden. Dabei ist dann zu beachten, dass mögliche Änderungen am Tabellenaufbau zu Fehlern führen können, wenn bei der Verwendung von INSERT INTO die Feldnamenliste weggelassen wird. Durch das Hinzufügen neuer Spalten oder das Umdefinieren bereits bestehender Tabellenelemente kann die Ausführung einer INSERT-INTO-Anweisung ohne Feld-/Spaltennamenliste zu Fehlern führen. Deshalb gilt als Empfehlung, dass beim Einsatz von INSERT INTO dann eine Feldnamenliste verwendet wird, wenn diese Anweisung mehrfach genutzt werden soll. Damit ist in jedem Fall sichergestellt, dass die Reihenfolge der Werte und die Zuordnung zu den jeweiligen Spalten passen. Verzichten Sie auf den Einsatz dieses zugegebenermaßen bei der Erfassung lästigen Verfahrens der Feldnamenliste, dann kann im Extremfall in einer Spalte ein falscher Wert eingestellt werden.

7.6.10
Seite 14

7.6.11 Datensätze aktualisieren – Die Anweisung UPDATE

SQL bietet neben den unterschiedlichsten Möglichkeiten des Datenzugriffs natürlich auch Funktionen, mit denen bereits bestehende Daten verändert werden können.

Solche Veränderungen an den Inhalten von Tabellen werden für bestehende Datensätze mit der SQL-Funktion UPDATE durchgeführt. Dabei wird zunächst mit einer Zugriffsanweisung der gewünschte Datensatz eingelesen und anschließend die Anpassung der geänderten Spalten durchgeführt.

Die SQL-Anweisung UPDATE gestattet es uns, den Inhalt von Spalten eines Datensatzes zu verändern. Dabei kann wahlweise eine einzelne Spalte oder auch mehrere Spalten zeitgleich verändert werden. UPDATE ist aber nicht nur in der Lage, einen einzelnen Datensatz zu verarbeiten, sondern mit einer Ausführung die zutreffenden Datensätze auszuwählen und direkt zu verändern.

Nachfolgend finden Sie ein erstes Beispiel mit den Mindestangaben, die für eine erfolgreiche Ausführung der UPDATE-Anweisung erforderlich sind:

```
UPDATE sqldemo.artikel set Beschaff = 'X'
```

Mit dieser einfachen UPDATE-Anweisung veranlassen wir SQL dazu, alle Datensätze innerhalb der Tabelle ARTIKEL in dem Schema SQLDEMO zu verarbeiten und den Wert der Spalte „Beschaff“ auf „X“ zu setzen. Wie Sie sehen, gibt es in diesem einfachen Beispiel keine Einschränkungen auf die zu verarbeitenden Datensätze. Doch dazu später mehr.

So einfach letztlich die Ausführung von UPDATE ist, so konsequent ist die ausgeführte UPDATE-Anwendung auch. Ein einmal veränderter Datensatz bleibt verändert. Es sei denn, dass man Zusatzfunktionen einsetzt, mit denen einmal durchgeführte Änderungen bei Bedarf auch wieder rückgängig gemacht werden können. Solche Funktionen sollen an dieser Stelle jedoch nicht behandelt werden. Journalfunktionen finden Sie in einem weiteren Kapitel.

Anmerkung

Bei der Anwendung von UPDATE sind einige Besonderheiten zu beachten. Viele Anwendungen auf IBM i arbeiten auf Basis von Satzsperren. Damit soll verhindert werden, dass zeitgleich verschiedene Benutzer bzw. unterschiedliche Jobs denselben Datensatz im Zugriff haben und verändern. Solche Satzsperren werden auch bei der Ausführung von UPDATE in SQL angewendet. Dadurch kann es unter Umständen zu kurzzeitigen Satzsperren kommen, die dann andere Anwendungen/Anwender behindern können. Das sollte man besonders dann bedenken, wenn es um die Ausführung von Änderungen an größeren Datenbanktabellen geht.

Es gilt bei der Verwendung von UPDATE, dass die Anweisung sorgfältig editiert und im besten Fall getestet wird. Deshalb empfiehlt es sich insbesondere bei der Manipulation von sensiblen Daten, diese zuerst mit einer adäquaten SELECT-Anweisung zu selektieren, um sicherzustellen, dass die Selektion, die dann auch auf die UPDATE-Anweisung angewendet werden kann, die zutreffenden Datensätze liefert. Damit kann man ausschließen, dass versehentlich die falschen Daten mit UPDATE unwiderruflich verändert werden.

Schauen wir uns nun die Verwendung von UPDATE anhand eines einfachen Beispiels an. Dazu verwenden wir wieder die bereits bekannte Tabelle ARTIKEL, die sich in dem Schema SQLDEMO befindet.

Nachfolgend ein Beispielauszug mit dem Inhalt der Tabelle ARTIKEL.

FIRMA	ARTIKELNR	BEZEICHUNG	SORTIMENT	AUSLAUFKZ	SPERRKZ	BESCHAFF...
001	10001	Schnellhefter A4 ...	Standard	N	N	
001	10002	Büroklammern 1000 St...	Standard	N	N	
001	10003	Bleistift ...	Standard	N	N	
001	10004	Ringbinder ...	Standard	N	N	
001	10005	Radiergummi ...	Standard	N	N	
001	10006	Locher ...	Standard	N	N	
001	10008	Drucker Toner ...	Drucker	N	N	
001	10009	Mauspad ...	Computer	N	N	
001	10010	Post It Klein ...	Standard	N	N	
001	10011	Post It Mittel ...	Standard	N	N	
001	10012	Post It groß ...	Standard	N	N	
100	10010	DIN A4 Druckerpapier ...	Papier	N	N	
001	10015	DIN A4 Druckerpapier ...	Standard	N	N	
001	20001	Schere ...	Standard	N	N	
010	10011	Ladegerät Universal ...	Elektro	N	N	
010	10020	Heftklammern 1000 St...	Standard	N	N	
010	10030	USB Anschlusskabel ...	IT	N	N	
001	10030	Fachbuch SQL ...	Buecher	N	N	
001	10013	Atlas 2008 ...	Standard	J	J	
001	20014	Atlas 2009 ...	Standard	J	N	
001	20015	Atlas 2010 ...	Standard	N	N	

Beispielinhalt der Tabelle ARTIKEL

Es handelt sich wieder um den Artikelstamm. Darin befindet sich unter anderem das Feld „Beschaffung“, in dem gekennzeichnet werden soll, ob es sich bei dem Teil um ein Kaufteil oder eine Eigenfertigung handelt. Stellen wir uns vor, wir müssen den Inhalt dieser Spalte „Beschaffung“ verändern. Dies kann man leicht mit SQL durchführen.

Die Syntax der UPDATE-Anweisung ist wie folgt anzugeben:

UPDATE Tabellenname
Die UPDATE-Anweisung ist mit dem Tabellennamen anzugeben. Auch hier gilt, dass der Tabellenname zwar einfach angegeben werden kann und das System die passende Tabelle in der Bibliotheksliste sucht, es aber hinsichtlich der Flexibilität und Sicherheit empfohlen wird, den qualifizierten Tabellennamen zu verwenden. Dieser ist dann mit dem vorgestellten Schema anzugeben, in dem sich die Tabelle befindet.

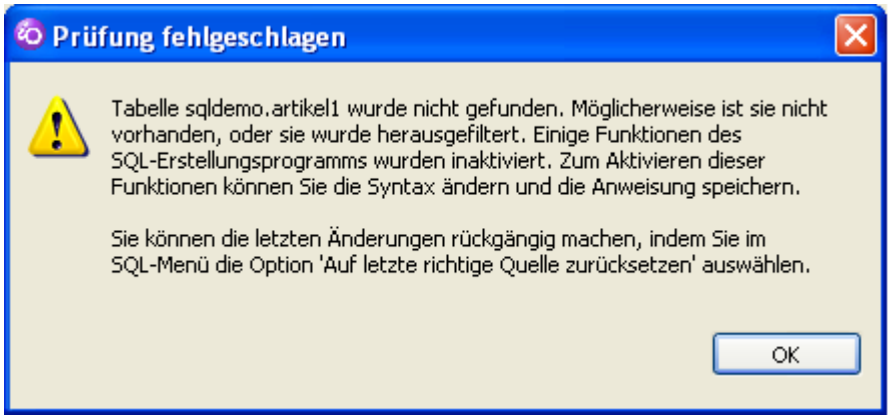
SET Spaltenname=Wert
Die SET-Klausel definiert die zu ändernden Spalten und deren Inhalte. Hier sind Mehrfachangaben für unterschiedliche Spalten möglich.

WHERE Bedingungsliste
Mit der WHERE-Klausel schränken wir die zu verarbeitenden Datensätze ein. Die Angabe der WHERE-Bedingung ist optional. Wird sie nicht angegeben, dann werden mit der UPDATE-Anweisung alle Datensätze innerhalb der Tabelle verarbeitet.

Die einfachste UPDATE-Anweisung besteht aus den folgenden Komponenten:

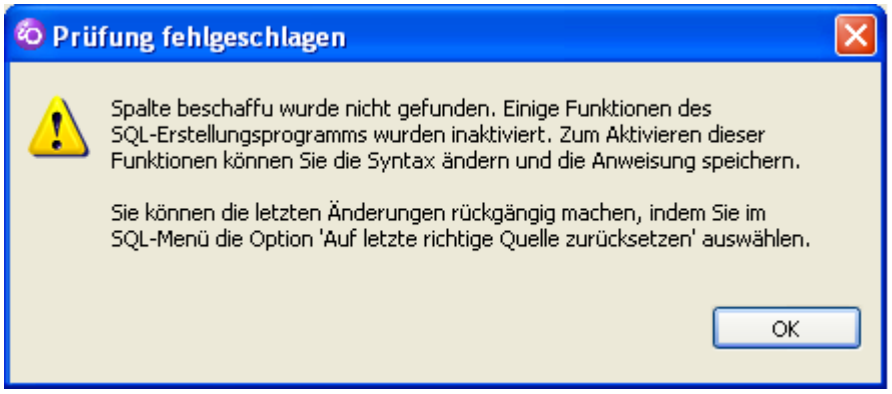
```
UPDATE schema.tabelle set spaltenname = Wert
```

Natürlich müssen für die Änderungen das Schema und die Tabelle auf dem System existieren. SQL-Editoren prüfen das Vorhandensein des Schemas bzw. der Tabelle. Existiert einer der beiden Teilbereiche nicht, wird eine Fehlermeldung ausgegeben, wie das folgende Beispiel zeigt.



Fehlerhafter Tabellenname

Wenn wir UPDATE-Anweisungen verwenden, dann muss der SQL-Interpreter wissen, welche Spalten verändert werden sollen. Die zu verändernden Spalten werden im Bereich SET der UPDATE-Anweisung angegeben. Dabei ist darauf zu achten, dass die Spaltennamen korrekt geschrieben werden. Fehlerhaft angegebene Spaltennamen führen zu einem Fehlschlagen der gewünschten Änderungen bzw. werden sie je nach verwendetem SQL-Eingabewerkzeug unmittelbar als Fehler erkannt und ausgegeben. Ein Beispiel für eine solche Fehlermeldung finden Sie in der nachfolgenden Abbildung.



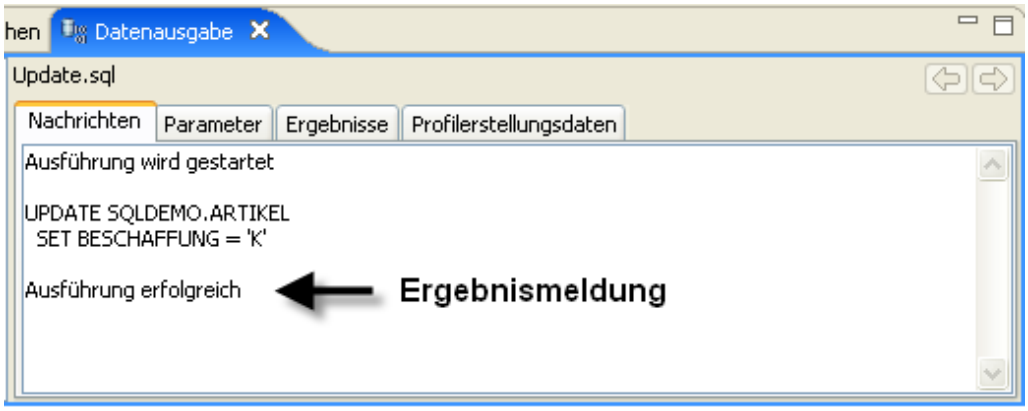
Fehlermeldung falscher Spaltenname

Wenn alle Angaben innerhalb der UPDATE-Anweisung korrekt waren, kann diese ausgeführt werden. Dies wollen wir anhand des ersten einfachen Beispiels einmal ausführen. Dabei ändern wir alle Spalten „Beschaffungsart“ so ab, dass diese als Wert ein „K“ beinhalten.

Dazu verwenden wir die folgende SQL-Anweisung:

```
UPDATE sqldemo.artikel set Beschaffung = 'K'
```

Nachdem die Anweisung ausgeführt wurde, meldet der SQL-Bereich die Beendigung. Je nach verwendetem SQL-Werkzeug können die Informationen über die UPDATE-Ausführung mehr oder weniger Informationen beinhalten.

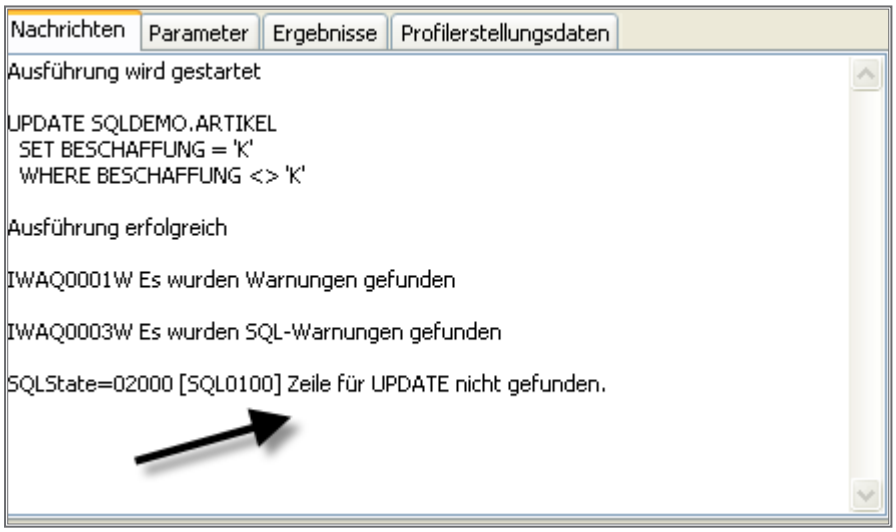


UPDATE wurde ausgeführt

Mit dieser Anweisung wurden alle Datensätze in der Tabelle ARTIKEL verarbeitet und der Wert der Spalte „Beschaffung“ auf „K“ gesetzt. Sollte in der Spalte „Beschaffung“ vorher ein anderer Wert gestanden haben, so wird dieser ohne weitere Prüfung unwiederbringlich überschrieben.

Diese Veränderung wurde für alle Datensätze durchgeführt – unabhängig davon, welcher Inhalt vorher in der Spalte gestanden hat.

Wenn keine für das UPDATE zutreffende Zeile gefunden wurde, dann erhalten wir einen Hinweis, wie er exemplarisch auch in der folgenden Abbildung zu sehen ist:



Keine Zeile für UPDATE gefunden

Damit haben wir also alle Datensätze innerhalb einer Tabelle verarbeitet und den Inhalt einer einzelnen Spalte geändert.

Bei Bedarf können alle oder mehrere Spalten mit einer einzigen UPDATE-Anweisung geändert werden. Dabei müssen lediglich in den SET-Angaben die einzelnen Spalten jeweils mit Zuweisungen versehen und voneinander mittels eines Kommas getrennt werden.

Ein Beispiel für die Anpassung mehrerer Spalten im Zuge der Ausführung einer einzelnen UPDATE-Anweisung kann wie folgt aussehen:

```
UPDATE SQLDEMO.ARTIKEL
SET BESCHAFFUNG = 'K', SPERRKZ = DEFAULT, AUSLAUFKZ = 'J'
```

Mit dieser Anweisung werden in der Tabelle ARTIKEL neben der Spalte Beschaffung auch die Spalten SPERRKZ und AUSLAUFKZ geändert. Wie Sie sehen, werden die einzelnen Spalten und deren Zuweisungen mit einem Komma getrennt.

Mögliche Werte, die mit UPDATE den Spalten zugewiesen werden, richten sich natürlich erst einmal nach dem Spaltentyp. Hier gelten dieselben Regeln, wie wir sie bereits bei der INSERT Anweisung kennen gelernt haben. Achten Sie deshalb auch beim Einsatz von UPDATE stets darauf, die korrekten Eingaben für die Spalten zu verwenden. Mögliche neue Werte können mit UPDATE zum Beispiel Konstanten, Ausdrucke oder die Schlüsselwörter NULL oder DEFAULT sein.

Natürlich ist die Zuweisung von neuen Spalteninhalten nicht auf Konstanten begrenzt. Vielmehr wird der Einsatz von Feldwerten oder Sonderwerten ebenfalls unterstützt.

Neben diesen Varianten der UPDATE-Anweisung gibt es noch weitere Formen, die im Wesentlichen mit der Einschränkung der zu verarbeitenden Datensätze in Verbindung stehen. Die Rede ist von der WHERE-Bedingung.

Wie bereits zuvor erwähnt, kann die UPDATE-Anweisung dazu genutzt werden, alle Datensätze innerhalb einer Tabelle zu verarbeiten. Das kann in dem einen oder anderen Fall sinnvoll und auch gewünscht sein, sicher wird es aber mindestens genauso normal sein, dass man nur bestimmte Daten innerhalb einer Tabelle ändern muss. Für solche Anforderungen bietet die UPDATE-Funktion die optionale WHERE-Bedingung.

WHERE dient der Selektion der zu verarbeitenden Datensätze. Die bereits bekannte SET-Klausel bezieht sich damit dann nicht mehr auf jede Zeile bzw. jeden Datensatz innerhalb der Tabelle, sondern nur noch auf das Subset an Zeilen, die durch die WHERE-Bedingungen ermittelt worden sind.

7.6.11.1 Bedingte Datenaktualisierungen

Neben den nun bereits bekannten Bestandteilen der UPDATE-Anweisung bildet die WHERE-Bedingung den Abschluss der Angaben in der SQL-Anweisung. Ein Beispiel finden Sie hier:

```
UPDATE SQLDEMO.ARTIKEL
SET BESCHAFFUNG = 'K'
WHERE BESCHAFFUNG <> 'K'
```

Nach der Angabe von WHERE geben wir die Bedingung(en) an. Diese besteht zum Beispiel aus einem Vergleich einer Spalte mit einem Wert, wie es das Beispiel zuvor zeigt. Dabei können wir die folgenden Vergleichsoperanten, die auch als „Prädikate“ bezeichnet werden, verwenden:

= (gleich)
Die Bedingung gilt als zutreffend, wenn der Inhalt der Spalte dem nach dem =-Zeichen angegebenen Wert entspricht.

<> (ungleich)
Die Bedingung trifft dann zu, wenn der Inhalt der Spalte nicht dem nach dem Vergleichsoperanten angegebenen Wert entspricht.

> (größer als)
Die Bedingung gilt als zutreffend, wenn der Inhalt der Spalte größer ist als der nach dem >-Zeichen angegebene Wert.

< (kleiner als)
Die Bedingung gilt als zutreffend, wenn der Inhalt der Spalte kleiner ist als der nach dem <-Zeichen angegebene Wert.

> = (größer gleich)
Die Bedingung gilt als zutreffend, wenn der Inhalt der Spalte größer als oder gleich wie der nach dem >=-Zeichen angegebene Wert ist.

<= (kleiner gleich)
Die Bedingung gilt als zutreffend, wenn der Inhalt der Spalte kleiner oder gleich dem nach dem <=-Zeichen angegebenen Wert ist.

LIKE
Mit dieser Form des Vergleichs können wir eine Abfrage nach einem Muster durch die Verwendung von Platzhaltern durchführen. Mit dieser Variante können wir zum Beispiel Textelemente innerhalb von Spaltenwerten ermitteln und für die SQL-Ausführung nutzen.

IS NULL
Mit dieser Angabe wird das Fehlen eines NULL-Werts in einer Spalte geprüft.

Anmerkung

Diejenigen Leser, die vielleicht schon Erfahrung mit SQL im Zusammenhang mit anderen relationalen Datenbanken gesammelt haben, sind vielleicht verwundert, einige Vergleichsoperanten in der Auflistung nicht wiederzufinden. Der Grund liegt darin, dass die DB2/UDB des IBM i leider einige vom allgemeinen SQL-Standard abweichende Eigenschaften aufweist. Eine der Eigenschaften ist hier die Unterstützung von Vergleichsoperanten. So unterstützt SQL auf IBM i zum Beispiel keine „Ungleich"-Angabe in Form des ansonsten möglichen !=, was der Angabe von <> entspricht.

Unter anderem werden auch die von anderen Datenbanken zum Teil bekannten Wildcards mit der DB2/UDB nicht unterstützt. Diese Besonderheiten gilt es zu beachten.

Mit den unterschiedlichen Vergleichsoperanten sind wir in der Lage, nahezu jede Datenkonstellation prüfen zu können. Dabei sind natürlich auch AND/OR-Kombinationen zulässig, mit denen bei Bedarf sehr komplexe Abfragen und Prüfungen von Bedingungen erreicht werden können.

Anmerkung

Eine Anweisung ohne zugehörige WHERE-Bedingung birgt eine gewisse Gefahr, da damit ausnahmslos alle Datensätze einer Tabelle verändert werden. Sorgen Sie deshalb dafür, dass vor der Ausführung der UPDATE-Anweisung ohne WHERE-Klausel sichergestellt ist, dass auch tatsächlich alle Datensätze verarbeitet werden sollen!

7.6.12 Datensätze löschen – Die Anweisung DELETE

Wir haben nun zwei SQL-Anweisungen kennengelernt, mit denen wir in der Lage sind, den Inhalt von Tabellen zu verändern. Die dritte SQL-Anweisung, mit der Änderungen an Tabelleninhalten durchgeführt werden können, ist die Löschanweisung DELETE.

DELETE wird dazu genutzt, um in einer Tabelle einen ganzen Datensatz bzw. eine Zeile löschen zu können. Das Löschen einzelner Spalteninhalte ist mit DELETE nicht möglich. Vielmehr ist bei einem solchen Vorhaben die Anweisung UPDATE zu verwenden und die gewünschte Spalte mit dem Initialisierungswert zu versehen.

Die DELETE-Anweisung kann global zum Löschen aller in der Tabelle befindlichen Zeilen angewendet werden, oder aber – qualifiziert angegeben – zum Entfernen einzelner spezieller Zeilen.

Die Syntax der DELETE-Anweisung ist wie folgt anzugeben:

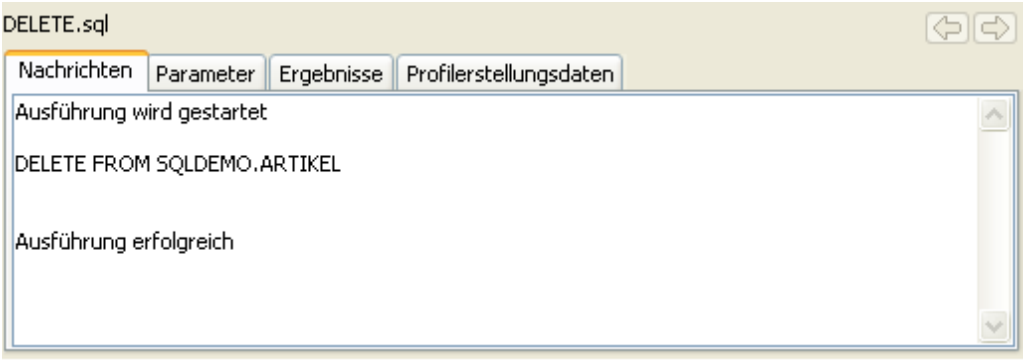
DELET FROM Tabellenname
Die DELETE-Anweisung ist mit dem Tabellennamen anzugeben. Auch hier gilt, dass der Tabellenname zwar einfach angegeben werden kann und das System die passende Tabelle in der Bibliotheksliste sucht, es aber hinsichtlich der Flexibilität und Sicherheit empfohlen wird, den qualifizierten Tabellennamen zu verwenden. Dieser ist dann mit dem vorgestellten Schema anzugeben, in dem sich die Tabelle befindet.

WHERE Bedingungsliste
Mit der WHERE-Klausel schränken wir die zu verarbeitenden Datensätze ein. Die Angabe der WHERE-Bedingung ist optional. Wird diese nicht angegeben, dann werden mit der DELETE-Anweisung alle Zeilen innerhalb der Tabelle verarbeitet. Die Verwendung der WHERE-Klausel ist auch beim Einsatz der SQL-Funktion DELETE optional.

Das folgende Beispiel zeigt die Syntax von DELETE zum Löschen aller Zeilen in einer Tabelle.

```
DELETE FROM SQLDEMO.ARTIKEL
```

Nach der Ausführung erhalten wir von SQL die Ausführungsquittierung:



Erfolgreiches Löschen aller Zeilen

Anmerkung

Wie bei der Verwendung von UPDATE so gilt auch für den Einsatz von DELETE die dringende Empfehlung, die zu löschenden Datensätze zunächst mittels einer vergleichbaren SELECT-Anweisung zu ermitteln. Sind die Auswahlkriterien der SELECT-Anweisung so, dass das gewünschte Ergebnis geliefert wird, dann können diese auch für die DELETE-Ausführung angewendet werden. Damit vermeidet man das ungewollte Löschen von Datensätzen, die unter Umständen nicht wiederhergestellt werden können.

Die DELETE-Anweisung kann ohne eine optional anwendbare WHERE-Klausel ausgeführt werden. Dabei ist zu beachten, dass alle Datensätze innerhalb der ausgewählten Tabelle gelöscht werden. Deshalb bietet es sich aus Sicherheitsgründen an, stets eine WHERE-Bedingung anzugeben, um das versehentliche Löschen aller Zeilen in der Tabelle zu verhindern.

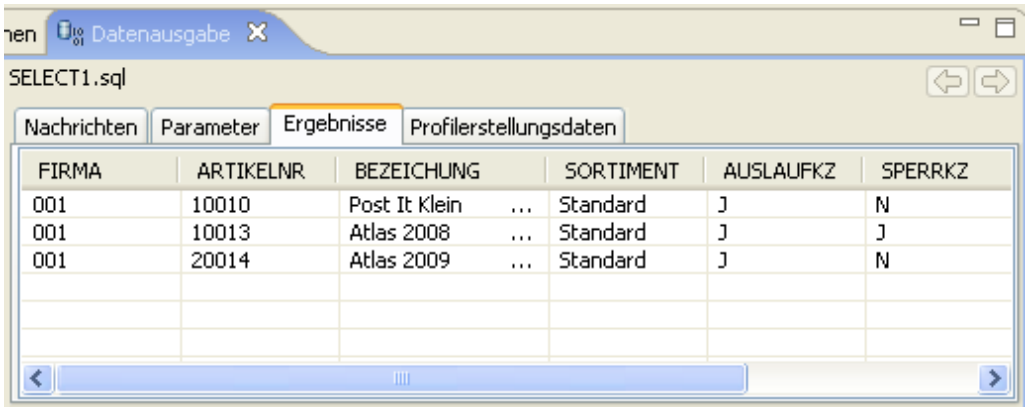
Besser ist folglich die Verwendung einer bedingten DELETE-Anweisung, mit der wir gezielt die zu löschenden Zeilen selektieren können.

Nehmen wir einmal an, wir wollen all diejenigen Zeilen einer Tabelle löschen, deren Spaltenwert einem bestimmten Inhalt entspricht. Bezogen auf unsere Beispieltabelle ARTIKEL kann dies das Löschen aller Zeilen sein, deren Spalte „Auslaufkennzeichen“ den Wert „J“ besitzt. Damit können wir die Artikelsätze löschen, die nicht mehr im aktuellen Sortiment enthalten sind.

Auch hier bietet sich zunächst eine SELECT-Anweisung an, damit wir auch sicher sein können, die korrekten Zeilen zu löschen.

```
SELECT *
  FROM SQLDEMO.ARTIKEL where auslaufkz = 'J'
```

Nehmen wir einmal an, das Ergebnis dieser SELECT-Anweisung liefert die folgenden Zeilen:



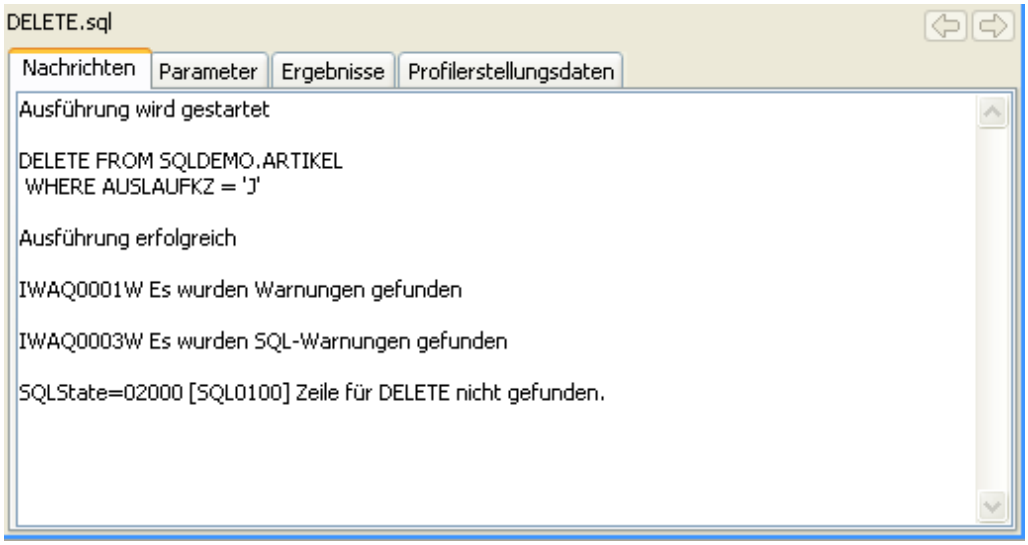
Zeilen mit Auslaufkennzeichen

Um diese Zeilen gezielt aus der Tabelle zu löschen, können wir die DELETE-Anweisung wie folgt codieren:

```
DELETE FROM SQLDEMO.ARTIKEL where auslaufkz = 'J'
```

Mit dieser Anweisung werden aus der Tabelle ARTIKEL alle Zeilen gelöscht, deren Spalte AUSLAUFKZ den Wert „J“aufweist.

Sollte mittels der verwendeten DELETE-Anweisung keine Zeile in der Tabelle gefunden und gelöscht werden, dann erhalten wir von SQL eine entsprechende Hinweismeldung.



Keine Zeilen zum Löschen gefunden

7.6.12
Seite 4

Anmerkung

Während auf anderen Plattformen zum globalen Löschen aller Zeilen in einer Tabelle ohne zusätzliche Protokollierungsfunktionen die SQL-Anweisung TRUNCATE genutzt werden kann, fehlt diese beim Einsatz von SQL auf IBM i. Dies als Hinweis für alle SQL-Kenner, die ihre Erfahrungen auf anderen Plattformen gesammelt haben.

7.7 Prozeduren

Hat man einige Zeit mit SQL gearbeitet, so wird man feststellen, dass es in einigen Situationen gewisse Defizite aufweist und manche Aufgabe mit einem einzigen SQL-Statement nicht gelöst werden kann. Diese Defizite lassen sich jedoch geschickt ausgleichen, indem man Stored Procedures (= gespeicherte Prozeduren) erstellt.

Welche Vorteile hat der Einsatz dieser Prozeduren?
Komplexe Aufgaben und Datenbankzugriffe können in den Prozeduren gekapselt und in Teilaufgaben zerlegt werden. Zudem können Variablen, Bedingungen und Schleifen zur Lösung der Aufgabenstellung verwendet werden, wodurch die Lösung übersichtlicher wird.

Die Datenbankzugriffe und die Abarbeitung der gesamten Aufgabe erfolgen direkt auf dem Server. So müssen keine Zwischenergebnisse zwischen Server und Client hin und her transportiert werden, das Netz wird entlastet und die Performance verbessert sich.

Die Prozedur kann von unterschiedlichen Anwendungsprogrammen aufgerufen werden. Anstatt in jedem Client-Programm die einzelnen Aktionen aufs Neue zu implementieren, können alle Clients eine gemeinsame Stored Procedure verwenden. Änderungen beim Datenbankzugriff oder bei der Modifikation der Aufgabenstellung müssen dann nur einmal anstatt x-mal durchgeführt werden.

Außerdem können die Zugriffsrechte auf die Datenbank einfacher geregelt werden, wenn jeder Benutzer nur über die Stored Procedures Daten auslesen oder modifizieren kann.

Und letztendlich wird es einfacher, die Datenbank zu portieren – falls Sie doch einmal die DB2 for i verlassen sollten. Denn aufgrund der Stored Procedures steht Ihnen eine eindeutige Datenbankzugriffsschicht zur Verfügung, die sich sehr leicht auch für andere Datenbankserver nutzen lässt.

Ergänzung 30./2014

7.7
Seite 2

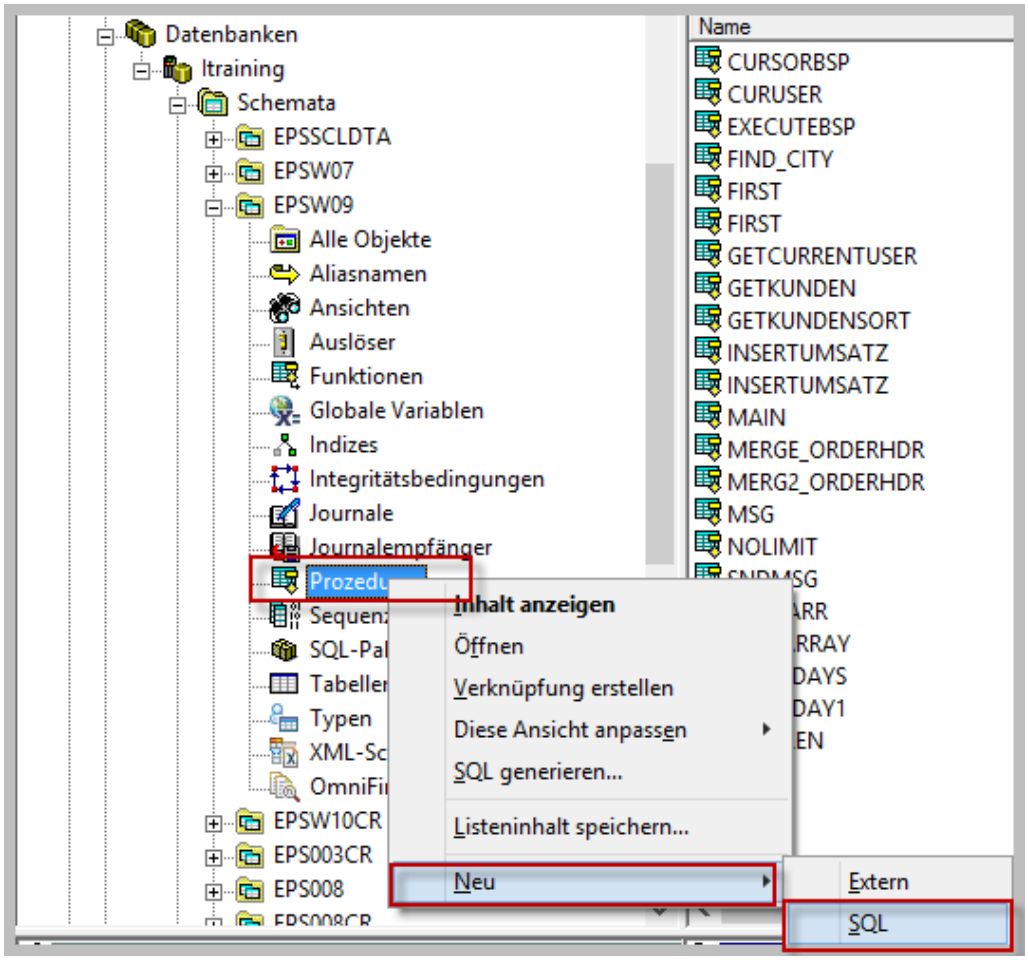
7.7.1 Prozeduren mit dem System i Navigator erstellen

Nun aber genug der Theorie. Erstellen wir einfach mal eine kleine Prozedur.

Bevor wir allerdings anfangen, brauchen wir noch eine Aufgabenstellung: Ich möchte die Prozedur WORKDAYS erstellen. Die Prozedur soll zwei Datumswerte erhalten und mir einfach die Anzahl Tage ermitteln, die zwischen den beiden Datumsfeldern liegt. Nach und nach können wir die Prozedur dann dahingehend erweitern, dass wir die Arbeitstage unter Berücksichtigung der Sonn- und Feiertage ermitteln.

Für unsere erste Prozedur nutzen wir zunächst das grafische Interface des System i Navigators.

Ich öffne also im System i Navigator den Bereich „Datenbanken ➜ Schemata ➜ Prozeduren ➜ Neu ➜ SQL“.



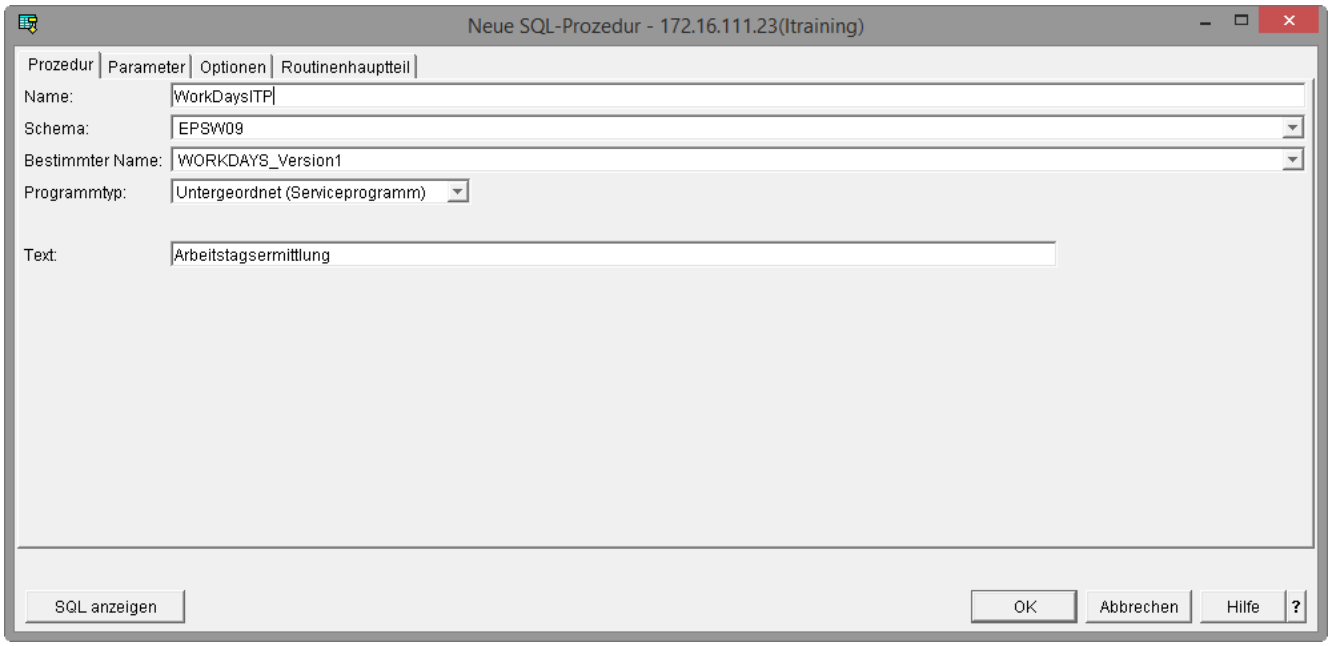
SQL-Prozeduren im System i Navigator

7.7.1
Seite 2

Auf diesem Weg erhalten Sie die Möglichkeit, neue SQL-Prozeduren oder neue externe Prozeduren zu erstellen. Worin liegt der Unterschied?

Bei externen Prozeduren ist das Prozedurprogramm in einer Sprache wie RPG, C, COBOL oder Java geschrieben. Um diese Programme in externen Umgebungen – also Weboberflächen, Java-Programmen oder Ähnlichem – nutzen zu können, erhalten Sie lediglich eine Prozedurbeschreibung mit diversen Attributen. Am eigentlichen HLL-Programm werden keine Änderungen vorgenommen.

SQL-Prozeduren sind dagegen vollständig in SQL codiert. Lassen Sie und also eine SQL-Prozedur erstellen. Sobald Sie Ihre Wahl (siehe Abbildung oben) getroffen haben, öffnet sich ein neuer Dialog, in dem wir die Grundeigenschaften festlegen:



Grundeigenschaften festlegen

Sie können in der Abbildung sehen, dass die SQL-Prozedur als Erstes einen allgemeinen und einen spezifischen Namen erhält. Worin liegt hier der Unterschied?

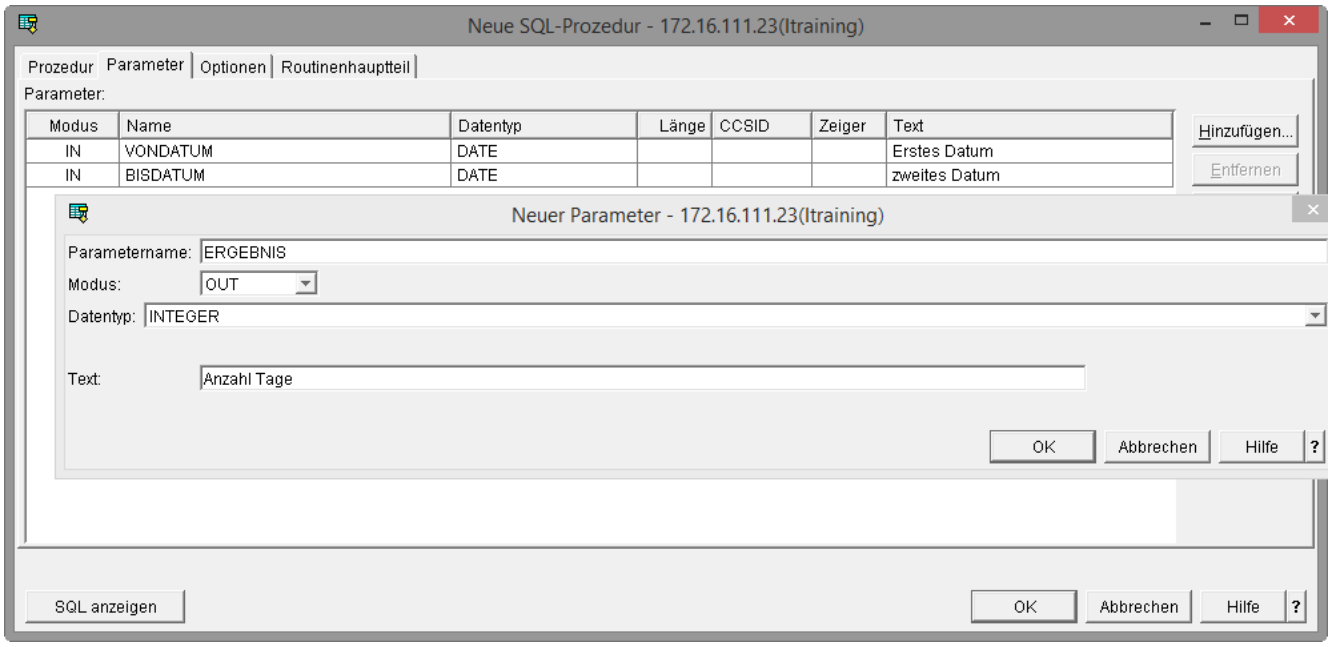
Der Prozedurname
Der Prozedurname ist der „offizielle“ Name der Prozedur. Es ist der Name, den Sie auch im Betriebssystem sehen werden. Achten Sie also darauf, dass Sie nicht zu lange Namen verwenden, wenn Sie dass Programm später im 5250-Umfeld noch eindeutig identifizieren wollen. Falls Sie mehr als zehn Zeichen für den Namen verwenden, nutzt das Betriebssystem lediglich die ersten fünf Zeichen und inkrementiert diesen Wert. Mein Programm wird folglich WORKD00001 heißen. Möglich ist die qualifizierte und unqualifizierte Adressierung des Programmobjekts. Der IBM i Server verhält sich hier konform: Geben Sie keine Bibliothek explizit an, wird das Programm in der aktuellen Bibliothek des Jobs gespeichert. Achtung, dies könnte die Bibliothek QGPL sein!

Der spezifische Name
Der spezifische Name ist der „Zweitname“. Er ermöglicht es, innerhalb eines Schemas identische Prozedurnamen mit unterschiedlicher Parameteranzahl zu verwenden. Man sagt: Die Prozedur wird überladen. Dazu mehr in Kapitel 7.7.6. Zum jetzigen Zeitpunkt sollten Sie sich lediglich merken, dass es besser ist, explizit eindeutige spezifische Prozedurnamen für Lösch- oder Berechtigungsanweisungen zu verwenden. Wenn der spezifische Name nicht explizit genannt ist, wird der Prozedurname selbst als spezifischer Name verwendet.

Programmtyp
Neben dem Prozedurnamen lege ich auch den Programmtyp fest. Die Eigenschaft „Programmtyp” bestimmt, ob das ausführbare Objekt ein Programm (*PGM) oder ein Serviceprogramm (*SRVPGM) sein soll. Serviceprogramme haben in der Regel die bessere Performance. Übersteigt allerdings die Anzahl der Parameter den Wert 400, wird automatisch ein Programmobjekt erstellt.

Abschließend folgt der beschreibende Text, um die Programmfunktion kurz zu dokumentieren.

Im nächsten Schritt sind die Parameter an der Reihe:



Das Register „Parameter”

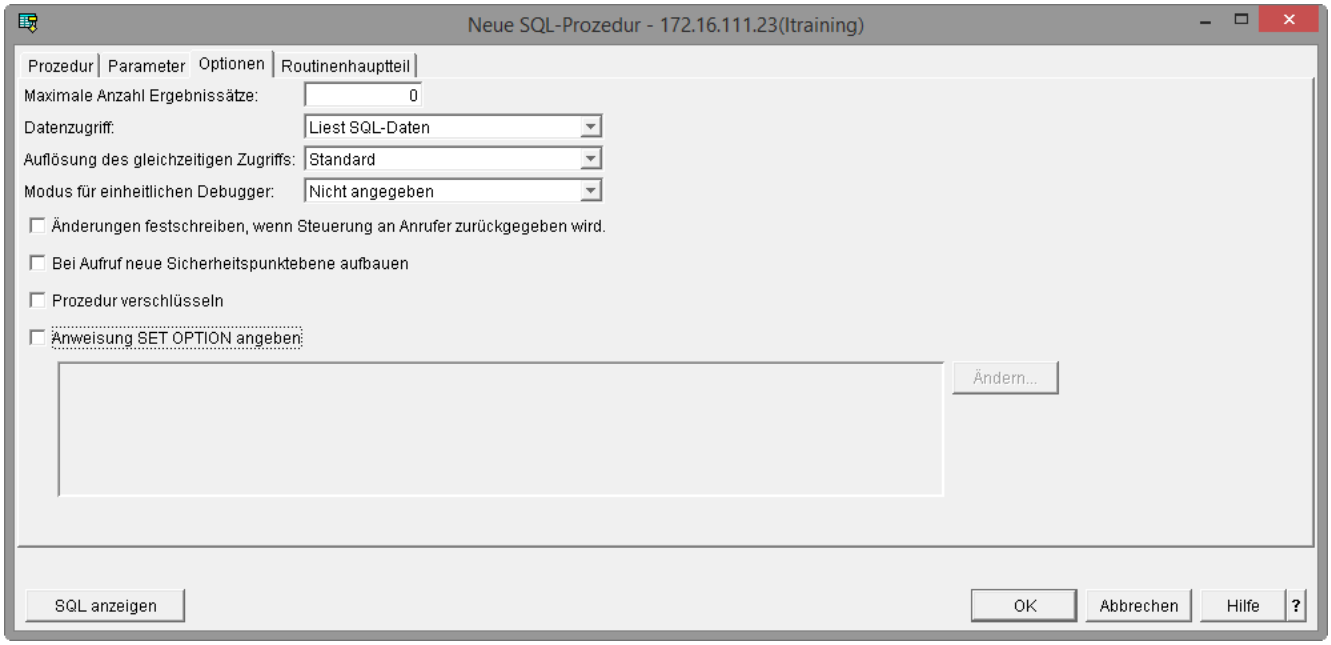
Wenn wir unserer Prozedur Parameter übergeben wollen, dann können wir jeden zu verwendenden Parameter mit dem Modifikator „IN“, „OUT“ oder „INOUT“ definieren, gefolgt von dem Parameter-namen und dem Typ des Parameters:

- IN — definiert einen Eingabeparameter.
- OUT — steht für die Definition von Ausgabeparametern.
- INOUT — definiert Ein- und Ausgabeparameter.

7.7.1
Seite 4

Meine Prozedur erhält die beiden INPUT-Parameter „VonDatum“ und „BisDatum“ und liefert mir das Ergebnis im OUTPUT-Parameter „Ergebnis“ zurück.

Neben diesen Basisangaben erhalten Prozeduren weitere optionale Eigenschaften. Diese finden wir im Register “Optionen”:

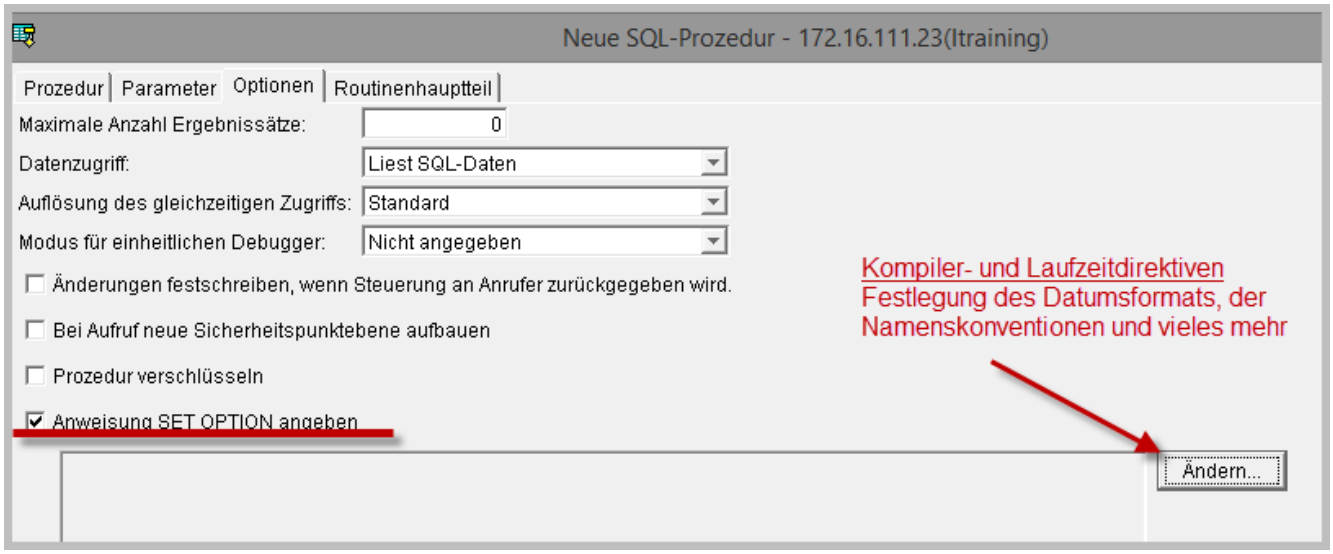


Das Register „Optionen”

Interessant ist für uns auf dieser Seite zunächst einmal die Eigenschaft „Datenzugriff”. Mit dieser Eigenschaft können wir festlegen, welche Aktionen die SQL-Prozedur ausführen darf. Schauen Sie sich die folgende Tabelle an, dann wird schnell klar, was gemeint ist:

Attribute	Beschreibung
Enthält SQL	Folgende Statements dürfen in der Prozedur verwendet werden: • CALL für Prozeduren, die ihrerseits NO SQL- oder CONTAINS SQL-Eigenschaften haben. • SET RESULT SET • SET and VALUES INTO (mit Variablen oder Konstanten) • COMMIT, ROLLBACK, SET TRANSACTION • CONNECT, DISCONNECT, SET CONNECTION
Liest SQL-Daten	Die Daten werden mit SQL gelesen (SELECT). Es können alle SQL-Statements verwendet werden – **außer:** COMMIT, ROLLBACK, SET TRANSACTION • CONNECT, DISCONNECT, SET CONNECTION • DELETE, INSERT, UPDATE • ALTER TABLE, COMMENT ON • CREATE, DROP, RENAME • GRANT, REVOKE • LABEL ON
Aktualisiert SQL-Daten	Die Daten werden gelesen und/oder mit SQL modifiziert.

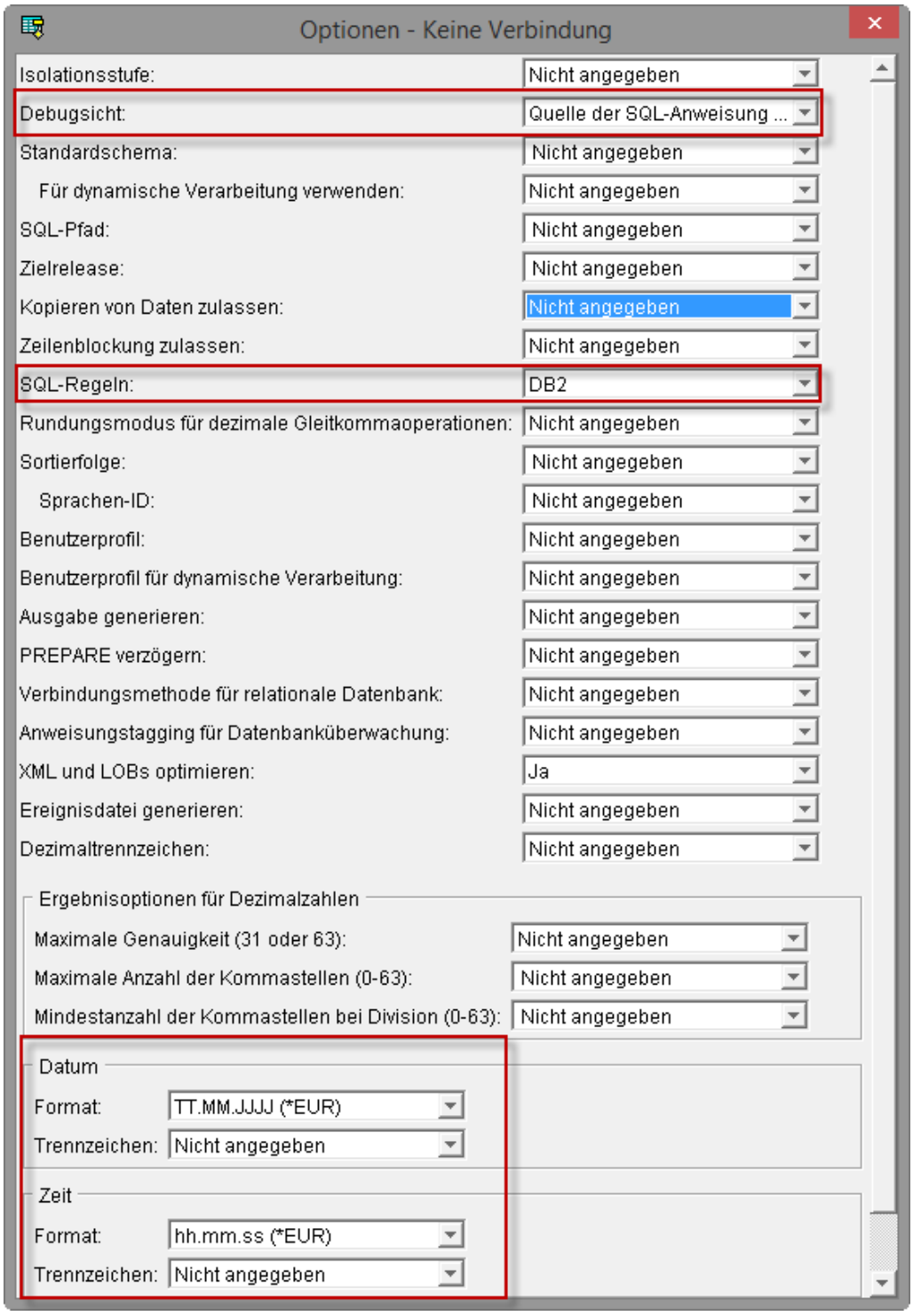
Zusätzlich zum Modus sollten Sie von Anfang an die richtigen Kompiler- und Laufzeitdirektiven – vergleichbar mit den H-Bestimmungen eines RPG-Programms – verwenden. Dazu wählen Sie die Anweisung SET OPTION und klicken auf den Button „Ändern“.



Kompiler- und Laufzeitdirektiven

7.7.1
Seite 6

Es folgt ein weiterer Dialog:



Prozeduroptionen

Hier sollten Sie darauf achten, dass Sie für den Debugger zusätzlichen Sourcecode generieren, solange Sie die Prozedur testen und ggf. auch mal mit einem Debugger ausführen wollen. Dieser Sourcecode wird übrigens in der Quellendatei QSQDSRC im Schema der Prozedur erzeugt. Als Namenskonvention verwenden Sie am besten immer DB2, und beim Datums- und Zeitformat bietet es sich an, mit dem europäischen Format zu arbeiten. Letzteres hängt natürlich mit Ihrem Firmenumfeld zusammen. Ausführliche Erläuterungen der weiteren Optionen finden Sie im entsprechenden SQL-Referenzhandbuch der IBM oder in deren Information Center.

So, nun haben wir es fast geschafft! Es bleibt noch das letzte Register „Routinenhauptteil". Hier codieren Sie die Logik der SQL-Prozedur. Sie müssen lediglich darauf achten, dass die SQL-Anweisungen mit BEGIN und END gekapselt werden, und im grafischen Interface darf die END-Anweisung nicht mit einem Semikolon beendet werden.

Meine Logik ist derzeit recht einfach gehalten:

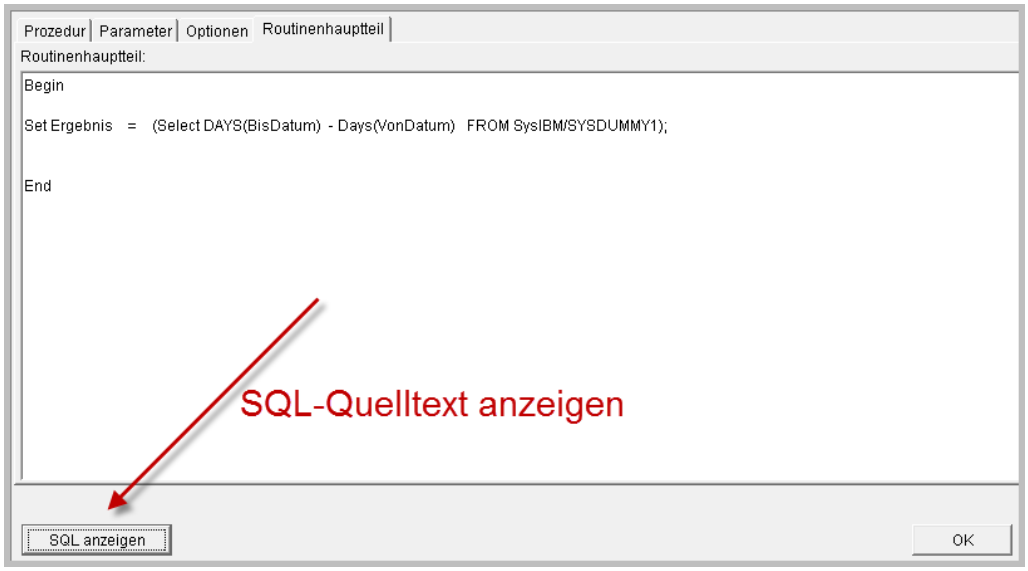
```
SET ERGEBNIS = (SELECT DAYS(BISDATUM) – DAYS(VONDATUM)
FROM SYSIBM/SYSDUMMY1);
```

Um die Datumsdifferenz zu ermitteln, konvertiere ich zunächst die beiden Datumsfelder in Tage und bilde die Differenz. Da ich keinen Datenbankzugriff benötige, nutze ich IBMs Dummytabelle SYSIBM/SYSDUMMY1, die aus einer Zeile mit einer Spalte besteht und somit keinen Overhead im Arbeitsspeicher erzeugen wird. Das Ergebnis der Abfrage übertrage ich mit einer SET-Anweisung in den Parameter „Ergebnis". Die Anweisung SET können Sie als RPG-Programmierer mit der Anweisung EVAL oder MOVE vergleichen oder auch mit der Anweisung CHGVAR im CL-Programm. Sie können einzelne oder mehrere Werte oder das Ergebnis einer SELECT-Anweisung übertragen. Schauen Sie sich einfach mal die nachfolgenden Beispiele an:

```
SET RGNR = 110;

SET REGION = (SELECT REGBEZ FROM REGIONEN WHERE REGION =
RGNR);

SET (RGNR, MAX_UMS) = (0, 0);

SET RGNR = 0,    MAX_UMS = 0;

SET VAR = VAR + 10;
```

Weiterführende Erläuterungen finden Sie natürlich wieder in IBMs SQL-Referenzhandbuch oder im IBM i Information Center.

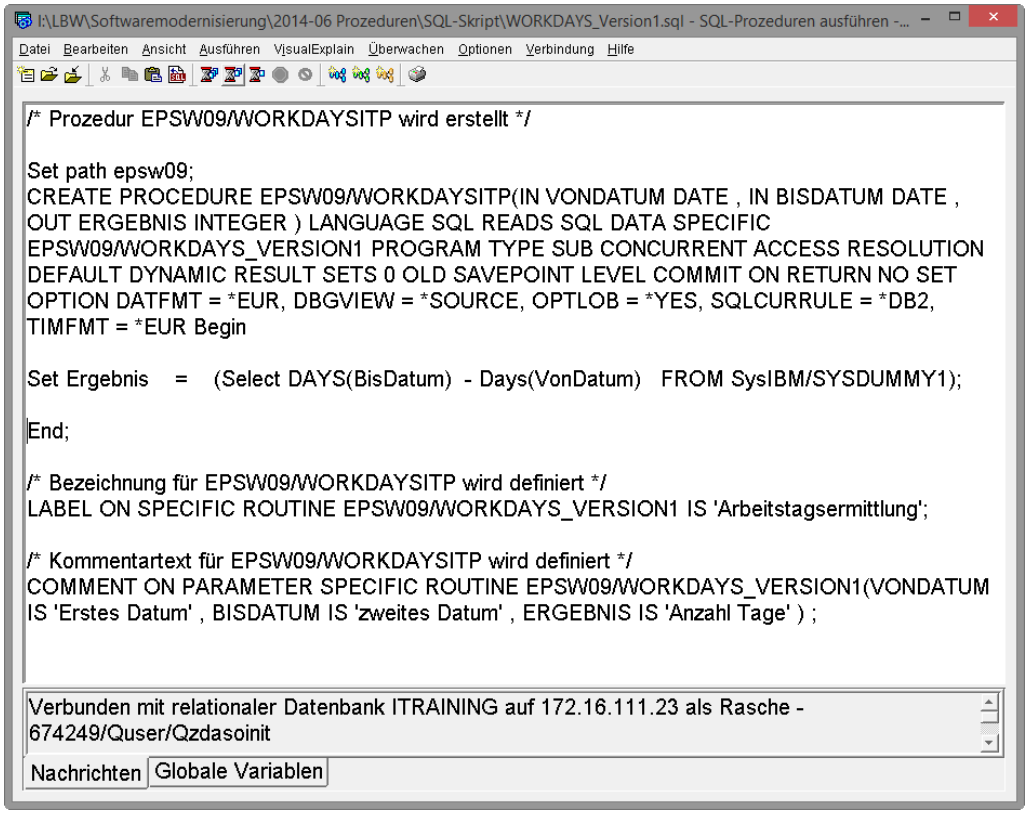
7.7.1
Seite 8



Das Register „Routinenhauptteil“

Natürlich könnte ich jetzt einfach auf den Button „OK“ klicken, um die Prozedur zu erstellen. Aber Sie müssen den Quelltext ja irgendwo speichern. Oder verzichten Sie bei Ihren RPG-Programmen etwa auch auf die Speicherung des Source-Codes? Außerdem ist die SQL-Programmierung mit dem grafischen Interface auf Dauer etwas mühsam, zumal es auch nicht alle Prozedureigenschaften unterstützt. Daher empfehle ich Ihnen, hier im letzten Register den SQL-Quelltext anzuzeigen und diesen dann aus der Prozedurumgebung des System i Navigators heraus in einer Quellendatei zu speichern. Anschließend können Sie das SQL-Skript mit dem Befehl RUNSQLSTM ausführen und die Prozedur auf diesem Weg erstellen. Eine andere Möglichkeit, das SQL-Skript zu bearbeiten und auszuführen, bietet RDI mit der Datenbankperspektive.

Schauen wir uns also jetzt den generierten Sourcecode an. Klicken Sie dazu auf „SQL anzeigen“. Daraufhin öffnet sich automatisch das Prozedurfenster des System i Navigators:



Der generierte Quelltext

Der generierte Quelltext ist leider nicht strukturiert und in dieser Form auch wenig verständlich. Sie sollten Ihn daher hier im Skriptfenster so aufbereiten, dass der Quelltext gut lesbar ist, bevor Sie ihn im IFS oder in einer Quellendatei einer beliebigen Bibliothek speichern.

Fazit: Um mal mit SQL-Prozeduren zu „spielen“ und den Einstieg zu finden, ist die grafische Darstellung sicherlich ganz gut, um aber professionell mit SQL-Prozeduren zu arbeiten, brauchen Sie mindestens das Skriptfenster des System i Navigators, noch besser aber einen leistungsfähigen Editor wie Sie Ihn beispielsweise im RDI finden.

7.7.1
Seite 10

7.7.2 Die Anweisung CREATE PROCEDURE

Ziehen wir unser Beispiel dazu heran, den Quellcode etwas genauer zu untersuchen.

```
CREATE PROCEDURE WORKDAYSITP …
```

Die Anweisung CREATE PROCEDURE ist die DDL-Anweisung, um eine Prozedur auf dem lokalen Server zu erstellen. Danach folgt der Prozedurname. Wenn Sie mit SQL-Skripten arbeiten, können Sie seit einiger Zeit die Klausel OR REPLACE verwenden. Die Klausel OR REPLACE wird ignoriert, wenn die Prozedur bislang nicht auf dem Server existiert. Wenn die Klausel OR REPLACE verwendet wird, wird die existierende Prozedur auf dem lokalen Server gelöscht, bevor die neue Definition implementiert wird. Bestehende Berechtigungen sind hiervon jedoch nicht betroffen. Die REPLACE-Klausel wurde mit Release i7.1 für ALIAS-Namen, Funktionen, Prozeduren, Sequenzen, globale SQL-Variablen und SQL-Views integriert.

Da ich das SQL-Skript zukünftig im Batch auf dem Server oder in der Prozedurumgebung ausführen möchte, ergänze ich die neue Klausel in meinem Skript und spare mir auf diesem Weg die Anweisung DROP PROCEDURE, mit der ich ansonsten die Prozedur löschen müsste, bevor ich Sie erneut erstellen kann.

```
CREATE OR REPLACE PROCEDURE WORKDAYSITP
(IN VONDATUM DATE ,
IN BISDATUM DATE ,
OUT ERGEBNIS INTEGER )
```

Unmittelbar nach dem Prozedurnamen folgt die Parameterdeklaration. Dabei ist darauf zu achten, dass nach dem Modus IN, OUT bzw. INOUT der Parametername und anschließend – durch Leerzeichen getrennt – der Parametertyp folgt. Mehrere Parameter werden durch Komma getrennt. Alle Parameterdeklarationen sind in Klammern eingeschlossen.

```
LANGUAGE SQL
READS SQL DATA
SPECIFIC WORKDAYS_VERSION1
PROGRAM TYPE SUB
```

7.7.2
Seite 2

Hinter den Parametern finden wir die Prozedureigenschaften. Programmiersprache ist SQL, die Prozedur liest lediglich Daten und wird als Serviceprogramm auf dem Server erstellt. Zusätzlich erhält die Prozedur den spezifischen Namen WORKDAYS_VERSION1. Diese Eigenschaften haben wir bereits in der grafischen Umgebung kennengelernt.

Dahinter finden wir weitere Eigenschaften, die automatisch generiert wurden:

```
CONCURRENT ACCESS RESOLUTION DEFAULT
DYNAMIC RESULT SETS 0
OLD SAVEPOINT LEVEL
COMMIT ON RETURN NO
```

Der Vollständigkeit halber sollen sie hier kurz erklärt werden, obwohl diese Eigenschaften für unsere Prozedur irrelevant sind und getrost aus dem Skript entfernt werden können, solange wir nicht mit Transaktionssteuerung und Cursorn in den Prozeduren arbeiten.

Die Eigenschaft CONCURRENT ACCESS RESOLUTION
Diese Eigenschaft spezifiziert den Umgang mit konkurrierenden Datenbankzugriffen in Multiuser-Umgebungen. Dabei werden folgende Modi unterschieden:

Wert	Bedeutung
WAIT FOR OUTCOME	Der Datenbank-Manager wartet auf die Freigabe der Zeilen.
USE CURRENTLY COMMITTED	Verwendet werden die aktuell freigegebenen Zeilen. Gesperrte Zeilen werden ignoriert.
DEFAULT	Es wird der Standard-Isolationslevel verwendet.

Die Eigenschaft DYNAMIC RESULT SETS x
Sie bestimmt die Anzahl Cursor, die an den Aufrufer zurückgeliefert werden. Maximal können 32.766 Result Sets an den Aufrufer zurückgeben werden. Diese Option wird dann eingesetzt, wenn eine Stored Procedure erstellt werden soll, die einen SQL-Cursor an die aufrufende Prozedur oder die Client-Anwendung zurückgibt. Wenn kein DYNAMIC RESULT SETS verwendet wird, werden alle geöffneten CURSOR an den Aufrufer zurückgegeben, sobald die Prozedur endet. Die Rückgabe der Cursor erfolgt in der Reihenfolge der Cursoröffnung.

OLD SAVEPOINT LEVEL / NEW SAVEPOINT LEVEL
Die Eigenschaft SAVEPOINT LEVEL bestimmt, ob die Prozedur eine neue Transaktion startet, wenn sie ausgeführt wird. OLD SAVEPOINT LEVEL ist der Standardwert. Die Prozedur befindet sich dann in der Transaktion des Aufrufers. Bei NEW SAVEPOINT LEVEL wird ein neuer Sicherungspunkt erstellt.

COMMIT ON RETURN NO/YES
Standardmäßig veranlasst der Datenbankmanager kein COMMIT beim Verlassen der Prozedur (COMMIT NO). Hingegen veranlasst der Datenbankmanager automatisch ein COMMIT, wenn die Prozedur unter Verwendung der Klausel COMMIT YES erfolgreich beendet wird. Falls die Prozedur einen geöffneten Cursor an den Aufrufer übergibt, muss der Cursor mit dem Zusatz WITH HOLD definiert sein, damit das Result Set auch noch nach dem COMMIT verwendet werden kann.

Es gibt noch viele weitere Prozedureigenschaften, die wir setzten können, aber nicht verwenden müssen. Hier einige Beispiele, die ich gleich in meiner Prozedur verwenden werde:

NOT DETERMINISTIC /DETERMINISTIC
Diese Option ist ein wichtiger Hinweis für SQL. Damit wird angezeigt, dass die Prozedur, wenn sie mehrfach mit identischen Parameterwerten aufgerufen wird, unterschiedliche Ergebnisse zurückgeben wird. Wenn eine Prozedur als DETERMINISTIC deklariert wird, wird sie in nachfolgenden Aufrufen mit identischen Aufrufparametern nicht ausgeführt. Stattdessen werden die Ergebnisse von früheren Aufrufen in der Annahme genutzt, dass sich die zugrunde liegende Datenbasis nicht geändert hat. Standard: NOT DETERMINISTIC.

CALLED ON NULL INPUT zeigt an, dass die Prozedur auch dann aufgerufen werden soll, wenn NULL-Werte als Eingabeparameter übergeben werden.

INHERIT SPECIAL REGISTERS
Sonderregister des Aufrufers werden an diese Prozedur vererbt. CURRENT DATE, CURRENT SERVER, CURRENT PATH, CURRENT SCHEMA sind Beispiele für Sonderregister.

7.7.2
Seite 4

Nach den Prozedureigenschaften folgt die Anweisung SET OPTION, die Sie vielleicht kennen, wenn Sie embedded SQL im RPG- oder Cobol-Programm verwenden. Die verwendeten Direktiven haben wir bereits besprochen:

```
SET OPTION
       DATFMT = *EUR,
       DBGVIEW = *SOURCE,
       OPTLOB = *YES,
       SQLCURRULE = *DB2,
       TIMFMT = *EUR
```

Nach der Anweisung SET OPTION folgt der Quellcode, der durch den BEGIN … END-Block eingeschlossen wird.

```
BEGIN
SET ERGEBNIS = (SELECT DAYS(BISDATUM) – DAYS(VONDATUM) FROM
SYSIBM/SYSDUMMY1);
END;
```

Und so sieht das SQL-Skript komplett aus:

```
CREATE OR REPLACE PROCEDURE WORKDAYSITP
(IN VONDATUM DATE ,
IN BISDATUM DATE ,
OUT ERGEBNIS INTEGER )

LANGUAGE SQL
READS SQL DATA
SPECIFIC WORKDAYS_VERSION1
PROGRAM TYPE SUB
CONCURRENT ACCESS RESOLUTION DEFAULT
DYNAMIC RESULT SETS 0
OLD SAVEPOINT LEVEL
COMMIT ON RETURN NO
SET OPTION
       DATFMT = *EUR,
       DBGVIEW = *SOURCE,
       OPTLOB = *YES,
       SQLCURRULE = *DB2,
       TIMFMT = *EUR
BEGIN
SET ERGEBNIS = (SELECT DAYS(BISDATUM) – DAYS(VONDATUM) FROM
SYSIBM/SYSDUMMY1);
END;
```

7.7.3 Objektadressierung in Prozeduren

Nun soll die Prozedur in der Bibliothek EPSW09 erstellt werden. Es wäre ein Leichtes, einfach den Bibliotheksnamen im Skript anzugeben. Aber wenn die Prozedur zu Testzwecken zunächst in einer Testbibliothek erstellt werden muss, bedeutet das, dass wir das Skript ändern müssen, bevor wir die Prozedur in der Echtumgebung erstellen. Warum sollten wir das tun? Wir nutzen doch auch ansonsten immer die Suchliste, um unsere Objekte zu qualifizieren. Beschäftigen wir uns also ein wenig mit der Objektadressierung im SQL-Umfeld.

SQL unterscheidet grundsätzlich zwischen Datei- bzw. Tabellenzugriffen sowie Zugriffen auf andere Datenbankobjekte, wie beispielsweise Prozeduren, Funktionen oder globale Variablen. Wir haben – wenn Sie so wollen – zwei unterschiedliche „Bibliothekssuchlisten". Nur nennt man diese Suchpfade nicht Bibliothekslisten, sondern SONDEREGISTER.

Alle Dateiobjekte werden über das aktuelle Schema gesucht. Welches derzeit das aktuelle Schema ist, zeigt stets das Sonderregister CURRENT SCHEMA:

```
SELECT CURRENT SCHEMA FROM SYSIBM/SYSDUMMY1;
```

Alle anderen Objekte werden über die aktuelle Pfadeinstellung gesucht. Auch hier lässt sich die Einstellung schnell mittels des Sonderregisters CURRENT PATH ermitteln:

```
SELECT CURRENT PATH FROM SYSIBM/SYSDUMMY1;
```

Abhängig von der gewählten Namenskonvention werden die beiden Sonderregister allerdings unterschiedlich initialisiert, wenn der Job auf dem Server initiiert wird.

Falls Sie mit der Namenskonvention SQL arbeiten, wird das in den Treiberdefinitionen angegebene Standardschema verwendet. Falls keine Einträge im Treiber gemacht wurden, wird als Standardschema der Benutzername verwendet. Sie können diese Einstellungen allerdings immer mit den folgenden Anweisungen übersteuern:

```
SET SCHEMA EPSW09;
SET PATH EPSW09;
```

Etwas aufpassen muss man in diesem Kontext mit der Anweisung SET SCHEMA = DEFAULT. Die Anweisung verwendet den aktuellen Benutzernamen für das Sonderregister CURRENT SCHEMA, wenn die Namenskonvention SQL gewählt wurde.

Ergänzung 30/2014

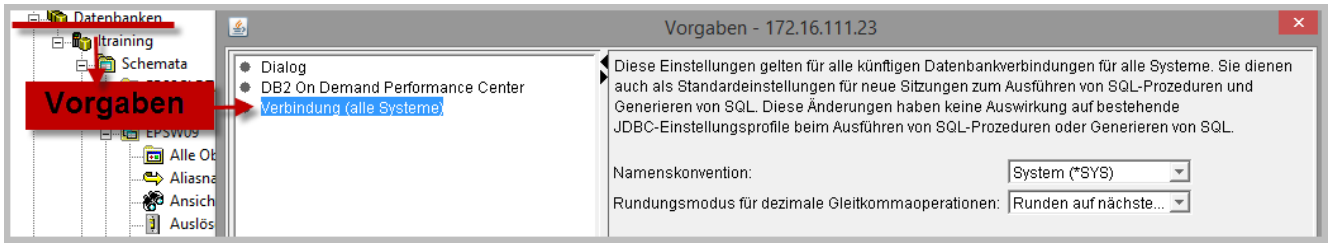
7.7.3
Seite 2

Eigentlich bietet es sich jedoch an, die Systemnamenskonvention (naming=system) zu verwenden.

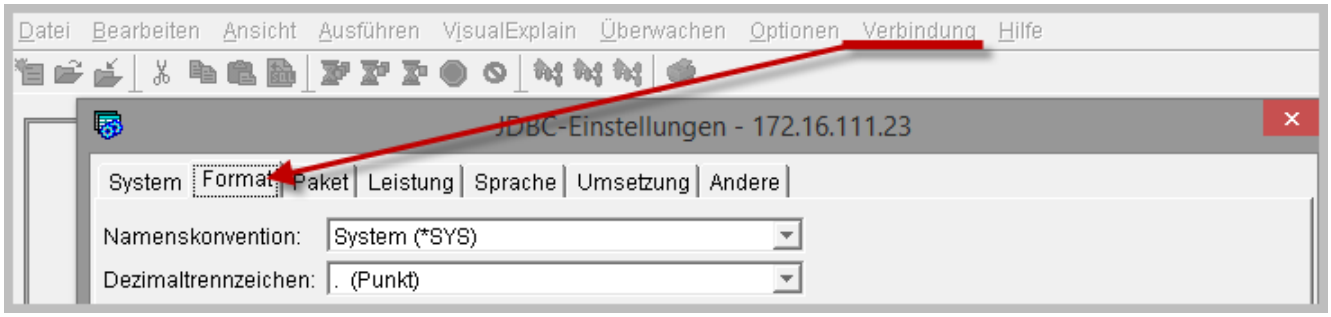
Treffen Sie keine weiteren Vorkehrungen, werden in diesem Fall die Sonderregister CURRENT SCHEMA und CURRENT PATH immer auf den Sonderwert *LIBL gesetzt. Es greift somit die Bibliothekssuchliste des aktuellen Jobs. Die JDBC-Einstellungen werden jetzt nicht berücksichtigt.

Sie können aber auch in diesen Fällen den Befehl SET SCHEMA oder SET PATH verwenden, um die Sonderregister zu manipulieren.

Am einfachsten ist es, wenn Sie immer dafür sorgen, dass das jeweilige Interface stets die Systemnamenskonventionen verwendet:



Globale Namenskonventionen System i Navigator



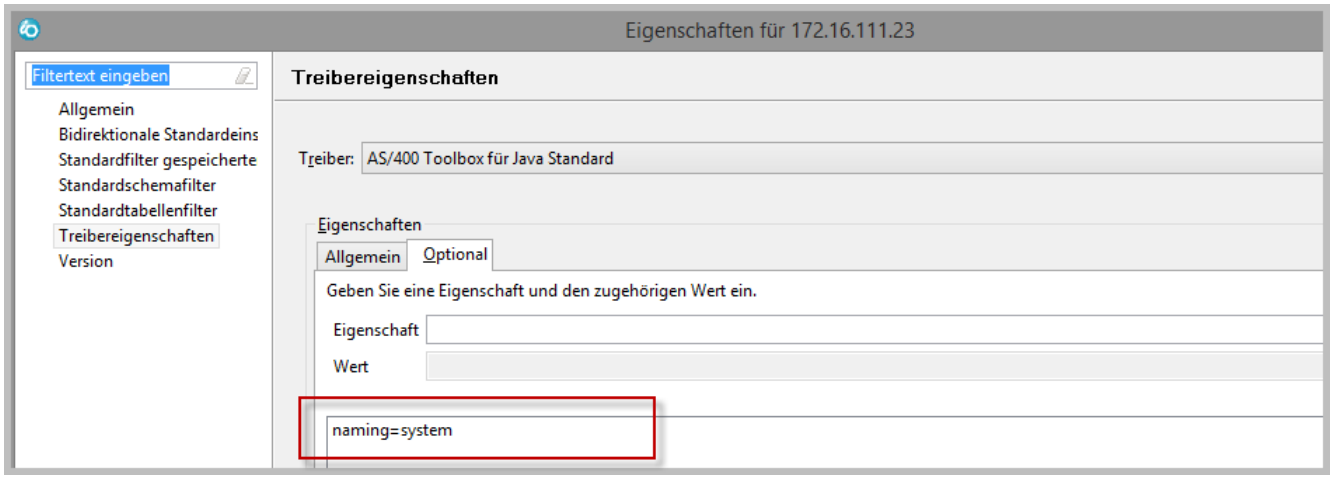
Systemnamenskonventionen in der Prozedurumgebung des Navigators wählen

Auch wenn Sie mit der 5250-Emulation und der STRSQL-Umgebung arbeiten, können Sie die Namenskonventionen frei wählen:

```
                    Sitzungsattribute ändern

Auswahl eingeben und Eingabetaste drücken.

  Anweisungsverarbeitung . . . .   *RUN           *RUN, *VLD, *SYN
  SELECT-Ausgabe . . . . . . . .   1             1=Datensichtgerät, 2=Drucker
                                                 3=Datei
  COMMIT-Steuerung . . . . . . .   *CHG          *NONE, *CHG, *CS, *ALL, *RR
                                                 *NC, *UR, *RS
  Datumsformat . . . . . . . . .   *DMY          *JOB, *USA, *ISO, *EUR, *JIS
                                                 *MDY, *DMY, *YMD, *JUL
  Datumstrennzeichen . . . . . .   '.'           *JOB, '/', '.', ',', '-'
                                                 ' ', *BLANK
  Zeitformat . . . . . . . . . .   *HMS          *HMS, *USA, *ISO
                                                 *EUR, *JIS
  Zeittrennzeichen . . . . . . .   ':'           *JOB, ':', '.', ','
                                                 ' ', *BLANK
  Datenaktualisierung  . . . . .   *ALWAYS       *ALWAYS, *FORWARD
  Datenkopie zulässig  . . . . .   *YES          *YES, *OPTIMIZE, *NO
  Namenskonvention . . . . . . .   *SYS          *SYS, *SQL

                                                                  Weitere ...
F3=Verlassen    F4=Bedienerführung    F5=Aktualisieren    F12=Abbrechen
```

F13 = Sitzungsattribute in der STRSQL-Umgebung ändern

Und auch die JAVA-Treiber lassen sich konfigurieren:



JDBC-Treibereigenschaften der Datenbankverbindung

Manchmal muss man etwas suchen, aber bisher habe ich noch keine Datenbankverbindung zum IBM i Server gesehen, bei der man zwingend mit der SQL-Konvention arbeiten muss. Suchen Sie lieber ein wenig nach den richtigen Einstellungen, bevor sie unnötge Adressierungsprobleme bekommen und später ständig die Bibbliotheksnamen in Ihren Skripten verwenden müssen.

7.7.3
Seite 4

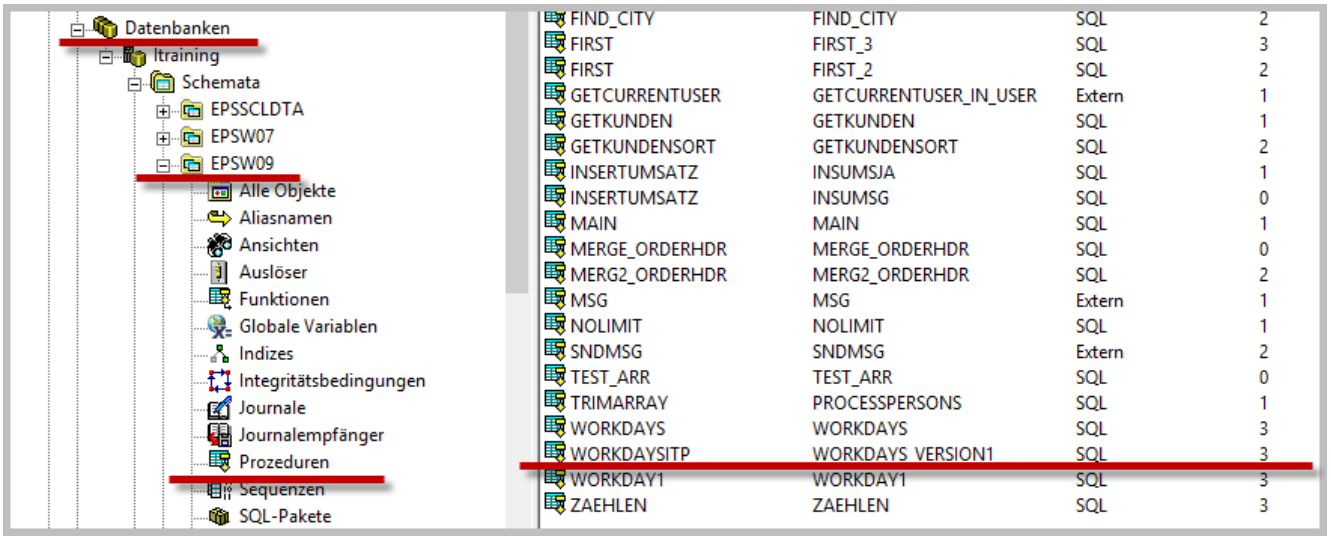
Aber nicht nur auf die Adressierung Ihrer Objekte hat die Namenskonvention Einfluss. Auch das Berechtigungssystem ist betroffen. Wenn Sie sich für die SQL-Namenskonvention entscheiden, werden für *PUBLIC keine Rechte (*EXCLUDE) erteilt. Bei der Namenskonvention *SYS entscheidet der Parameter CRTAUT des Schemas, welche *PUBLIC-Rechte für die Prozedur gelten. Und auch die Eigentumsrechte werden unterschiedlich gesetzt. Arbeiten Sie mit der Namenskonvention *SQL, wird der erstellende Benutzer auch Eigentümer des Objekts. Bei der Namenskonvention *SYS greifen die Systemeinstellungen: Gehört der Benutzer zu einer Gruppe und wurde im Benutzerprofil vereinbart, dass das Eigentumsrecht stets auf die Gruppe übertragen wird, wird die Gruppe beispielsweise Eigentümer der Prozedur, ansonsten der Programmierer.

Ich werde im weiteren Verlauf dieses und der folgenden Kapitel nicht weiter auf die Wirkung der Namenskonventionen eingehen, sondern gehe einfach davon aus, dass Sie – so wie ich es in allen Beispielen tue – mit der Systemnamenskonvention arbeiten.

Nun aber zurück zur Prozedurerstellung:

```
CL: CHGCURLIB EPSW09;

CREATE OR REPLACE PROCEDURE WORKDAYSITP
(IN VONDATUM DATE ,
IN BISDATUM DATE ,
OUT ERGEBNIS INTEGER )
LANGUAGE SQL
READS SQL DATA
SPECIFIC WORKDAYS_VERSION1
PROGRAM TYPE SUB

INHERIT SPECIAL REGISTERS
CALLED ON NULL INPUT
DETERMINISTIC

SET OPTION
       DATFMT = *EUR,
       DBGVIEW = *SOURCE,
       OPTLOB = *YES,
       SQLCURRULE = *DB2,
       TIMFMT = *EUR
BEGIN
SET ERGEBNIS = (SELECT DAYS(BISDATUM) – DAYS(VONDATUM) FROM
SYSIBM/SYSDUMMY1);
END;
```

Führen Sie das oben gezeigte SQL-Skript im System i Navigator aus, sollte die Prozedur im Schema EPSW09 ohne Probleme erstellt werden. Anschließend können Sie das Ergebnis ganz einfach überprüfen, indem Sie im System i Navigator – wie unten gezeigt – die Ansicht „Datenbanken ➜ Prozeduren“ öffnen:



Die erstellte Prozedur finden wir im Navigator

Sie können die vorhandenen Prozeduren aber auch in unterschiedlichen Datenbankkatalogen finden. Sie können die Kataloge QSYS2/SYSPROCS, QSYS2/SYSROUTINEDEP oder QSYS2/SYSROUTINES nutzen:

```
SELECT * FROM QSYS2/SYSROUTINES
WHERE routine_name ='WORKDAYSITP' ;
```

SPECIFIC_SCHEMA	SPECIFIC_NAME	ROUTINE_SCHEMA	ROUTINE_NAME	ROUTINE_TYPE
EPSW09	WORKDAYS_VERSION1	EPSW09	WORKDAYSITP	PROCEDURE
EPSW09	WORKDAYS_VERSION2	EPSW09	WORKDAYSITP	PROCEDURE

Die Prozeduren finden Sie in verschiedenen Datenbankkatalogen

Neben den oben gezeigten Informationen können Sie das Erstellungsdatum, die Prozedureigenschaften, aber auch den Quellcode lesen:

```
SELECT routine_definition FROM QSYS2.SYSROUTINES
WHERE routine_name ='WORKDAYSITP' ;
```

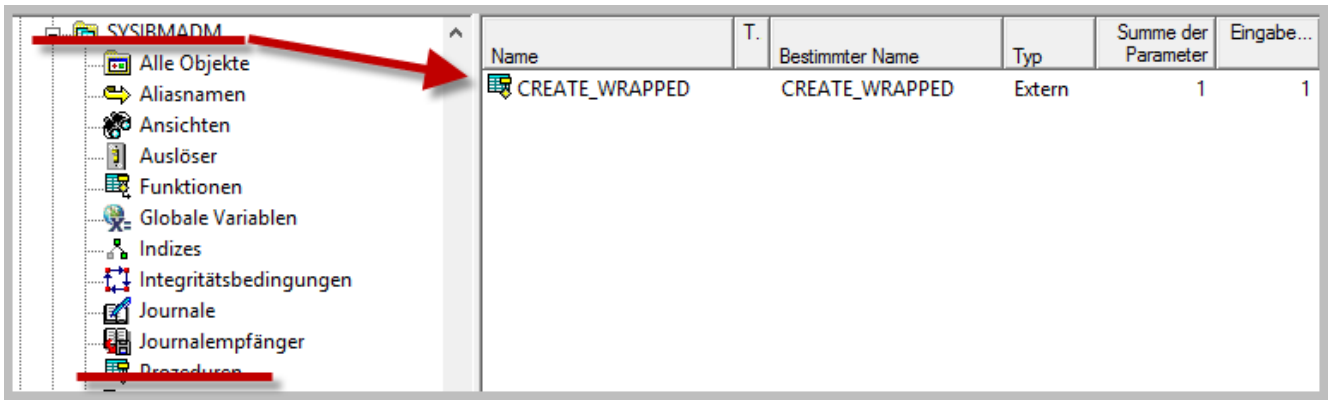
ROUTINE_DEFINITION
BEGINSET ERGEBNIS = (SELECT DAYS (BISDATUM) - DAYS (VONDATUM) FROM SYSIBM / SYSDUMMY1) ;END
BEGINSET ERGEBNIS = (SELECT DAYS (CURRENT_DATE) - DAYS (VONDATUM) FROM SYSIBM / SYSDUMMY1) ;END

Auch der Quellcode steht im DB-Katalog

7.7.3
Seite 6

7.7.4 Prozeduren im Datenbankkatalog verschlüsseln

Falls Sie unterbinden wollen, dass der Sourcecode Ihrer Prozeduren im Klartext über die Datenbankkataloge gelesen werden kann, können Sie die Prozeduren jetzt auch im Datenbankkatalog verschlüsseln. Dazu benötigen Sie die IBM-Prozedur CREATE_WRAPPED, die Sie im Schema SYSIBMADM finden:



Die IBM-Prozedur CREATE_WRAPPED

Die Prozedur erwartet einen Parameter vom Typ CLOB. Dieser Parameter erhält das CREATE PROCEDURE-Statement. Die Klausel OR REPLACE ist in diesem Umfeld nicht erlaubt.

7.7.4
Seite 2

Da die Prozedur CREATE_WRAPPED einen CLOB-Parameter erwartet, müssen Sie eine Transaktion starten, bevor das SQL-Statement ausgeführt wird, und dürfen dann natürlich nicht vergessen, die Ausführung festzuschreiben. Andernfalls wird die Prozedur wieder gelöscht, wenn Sie den Job beenden:

```
SET TRANSACTION ISOLATION LEVEL CS;

DROP SPECIFIC PROCEDURE WORKDAYS_VERSION2;

CALL SYSIBMADM/CREATE_WRAPPED
('CREATE PROCEDURE WORKDAYSITP
(IN VONDATUM DATE ,
IN BISDATUM DATE,
OUT ERGEBNIS INTEGER )
LANGUAGE SQL
READS SQL DATA
SPECIFIC WORKDAYS_VERSION2
PROGRAM TYPE SUB

INHERIT SPECIAL REGISTERS
CALLED ON NULL INPUT
DETERMINISTIC

SET OPTION
       DATFMT = *EUR,
       DBGVIEW = *SOURCE,
       OPTLOB = *YES,
       SQLCURRULE = *DB2,
       TIMFMT = *EUR
BEGIN
SET ERGEBNIS = (SELECT DAYS(BISDATUM) - DAYS(VONDATUM) FROM
SYSIBM/SYSDUMMY1);
END');

COMMIT;
```

Die Prozedur CREATE_WRAPPED verschlüsselt den Quelltext und implementiert die Prozedur in der Datenbank. Schauen Sie sich anschließend den Eintrag im Datenbankkatalog an, so werden Sie nur Folgendes sehen:

ROUTINE_DEFINITION
BEGINSET ERGEBNIS = (SELECT DAYS (BIS...
BEGINSET ERGEBNIS = (SELECT DAYS (CUR...
WRAPPED QSQ07010 aacxW8plW8FrG8pzG8V...

Die Wrapped-Prozedur im Datenbankkatalog

Der Eintrag beginnt mit der Zeichenfolge WRAPPED. Es folgt die Zeichenfolge QSQ07010, die Rückschlüsse auf den Server zulässt. QSQ zeigt, dass es sich um eine IBM i-Datenbank handelt. Dahinter finden Sie den Releasestand des Servers: i7.1 Modifikation 0.

Nutzen können Sie die Prozedur wie gewohnt. Nur das SQL-Statement kann natürlich nicht aus dem Objekt generiert werden bzw. können Sie den Quellcode zwar generieren – nur lesen können Sie ihn nicht:

```
CREATE PROCEDURE EPSW09/WORKDAYSITP (
                IN VONDATUM DATE ,
                OUT ERGEBNIS INTEGER )
WRAPPED QSQ07010 abhxW8plW8FrG8pzG8VnG8Vj69pf69Fbl9Ft5_Fh1qpdW8pdW8pd08FJvXebaqebarWsgrtXugaPTGBl5
gs42w53x6YvJXqYRWL3jTcPkHsmMR8:5JLSPZTC94fJDzRvdRxiq84KDuQAPdT672I7WLZrhQP0WZC:_bElLRLjlrnreK_mMPg
hRdSnlv7dbCeQcvgfjyHTQ2bJp_562wRlEldBnpz:dxXV7n3S3zkxTwHnBgyInOlYXCCh44uxNrLlK5Wjxfg0ytnkZesvrOLdPSmCgh
UyD20cIK2FCw:SqU4Yi5mPLsujkM4JpePQdE28RykxU_dr66tRVMFZOTePAvk1De26KAMr982hNQGKiZQdXj8hZZ5MTnbA_ND
PpI8gAMxF7u8ObM1xXTNX1DRnJyZZnt0VH0ZmBnA5O9aEcIF4FwhspgYdB74xUbfoYcih43uIekTi8qNNvkZxV__9EdqTEbCVx
IIf58lkxCLDv6NnUxYaVnFpqxEbIcYLetQ0GiCN8Osdd9c8oA47av4Cb2cJhSaoIsM3D0xhddblinSKz7KkKxb:j93osogDoo5z2ZnRwa
vdajbfrJBfKS_iljUybX4LCOMUKIMv:nQoZ:ZZV:NJS;
```

SQL-Generierung WRAPPED-Prozeduren

7.7.4
Seite 4

7.7.5 Der Prozeduraufruf

Nun wollen wir unsere Prozeduren aber auch aufrufen. Alle Prozeduren werden mit der Anweisung CALL aufgerufen:

```
CALL WORKDAYSITP ('03.02.2014', '03.05.2014', ?);
```

Der Aufruf kann unqualifiziert erfolgen, da wir die Bibliothek EPSW09 in der Suchliste haben. Achten Sie auf das Fragezeichen: Alle OUT-Parameter einer Prozedur müssen durch ein Fragezeichen maskiert werden. Beim Aufruf identifiziert die Datenbank die Prozedur nicht allein aufgrund ihres Namens, sondern auch anhand der Anzahl Parameter. Und eine Anmerkung zur Übergabe der Datumswerte: Da die Prozedur europäisches Datumsformat erwartet, müssen Sie die Konstanten und Variablen in diesem Format übergeben, andernfalls müssten die Konstanten explizit konvertiert werden:

```
CALL WORKDAYSITP (DATE('2014-03-02'), DATE('2014-04-01'), ?);
```

Seit einiger Zeit ist es auch möglich, optionale Parameter zu verwenden. Alle optionalen Prozedurparameter benötigen allerdings einen DEFAULT-Wert. Diese Tatsache kann spannend sein, wenn Sie an die vielen Prozeduren denken, die eventuell schon seit Jahren vorhanden sind und in den unterschiedlichsten Umfeldern genutzt werden. Bislang konnten diese Prozeduren nur schwer erweitert werden, und häufig setzte man Überlagerungstechniken ein, um nicht die vielen Prozeduraufrufe korrigieren zu müssen. Jetzt geht es sehr viel einfacher. Stellen Sie sich vor, dass unsere Prozedur, wenn nur ein Datum angegeben wird, die Differenz zum heutigen Tag berechnen soll.

Um diese Aufgabe zu integrieren, müssen wir als Erstes die Parameterstruktur ein wenig ändern:

```
CREATE OR REPLACE PROCEDURE WORKDAYSITP
(IN VONDATUM DATE ,
IN BISDATUM DATE DEFAULT CURRENT DATE ,
OUT ERGEBNIS INTEGER )
```

Das „BisDatum“ wird mit dem Tagesdatum aktualisiert. Anschließend können wir die Prozedur auf folgende Weise aufrufen:

```
-- AUFRUF WIE BISHER
CALL WORKDAYSITP (DATE('2014-03-02'), DATE('2014-04-01'), ?);

-- AUFRUF MIT EINEM DATUMSWERT
CALL WORKDAYSITP (DATE('2014-03-02'), DEFAULT, ?);
```

7.7.5

Seite 2

7.7.6 Overloading von Prozeduren

Wir könnten die Prozedur aber auch überladen, indem wir einfach eine zweite Prozedur mit identischem Prozedurnamen, aber unterschiedlichen spezifischen Namen schreiben:

```
CREATE OR REPLACE PROCEDURE WORKDAYSITP
(IN VONDATUM DATE ,
OUT ERGEBNIS INTEGER )
LANGUAGE SQL
READS SQL DATA
SPECIFIC WORKDAYS_VERSION2
PROGRAM TYPE SUB

INHERIT SPECIAL REGISTERS
CALLED ON NULL INPUT
DETERMINISTIC

SET OPTION
       DATFMT = *EUR,
       DBGVIEW = *SOURCE,
       OPTLOB = *YES,
       SQLCURRULE = *DB2,
       TIMFMT = *EUR
BEGIN
SET ERGEBNIS = (SELECT DAYS(current_date) - DAYS(VONDATUM)
FROM SYSIBM/SYSDUMMY1);
END;
```

Die neue Prozedur erhält den gleichen Prozedurnamen, hat aber lediglich zwei Parameter: das „VonDatum“ und das „Ergebnis“. Der Prozedurname und die Parameteranzahl werden als Prozedursignatur verstanden. So können Sie dieselben Prozedurnamen auch bei variierender Parameteranzahl verwenden.

Aber auch der spezifische Name variiert, damit sich die Prozeduren nicht nur anhand der Prozedursignatur unterscheiden. Sobald die zweite Prozedur erstellt ist, können Sie die Prozeduren jetzt folgendermaßen aufrufen:

```
CALL WORKDAYSITP (date('2014-03-02'), ?);
CALL WORKDAYSITP (date('2014-03-02'), date('2014-03-10'), ?);
```

Ergänzung 30/2014

7.7.6
Seite 2

Falls Sie die Prozedur WORKDAYSITP jetzt aber löschen müssen, ist der Prozedurname nicht mehr eindeutig. Sie müssten zur eindeutigen Identifikation der Prozedur die vollständige Signatur, also den Namen und die Parameteranzahl, angeben oder den spezifischen Prozedurnamen verwenden:

```
DROP PROCEDURE WORKDAYSITP (DATE), INTEGER);
-- oder
DROP SPECIFIC PROCEDURE WORKDAYS_VERSION2;
```

7.7.7 SQL Control Language

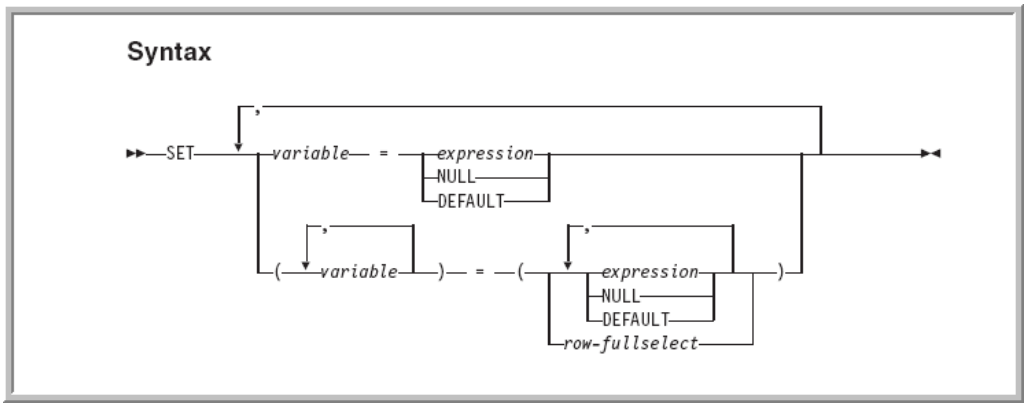
Nicht nur der Befehl CREATE PROCEDURE ist notwendig, um leistungsfähige Prozeduren zu schreiben, sondern Sie müssen natürlich auch die SQL-Prozedur-Sprache selbst verstehen, damit Sie diese Technik optimal einsetzen können. Die einzelnen SQL-Elemente, die wir für unsere Prozeduren verwenden, erinnern dabei an andere Programmiersprachen.

Variablen
Natürlich können Sie Variablen in der Prozedur deklarieren. Sie haben im ersten Beispiel bereits gesehen, dass die Parameter selbst als Variablen verwendet werden können. Darüber hinaus können Sie mit der Anweisung DECLARE weitere Variablen deklarieren:

```
DECLARE SQLCODE      INTEGER;
DECLARE ABS_VK       DECIMAL(11,2);
DECLARE ERRMSG       VARCHAR(250)    DEFAULT 'ALLES OKAY! ';
DECLARE DATUM_EIN    DATE;
DECLARE PICTURE      BLOB(10M);
DECLARE ZEITSTAMP    TIMESTAMP;
```

Anhand der Beispiele erkennen Sie, dass der Anweisung DECLARE der Variablenname folgt. Der Variablenname ist nicht case-sensitive. Anschließend geben Sie Datentyp und Länge und ggf. einen Standardwert an.

Übertragungen
Die Anweisung SET haben wir bereits im ersten Beispiel kenngelernt. Mit der Anweisung SET können Sie Wertzuweisungen vornehmen:



Syntaxdiagramm Anweisung SET

7.7.7
Seite 2

Schleifen

Auch Schleifen können wir in einer SQL-Prozedur codieren. Bei der Anweisung WHILE handelt es sich um eine Schleife mit Laufbedingung. Die Anweisungen im Schleifenbody werden so lange verarbeitet, solange die Schleifenbedingung wahr ist.

```
WHILE laufbedingung
DO

AUSFÜHRBARE ANWEISUNG;
AUSFÜHRBARE ANWEISUNG;

END WHILE ;
```

Bei der Anweisung REPEAT handelt es sich hingegen um eine Schleife mit Abbruchbedingung. Die Anweisungen im Schleifenbody werden verarbeitet bis die Schleifenbedingung wahr ist.

```
REPEAT

      AUSFÜHRBARE ANWEISUNG;
      AUSFÜHRBARE ANWEISUNG;

UNTIL abbruchbedingung
END REPEAT;
```

Bedingungen

Und natürlich können wir auch Verzweigungen mit der Anweisung IF-THEN-ELSE codieren. Dabei nutzen wir die IF-Anweisung auf dieselbe Art und Weise, wie wir es von anderen Programmier-sprachen kennen:

```
IF bedingung

THEN
      AUSFÜHRBARE ANWEISUNG;
      AUSFÜHRBARE ANWEISUNG;
ELSE
      AUSFÜHRBARE ANWEISUNG;
      AUSFÜHRBARE ANWEISUNG;

END IF;
```

Allerdings kennen wir in der SQL-Programmierung auch die Anweisung ELSEIF – für RPG-Programmierer sicherlich ungewohnt. In anderen Programmiersprachen gehört die Anweisung ELSEIF hingegen zum Standard:

```
IF bedingung

THEN
       AUSFÜHRBARE ANWEISUNG;
       AUSFÜHRBARE ANWEISUNG;


ELSEIF bedingung
THEN   AUSFÜHRBARE ANWEISUNG;
       AUSFÜHRBARE ANWEISUNG;
       ELSE
       AUSFÜHRBARE ANWEISUNG;
       AUSFÜHRBARE ANWEISUNG;
--     KEIN ENDIF FÜR ELSEIF

END IF;
```

Beachten müssen Sie hierbei lediglich, dass die ELSEIF-Anweisung nicht mit einem END IF abgeschlossen wird. Und natürlich können Sie auch mit der Anweisung CASE Fallunterscheidungen codieren:

```
CASE felddname

       WHEN BEDINGUNG1 THEN AUSFÜHRBARE ANWEISUNG;
                            AUSFÜHRBARE ANWEISUNG;
       WHEN BEDINGUNG2 THEN AUSFÜHRBARE ANWEISUNG;
                            AUSFÜHRBARE ANWEISUNG;
ELSE
       AUSFÜHRBARE ANWEISUNG;

END CASE;
```

7.7.7
Seite 4

Wie im interaktiven Umfeld unterscheiden wir auch in der Programmierung zwei Formate:

```
CASE
       WHEN BEDINGUNG1 THEN ANWEISUNGEN;
       WHEN BEDINGUNG2 THEN ANWEISUNGEN;
ELSE
       ANWEISUNGEN;

END CASE;
```

Und nun wollen wir diese steuernden Anweisungen nutzen und unsere Prozedur etwas verändern. Es soll nicht mehr nur die Anzahl Tage zwischen den beiden Daten ermittelt werden, sondern wir wollen auch die Wochentage Montag bis Freitag zählen.

Was ist zu tun? Wir müssen jeden Datumswert im gewählten Zeitraum auf den Wochentag hin überprüfen. Dafür nutzen wir die Funktion DAYOFWEEK_ISO. Sie liefert den Wert 6 für einen Sonnabend und den Wert 7 für einen Sonntag. Das Ergebnis speichern wir in einer Programmvariablen:

```
SET TAG = SELECT DAYOFWEEK_ISO ( DAT_VON ) FROM SYSIBM /
SYSDUMMY1
```

Anschließend prüfen wir den Wert und zählen die Werktage Montag bis Freitag:

```
IF TAG <> 6 AND TAG <> 7    THEN

        SET ERGEBNIS = ERGEBNIS + 1 ;
END IF ;
```

Und da wir diese Schritte für jedes Datum separat durchführen müssen, brauchen wir auch noch eine Schleife:

```
DECLARE TAG SMALLINT;

SET ERGEBNIS = 0;

WHILE VONDATUM <= BISDATUM
DO
       SET TAG = ( SELECT DAYOFWEEK_ISO ( VONDATUM ) FROM
SYSIBM / SYSDUMMY1 ) ;
       IF TAG <> 6 AND TAG <> 7   THEN
          SET ERGEBNIS = ERGEBNIS + 1 ;
       END IF ;

       SET VONDATUM = VONDATUM + 1 DAY ;

END WHILE ;
```

Etwas aufpassen müssen Sie mit dem Parameter ERGEBNIS. Er wird mit dem Wert NULL übergeben und muss daher initialisiert werden, damit Sie ihn in der Rechenoperation verwenden können.

Und falls Ihnen diese Lösung zu umständlich erscheint, geht es auch noch etwas kompakter:

```
BEGIN
       SET ERGEBNIS = 0;
       WHILE VONDATUM < BISDATUM
       DO

       IF 1 =   ( SELECT 1 FROM SYSIBM / SYSDUMMY1
          WHERE DAYOFWEEK ( VONDATUM ) BETWEEN 1 AND 5 )
       THEN  SET ERGEBNIS = ERGEBNIS + 1 ;
       END IF ;
       SET VONDATUM = VONDATUM + 1 DAY;
       END WHILE;
END ;
```

7.7.7
Seite 6

7.8 Funktionen

Wenn Sie bereits mit SQL gearbeitet haben, dann wissen Sie, dass insbesondere der Einsatz der vielen skalaren Funktionen hierbei das Arbeiten vereinfacht. So bietet SQL zum Beispiel viele integrierte Funktionen, die etwa die Konvertierung von Datentypen und die Modifikation von Spaltenwerten ermöglichen. Trotzdem lösen die vorhandenen SQL-Funktionen nicht immer alle Probleme. Deshalb besteht auch die Möglichkeit, eigene Funktionen mittels SQL zu erstellen, um beispielsweise unsere IBM i-spezifischen „Datumsfelder“ in den Datenbankdateien zu konvertieren oder Kennzahlen zu generieren etc. pp. Die Einsatzmöglichkeiten sind vielfältig.

Schauen wir uns als Erstes eine skalare Funktion an, die wir mit Hilfe des System i Navigator erstellen.

Ergänzung 41/2017

7.8 Seite 2

7.8.1 Skalare SQL-Funktionen

Bevor wir anfangen, stellen wir uns zunächst eine Aufgabe: Unsere Funktion soll uns den verfügbaren Bestand eines Artikels berechnen. Dabei soll auch der aktuelle Auftragsbestand berücksichtigt werden. Wir benötigen zur Lösung dieses Problems zwei Tabellen.

Die Tabelle „Lagerbestand“ zeigt uns die aktuell verfügbaren Lagermengen:

ARTIKELNUMMER	LAGERORT	BESTAND
00001	1	80104
00001	2	80104
00001	78	80202
00001	13	68203
00001	10	68203
00002	3	90000
00002	7	80107

Die Tabelle Lagerbestand

Zusätzlich benötigen wir noch die Auftragspositionen. Alle Auftragspositionen mit einem Statuswert kleiner als zwei sind bislang noch nicht im Lager verbucht:

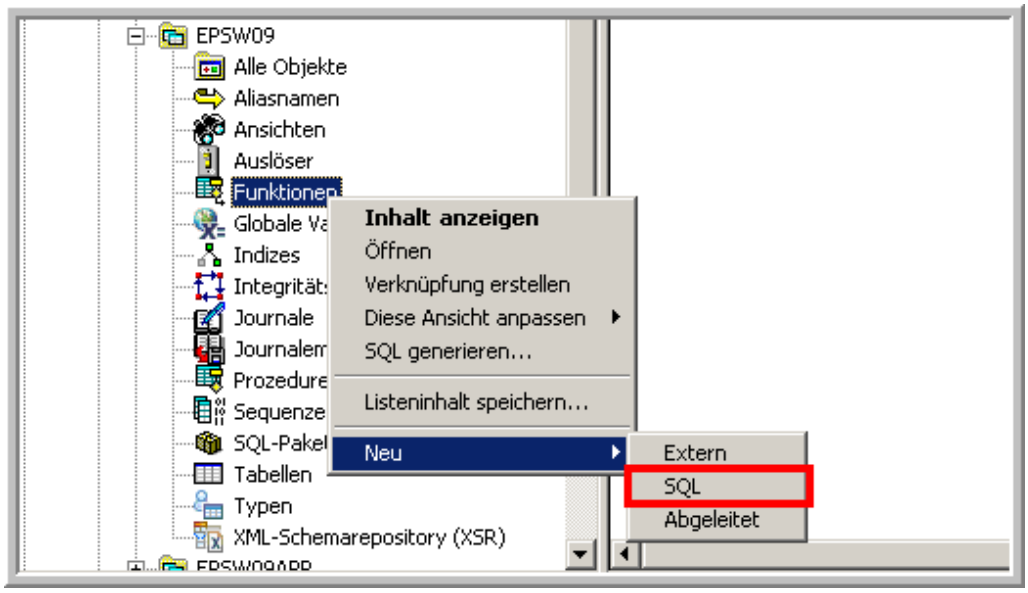
ARTIKE...	POSITIONSMENGE	STATUS
00001	217	1
00001	217	1
00001	159	1
00001	159	1
00001	159	1
00001	378	1
00001	78	1

Die Tabelle Auftragspositionen

7.8.1
Seite 2

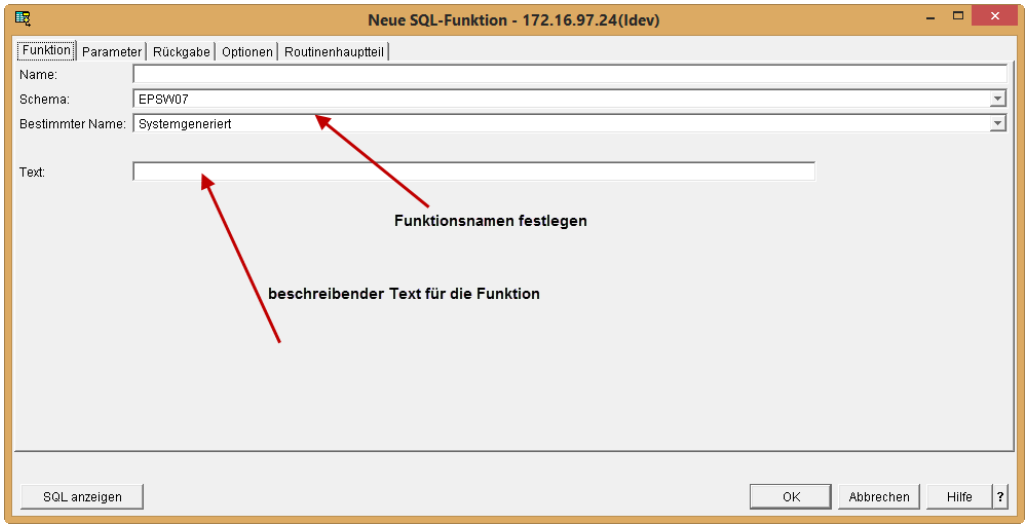
7.8.1.1 Skalare Funktionen mit dem System i Navigator definieren

Um eine skalare SQL-Funktion zu definieren, öffnen wir im System i Navigator im entsprechenden Schema mit einem Rechtsklick „Funktionen ➜ Neu ➜ SQL“.



System i Navigator: neue SQL-Funktion

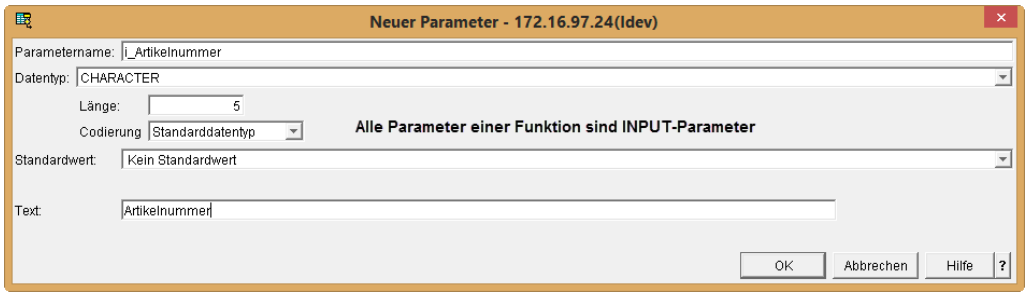
Daraufhin sollten Sie folgende Anzeige erhalten:



Funktionsnamen festlegen

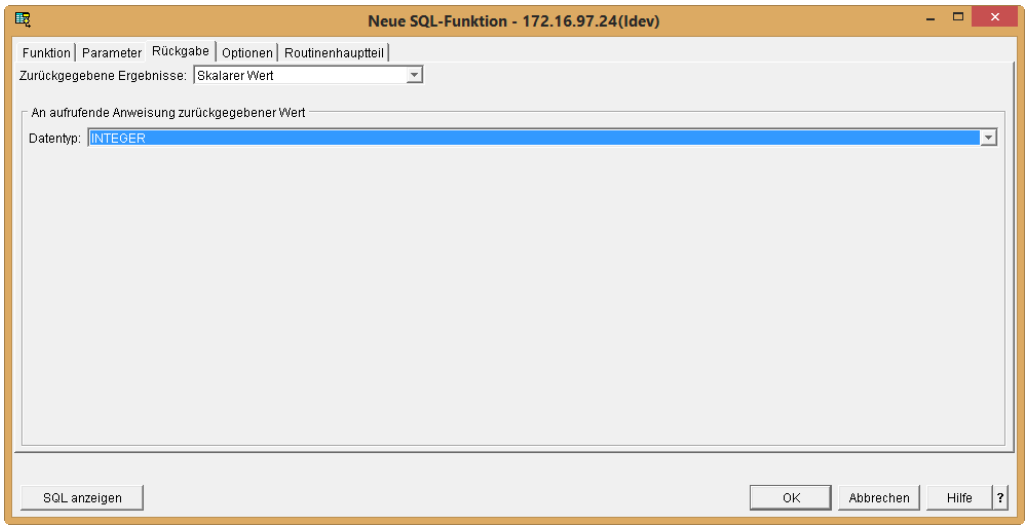
Als Erstes geben wir der Funktion einen Namen. Wie bei den Prozeduren können wir einen langen Prozedurnamen und einen kurzen Namen – maximal 10 Zeichen – festlegen. Letzterer wird wieder lediglich systemseitig verwendet. Zusätzlich kann ein beschreibender Text verwendet werden. Was sicherlich keine schlechte Idee ist!

Anschließend legen wir die Parameter fest. Anders als bei den Prozeduren gibt es hier nur Input-Parameter. Wir übergeben der Funktion in unserem Beispiel die Artikelnummer:



Die Parameterschnittstelle

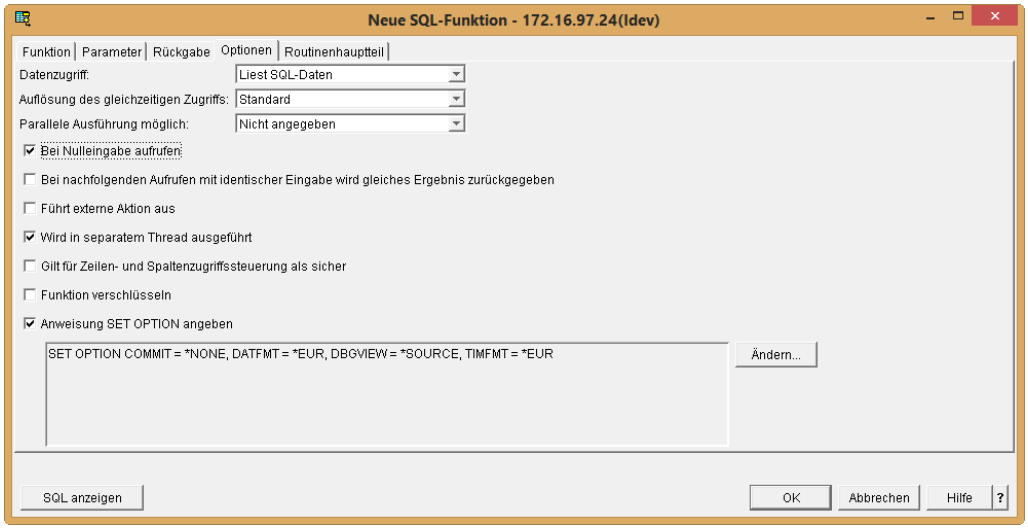
Von einer Funktion erhalten wir grundsätzlich einen Rückgabewert. In unserem Beispiel erhalten wir den verfügbaren Bestand, der als Integer-Wert zurückgegeben werden soll:



Der Rückgabewert

7.8.1
Seite 4

Im nächsten Schritt bestimmen wir die spezifischen Eigenschaften der Funktion:



Funktionseigenschaften

Schauen wir uns die festgelegten Eigenschaften kurz an:

Der Datenzugriff bestimmt, welche SQL-Anweisungen die Funktion ausführt. Wir müssen für unser Beispiel lediglich lesend auf die Tabellen zugreifen. Nachfolgend noch einmal die Übersichtstabelle:

Attribute	Beschreibung
enthält SQL	Folgende Statements dürfen verwendet werden: • CALL für Prozeduren, die ihrerseits NO SQL- oder CONTAINS SQL-Eigenschaften haben. • SET RESULT SET • SET and VALUES INTO (mit Variablen oder Konstanten) • COMMIT, ROLLBACK, SET TRANSACTION • CONNECT, DISCONNECT, SET CONNECTION
liest SQL-Daten	Die Daten werden mit SQL gelesen (SELECT). Es können alle Statements verwendet werden außer: • COMMIT, ROLLBACK, SET TRANSACTION • CONNECT, DISCONNECT, SET CONNECTION • DELETE, INSERT, UPDATE • ALTER TABLE, COMMENT ON • CREATE, DROP, GRANT, LABEL ON, RENAME, REVOKE
ändert SQL	Die Daten werden gelesen und/oder mit SQL modifiziert.

Auflösung bei gleichzeitigem Zugriff
Die Eigenschaft legt fest, wie das DBMS mit noch nicht festgeschriebenen Daten verfahren soll.

- **Ergebnis abwarten**
  Das DBMS wartet auf die Festschreibung der Datensätze.
- **Momentan festgeschriebene Version verwenden**
  Bei dieser Klausel werden nur die bislang festgeschriebenen Zeilen verwendet.
- **Standard**
  Die Eigenschaft wird zunächst nicht explizit festgelegt. Vielmehr wird die Klausel verwendet, die beim Aufruf der Funktion gültig ist.

Parallele Ausführung möglich
Hier wird spezifiziert, dass die Funktion ggf. parallel ausgeführt werden könnte. Ob dies jedoch wirklich der Fall ist, entscheidet das DBMS. Wird für die Funktion die Eigenschaft NOT DETERMINISTIC, EXTERNAL ACTION, MODIFIES SQL DATA, SCRATCHPAD oder FINAL CALL vereinbart, ist DISALLOW PARALLEL zwingend vorgeschrieben. Wir überlassen daher die Entscheidung dem DBMS.

Bei NULL-Parameter aufrufen
Die Funktion wird nicht aufgerufen, wenn NULL als Inputwert verwendet wird. Der Return-Wert ist in diesen Fällen NULL. Wird die Ausführung der Funktion hingegen mit NULL-Aufrufen erlaubt, muss die Funktion selbst in der Lage sein, diese NULL-Werte zu verarbeiten.

Bei nachfolgenden Aufrufen mit identischer Eingabe wird gleiches Ergebnis zurückgegeben
Diese Eigenschaft bestimmt, ob bei aufeinanderfolgenden Aufrufen mit denselben Parametern auch immer dieselben Ergebnisse zurückgegeben werden. In unserem Beispiel könnte sich der verfügbare Bestand zwischen zwei identischen Aufrufen ändern. Unsere Funktion ist NICHT deterministisch.

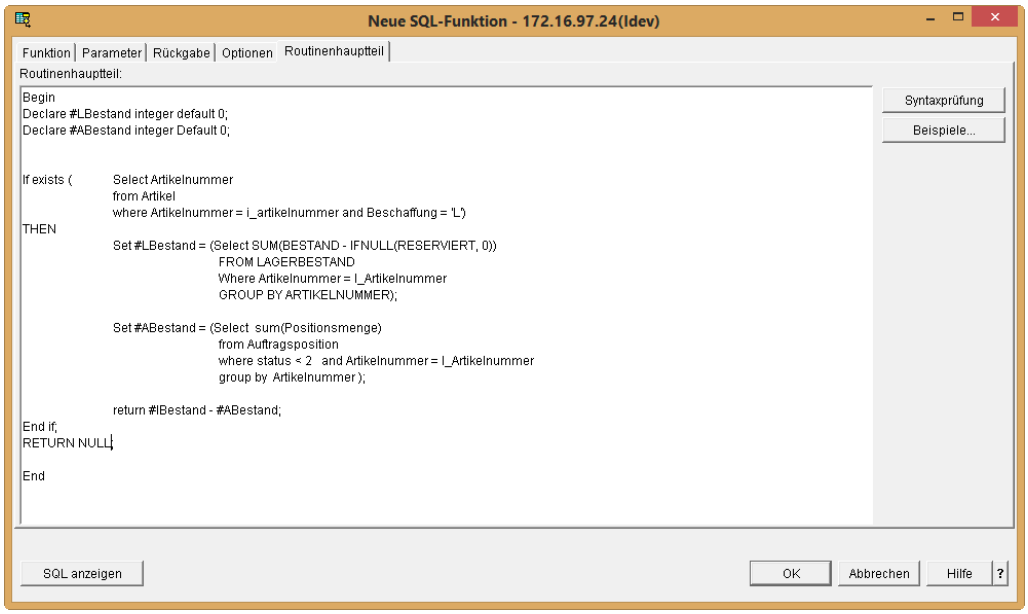
Führt externe Aktionen aus
Externe Aktionen sind Zugriffe auf Nicht-DB-Objekte. Das Senden einer Nachricht oder das Schreiben eines Satzes in eine Streamfile sind beispielsweise externe Aktionen.

Wird in separatem Thread ausgeführt (Standard)
Wird die Funktion in einem separaten Thread ausgeführt, können Cursor beispielsweise nicht zwischen mehreren individuellen Aufrufen geteilt werden. Der Cursor eines Threads ist stets unabhängig vom Cursor eines anderen Threads. So reduzieren sich Namenskonflikte deutlich. Wird aber die Funktion in einem gemeinsamen Thread ausgeführt, läuft die Funktion im selben Thread wie das aufgerufene SQL-Statement. SQL-Cursor können über verschiedene individuelle Aufrufe hinweg geöffnet bleiben. Da die Cursor geöffnet bleiben, wird auch die Cursorposition zwischen verschiedenen Aufrufen beibehalten. Da nun aber die Funktion und die SQL-Statements in einem Thread laufen, können schnell Namenskonflikte auftreten. Dafür ist die Performance in der Regel ein wenig besser.

Gilt für Zeilen- und Spaltenzugriffssteuerung als sicher
Standardmäßig gelten Funktionen als nicht sicher für Row and Column Access (RCAC). Dadurch können als Input-Argumente keine Tabellenspalten verwendet werden, für die Spaltenmasken definiert wurden, wenn RCAC für die Tabelle aktiviert ist. Soll die Funktion dennoch verwendet werden, muss die Funktion explizit als sicher gekennzeichnet sein.

Funktion verschlüsseln
Funktionen werden im Datenbankkatalog SYSFUNCS eingetragen. Im Katalog finden Sie neben den Funktionseigenschaften auch den Quellcode. Für Softwareanbieter ist es ärgerlich, wenn ihre Kunden auf diesem Weg auf den Quellcode zugreifen können. Hier bietet sich vielleicht die Verschlüsselung des Quellcodes an.

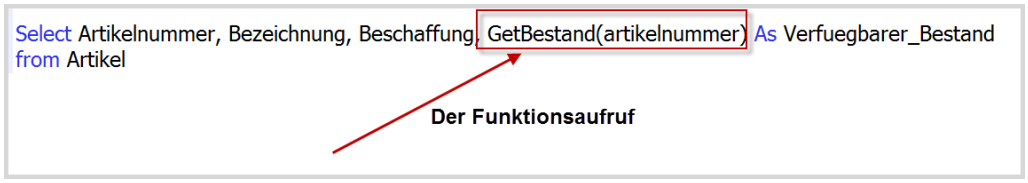
Als Letztes muss noch der Routinen-Hauptteil programmiert werden:



Der Routinen-Hauptteil

Für die Programmierung von Funktionen gibt es keine Einschränkungen. Sie können alle Kontrollanweisungen – also Bedingungen, Schleifen etc. – verwenden. Auch Cursor und Fehlerbedingungen unterliegen keinen Besonderheiten. Für meine Funktion brauchen wir nicht viel: Ich definiere zunächst zwei Variablen, anschließend prüfe ich mit der Anweisung IF EXISTS, ob es sich bei dem Artikel um Lagerware handelt. Ist das der Fall, ermittle ich den Auftragsbestand, der bislang nicht gebucht wurde, sowie den verfügbaren Lagerbestand. Die Differenz gebe ich als Return-Wert aus. Falls es sich bei dem Artikel um Kommissionsware handelt, liefert die Funktion derzeit den Wert NULL zurück.

Nun ist es an der Zeit, die Funktion zu testen:



Der Funktionsaufruf

7.8.1
Seite 8

Wenn Sie innerhalb einer Prozedur oder einer Funktion rechnen, sollten Sie den Wert NULL stets im Auge behalten. Achten Sie in der folgenden Grafik mal auf den Artikel mit Artikelnummer „00088“:

ARTIKELNUMMER	BEZEICHNUNG	BESCHAFFUNG	VERFUEGBARER_BESTAND
00001	Mauspad Gummies	L	338390
00002	Radiergummi (10)	L	-8003873
00003	Textmarker Rot (10)	L	200306
00004	Füllfederhalter	L	158281
00005	Kugelschreiber (10)	K	-
00006	Hefter	K	-
00088	Kabelbinder	L	-
44551	Notizblock	K	-
00013	Locher	L	68176

Das Ergebnis

Einige Artikel haben anscheinend keinen Bestand. Stimmt das wirklich?

```
Select
  Sum(Bestand)  as "Bestand", Sum(reserviert) as "Reservierte Menge",
 ( Select sum(Positionsmenge) from Auftragsposition where Artikelnummer = '00088'  and status < 2) as "Auftragsmenge"
from lagerbestand
where Artikelnummer = '00088';
```

Bestand	Reservierte Menge	Auftragsmenge
457950	12	-

Überprüfung der Ergebnisse

Anscheinend liefert unsere Funktion falsche Ergebnisse, denn wir haben vom Artikel „00088“ 457.938 Stück im Lager. Wo also liegt der Fehler? Das Problem hat eigentlich nicht direkt etwas mit der Funktion zu tun. Vielmehr resultiert es aus den Berechnungen und dem Wert NULL. Sie können in der vorangegangenen Abbildung erkennen, dass derzeit kein Auftragsbestand für diesen Artikel berücksichtigt werden muss. Die Auftragsmenge liefert den Wert NULL, und immer wenn Sie mit dem Wert NULL rechnen, erhalten Sie als Ergebnis ebenfalls den Wert NULL. In meinem Beispiel muss ich also dafür sorgen, dass der Wert NULL ersetzt wird, wenn keine zu berücksichtigenden Aufträge vorliegen:

```
THEN
        Set #LBestand =  (Select  SUM(  ifnull(BESTAND,0)  - ifnull(reserviert , 0) )
                    FROM LAGERBESTAND
                    Where Artikelnummer = I_Artikelnummer
                    GROUP BY ARTIKELNUMMER) ;

        Set #ABestand = (Select  sum( Positionsmenge  )
                    from Auftragsposition
                    where status < 2   and Artikelnummer = I_Artikelnummer
                    group by  Artikelnummer );
     return   ifnull(#LBestand, 0)  -  ifnull(#ABestand, 0) ;
  |
End if;
```

Korrektur der Funktion

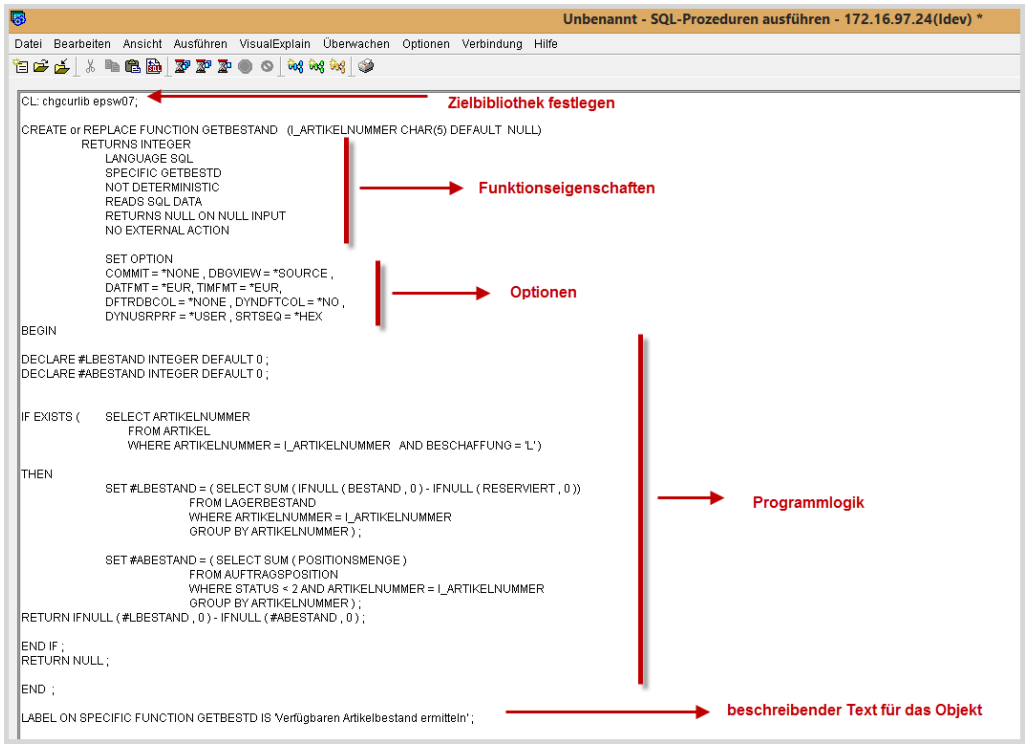
Und jetzt schaut das Ergebnis deutlich besser aus:

ARTIKELNUMMER	BEZEICHNUNG	BESCHAFFUNG	VERFUEGBARER_BESTAND
00001	Mauspad Gummies	L	338390
00002	Radiergummi (10)	L	-8003873
00003	Textmarker Rot (10)	L	200306
00004	Füllfederhalter	L	158281
00088	Kabelbinder	L	457938
00013	Locher	L	68176
00008	Kugelschreiber (5)	L	-471513
10002	Bürostuhl (ROT)	L	173239

Das endgültige Ergebnis

7.8.1.2 Skalare SQL-Funktionen in der Prozedurumgebung erstellen

Natürlich müssen wir die Funktion nicht zwingend mit dem grafischen Interface des System i Navigator erstellen. Wir können dafür auch die Prozedurumgebung des System i Navigator oder die SQL-Skriptumgebung von Access Client Solution verwenden oder einen beliebigen Texteditor. Auch RDI kann für die Objekterstellung genutzt werden (siehe Kapitel 7.4.7).



Funktionen in der Prozedurumgebung des System i Navigator erstellen

Schreiben wir Funktionen, lautet der Erstellungsbefehl CREATE OR REPLACE FUNCTION.

Es folgen in Klammern die Input-Parameter. Mit dem Schlüsselwort RETURNS geben wir bekannt, welchen Returnwert wir erwarten. Anschließend folgen die Funktionseigenschaften, die wir auf den vorangegangenen Seiten bereits besprochen haben. Nicht vergessen darf man die Optionen. Sie kennen deren Bedeutung bereits aus dem Kapitel „SQL-Prozeduren".

Der Programmcode kann alle SQL-Befehle enthalten, die Sie in den vorangegangenen Kapiteln bereits kennengelernt haben. Neu ist lediglich die Anweisung RETURN. Sie bestimmt den Rückgabewert. In der Abbildung oben sehen Sie zusätzlich die Anweisung LABEL ON SPECIFIC FUNCTION. Sie bestimmt den beschreibenden Text des auszuführenden Programmobjekts.

7.8.2 Tabellenfunktionen (UDTF)

Seit Release V5R2 können Funktionen als Ergebnis ganze Tabellen zurückgeben.

Der Unterschied zu den skalaren SQL-Funktionen besteht im Wesentlichen in der Parameterrückgabe. Im Gegensatz zu den bisher gezeigten Funktionen wird der Parameter RETURNS durch RETURNS TABLE ersetzt. Dieser Parameter ermöglicht es, die Tabellendefinition direkt anzugeben. Er beinhaltet sozusagen die Tabellendefinition. Und Sie können in der Funktion nur einmal die Anweisung RETURN codieren.

Schauen wir uns hierzu wieder ein Beispiel an. Dieses Mal soll unsere Funktion den verfügbaren Bestand in Form einer Tabelle zurückgeben. Die Business-Regeln bleiben bestehen. Allerdings soll die Funktion jetzt alle Artikelnummern inklusive den verfügbaren Beständen zurückgeben:

```
CREATE  or REPLACE FUNCTION  GetBestandAsTable ( )
RETURNS  TABLE
          (
          Artnr CHAR(5),
          Bestand Integer
          )

LANGUAGE  SQL
READS  SQL  DATA
NO  EXTERNAL ACTION
NOT  DETERMINISTIC
SET OPTION
DATFMT = *EUR,
TIMFMT = *EUR,
COMMIT = *NONE

RETURN
With
#Lagerbestand as
(
SELECT ARTIKELNUMMER, SUM(Ifnull(BESTAND, 0) -
IFNULL(RESERVIERT, 0)) AS MENGELAGER
FROM LAGERBESTAND
GROUP BY ARTIKELNUMMER
),
```

7.8.2
Seite 2

```
#Auftragsposition AS
(
Select   artikelnummer, sum(Positionsmenge) As MengeAuftrag
from Auftragsposition  join artikel   USING(Artikelnummer)
where status < 2  and beschaffung = 'L'
group by  Artikelnummer
)

Select  Artikelnummer, IFNULL(mengeLager, 0) -
IFNULL(MengeAuftrag, 0)
from #Lagerbestand left join #Auftragsposition
Using(Artikelnummer)
;
```

Die Funktion GetBestandAsTable erhält keine Eingabeparameter und liefert eine Tabelle, die aus zwei Spalten besteht, nämlich der Artikelnummer und dem verfügbaren Bestand. Beide Spalten deklarieren wir in den Eigenschaften, indem wir Folgendes codieren:

```
RETURNS  TABLE
         (
         Artnr CHAR(5),
         Bestand Integer
         )
```

Unsere Logik besteht aus nur einem SQL-Statement, das die Tabelle aufbaut. Das Result-Set des Statements muss dem Funktionsinterface entsprechen – also die Artikelnummer und den Bestand liefern.

Tabellenfunktionen können in jeder FROM-Klausel verwendet werden. Beachten müssen Sie lediglich das Schlüsselwort TABLE. Und Sie dürfen nicht vergessen, der temporären Tabelle einen Namen zu geben.

```
Select * from Table(GetBestandAsTable()) as Bestand;
```

Tabellenfunktionen, die aus nur einem SQL-Statement bestehen, sieht man häufig. Fehlerhandling oder andere Programmiertechniken fehlen in diesen Beispielen.

Und noch etwas fällt auf: Oftmals ist die Business-Logik so komplex, dass Sie sich nicht mit einem Statement abbilden lässt. Dadurch werden Zwischenergebnisse erforderlich.

Schauen Sie sich die folgende Funktion an:

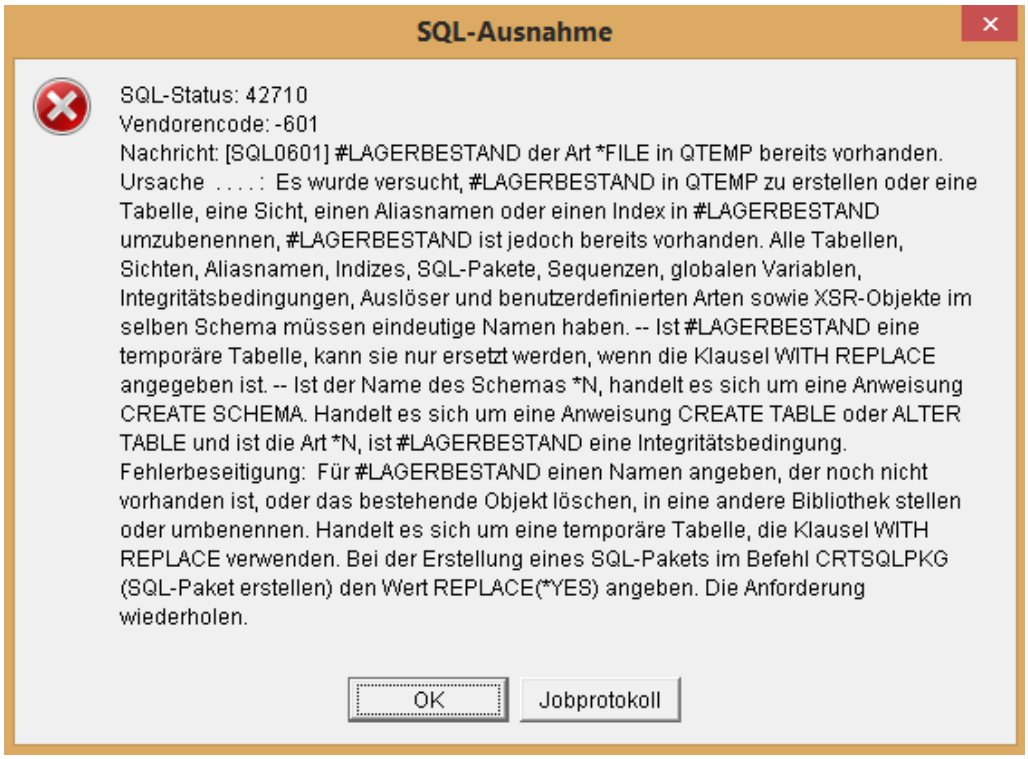
```
CREATE  OR REPLACE  FUNCTION  GETBESTANDASTABLE ( )
RETURNS  TABLE       (
          ARTNR CHAR(5),
          BESTAND INTEGER
          )

LANGUAGE  SQL
MODIFIES   SQL  DATA
NO  EXTERNAL ACTION
NOT  DETERMINISTIC
SET OPTION
DATFMT = *EUR,
TIMFMT = *EUR,
COMMIT = *NONE

BEGIN

CREATE TABLE QTEMP/#LAGERBESTAND
AS
(
SELECT ARTIKELNUMMER, SUM(IFNULL(BESTAND, 0) -
IFNULL(RESERVIERT, 0)) AS MENGELAGER
FROM LAGERBESTAND
GROUP BY ARTIKELNUMMER
) WITH DATA;

CREATE TABLE QTEMP/#AUFTRAGSPOSITION
AS
(
SELECT   ARTIKELNUMMER, SUM(POSITIONSMENGE) AS MENGEAUFTRAG
FROM AUFTRAGSPOSITION  JOIN ARTIKEL   USING(ARTIKELNUMMER)
WHERE STATUS < 2  AND BESCHAFFUNG = 'L'
GROUP BY  ARTIKELNUMMER
) WITH DATA;
```

7.8.2
Seite 4

```
RETURN
      SELECT  ARTIKELNUMMER,
IFNULL(MENGELAGER, 0) - IFNULL(MENGEAUFTRAG, 0)
      FROM #LAGERBESTAND LEFT JOIN #AUFTRAGSPOSITION
USING(ARTIKELNUMMER) ;
END;
```

Anstatt einer Common Table Expression werden in diesem Beispiel zunächst zwei Datenmengen in der Bibliothek QTEMP aufgebaut. Diese Zwischenergebnisse werden anschließend vereint und über die Anweisung RETURN ausgegeben. So weit, so gut.

Doch was passiert, wenn die Funktion innerhalb eines Jobs mehrfach aufgerufen wird?



Die Fehlermeldung

Derzeit können Sie die Anweisung CREATE or REPLACE TABLE nicht verwenden, wenn die Tabelle mit einem SELECT-Statement aufgebaut wird. Somit stürzt die Funktion beim zweiten Aufruf ab, da die Zwischentabelle schon existiert. Dieses Problem können Sie nur mit einem Handler lösen. Allerdings dürfen Sie in einer Tabellenfunktion die Anweisung RETURN nur ein einziges Mal verwenden. Und falls Sie die Funktion beenden ohne die Anweisung RETURN auszuführen, meldet die UDTF erneut einen Fehler. Die Lösung ist demzufolge etwas umständlich und muss mit einem CONTINUE-Handler gelöst werden:

```
CREATE  OR REPLACE FUNCTION  GETBESTANDASTABLE ( )
RETURNS  TABLE
            (
            ARTNR CHAR(5),
            BESTAND INTEGER
            )

LANGUAGE  SQL
MODIFIES   SQL  DATA
NO  EXTERNAL ACTION
NOT  DETERMINISTIC
SET OPTION
DATFMT = *EUR,
TIMFMT = *EUR,
COMMIT = *NONE
BEGIN

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
        IF EXISTS (SELECT 1 FROM QTEMP/#LAGERBESTAND)
        THEN
            DROP TABLE QTEMP/#LAGERBESTAND;
            CREATE TABLE QTEMP/#LAGERBESTAND AS
            (
            SELECT ARTIKELNUMMER,
            SUM(IFNULL(BESTAND, 0) - IFNULL(RESERVIERT, 0)) AS MENGELAGER
            FROM LAGERBESTAND
            GROUP BY ARTIKELNUMMER
            ) WITH DATA;
        END IF;
```

7.8.2
Seite 6

```
        IF EXISTS (SELECT 1 FROM QTEMP/#AUFTRAGSPOSITION)
        THEN
            DROP TABLE QTEMP/#AUFTRAGSPOSITION;
            CREATE TABLE QTEMP/#AUFTRAGSPOSITION  AS
            (
            SELECT ARTIKELNUMMER,
            SUM(POSITIONSMENGE) AS MENGEAUFTRAG
            FROM AUFTRAGSPOSITION  JOIN ARTIKEL   USING(ARTIKELNUMMER)
            WHERE STATUS < 2  AND BESCHAFFUNG = 'L'
            GROUP BY  ARTIKELNUMMER
            ) WITH DATA;

        END IF;

END;
```

Nachdem Sie den Handler integriert haben, können Sie die Funktion beliebig oft aufrufen.

Es bleibt allerdings das Problem der Zwischentabellen bestehen. Fazit: Es ist zwar eine Lösung, aber sicherlich **keine gute Lösung!**

7.8.3 Pipelined Table Function

Eine pipelined Tabellenfunktion ist eine Alternative zur Tabellenfunktion. In der pipelined Tabellenfunktion verwenden wir allerdings die Anweisung PIPE, um die Zeilen der Tabellenfunktion gesteuert an den Aufrufer zurückzugeben. Warum bzw. wann bieten sich die pipelined Tabellenfunktionen an?

Variablendeklarationen und Fehlerprüfungen fehlen in der Regel in den UDTFs. Zudem ist es schwer, komplexe Logiken zu integrieren, ohne ein „unleserliches" SQL-Statement zu verwenden, und selbst dann wird man irgendwann an seine Grenzen stoßen. Stellen Sie sich beispielsweise vor, dass wir unsere Bestandsprüfung noch um die aktuell vorliegenden Bestellungen und deren Liefertermine ergänzen müssten?

Derartig komplexe Businessregeln lassen sich irgendwann nicht mehr so einfach in ein Statement integrieren und häufig führt dies zu noch mehr Zwischenlösungen (QTEMP) oder gar zu komplexen Hostprogrammen (RPG/COBOL) mit vielen kleinen SQL-Statements. In einer pipelinded Tabellenfunktion können Sie die Logik ausprogrammieren und dürfen dafür viele SQL-Statements verwenden, die Sie hoffentlich auch noch in Jahren verstehen und ggf. modifizieren können.

Pipelined Tabellenfunktionen bringen also insbesondere dann Vorteile, wenn große Datenmengen aufgrund komplexer Abfragen erstellt werden, die ggf. sogar in temporären Tabellen im Arbeitsspeicher oder auf der Platte zwischengespeichert werden, um dann in einem nächsten Schritt weiterverarbeitet zu werden. Mit Hilfe der neuen Anweisung PIPE ist es möglich, die Daten besser zu verifizieren und spezifischer zu selektieren. Zwar werden die Zeilen der pipelined Tabellenfunktionen Zeile für Zeile verarbeitet, aber das Erstellen einer temporären Tabelle (QTEMP) entfällt. In diesen Fällen kann ggf. die Performance verbessert werden und die Auslastung des Arbeitsspeichers wird ebenfalls deutlich reduziert. Hinzu kommt, dass Fehler in einfachen Tabellenfunktionen nur aufwendig behandelt werden können. In pipelined Funktionen ist dies ohne Weiteres möglich.

7.8.3
Seite 2

Wie sehen die pipelined Tabellenfunktionen aus? Schauen wir uns anhand unserer Aufgabenstellung solch eine Funktion an.

Die ersten Definitionen sind identisch:

```
CREATE  OR REPLACE FUNCTION  GETBESTANDASTABLE ( )
RETURNS  TABLE        (
          ARTNR CHAR(5),
          BESTAND INTEGER
          )

LANGUAGE  SQL
MODIFIES   SQL  DATA
NO  EXTERNAL ACTION
NOT  DETERMINISTIC
SET OPTION
DATFMT = *EUR,
TIMFMT = *EUR,
COMMIT = *NONE
```

In einer pipelined Tabellenfunktion können wir beliebig viele Variablen, Cursor und Händlerdefinitionen verwenden:

```
Begin
-- LOKALE VARIABLEN
DECLARE #ARTNR CHAR(5) DEFAULT ' ';
DECLARE #POSMENGE INTEGER DEFAULT 0;
DECLARE #AUFTRAGSBESTAND INTEGER DEFAULT 0;
DECLARE #LAGERBESTAND INTEGER DEFAULT 0;
DECLARE #GESAMTLAGERBESTAND INTEGER DEFAULT 0;
DECLARE #ENDE SMALLINT DEFAULT 0;

--CURSOR
DECLARE C1 CURSOR FOR
SELECT ARTIKELNUMMER, SUM( IFNULL(LAGERBESTAND.BESTAND, 0) -
IFNULL(RESERVIERT, 0))
FROM ARTIKEL LEFT JOIN LAGERBESTAND USING(ARTIKELNUMMER)
WHERE BESCHAFFUNG = 'L'
GROUP BY ARTIKELNUMMER;

DECLARE C2 CURSOR FOR
SELECT    POSITIONSMENGE
FROM AUFTRAGSPOSITION  JOIN ARTIKEL USING(ARTIKELNUMMER)
WHERE BESCHAFFUNG = 'L'  AND  STATUS < 2  AND ARTIKELNUMMER = #ARTNR;

DECLARE CONTINUE HANDLER FOR NOT FOUND
        BEGIN
           SET #ENDE = 1;
        END;
```

7.8.3
Seite 4

Wir programmieren wie wir es bereits von Prozeduren und skalaren Funktionen kennen:

```
OPEN C1;

FETCH C1 INTO #ARTNR, #LAGERBESTAND;

WHILE #ENDE = 0
DO

            OPEN C2;
            FETCH C2  INTO    #POSMENGE;
            WHILE #ENDE = 0
            DO
               IF #POSMENGE IS NULL
                  THEN SET #POSMENGE   =  0;
               END IF;

               SET #AUFTRAGSBESTAND = #AUFTRAGSBESTAND + #POSMENGE;
               FETCH C2 INTO  #POSMENGE;
               END WHILE;
               CLOSE C2;

               SET #GESAMTLAGERBESTAND = #LAGERBESTAND - #AUFTRAGSBESTAND;
```

Der entscheidende Unterschied besteht in der Anweisung PIPE. Hierdurch bestimmen wir die Ausgabe und den Ausgabezeitpunkt:

```
PIPE (#ARTNR , #GESAMTLAGERBESTAND);

               SET #AUFTRAGSBESTAND = 0;
               SET #ENDE = 0;

FETCH C1 INTO #ARTNR, #LAGERBESTAND;
END WHILE;
CLOSE C1;
RETURN; -- ENDE DER UDTF (OHNE RÜCKGABE-WERT)
END;
```

Beachten Sie, dass die Ausgabe zeilenweise erfolgt!
In der Hauptabfrage merken Sie von diesem Verhalten jedoch nichts:

```
Select *
from table(GetBestandAsTable()) as Bestand
order by Artnr;
```

AR...	BESTAND
00001	338390
00002	-8003873
00003	200306
00004	158281
00008	-471513
00013	68176
00088	457938
10002	173239

Das Ergebnis unserer Tabellenfunktion

Achtung!

Ich weiß, dass mein Beispiel etwas umständlich programmiert ist und sicherlich optimiert werden kann. Das Beispiel sollte lediglich verdeutlichen, dass Schleifen und Bedingungen in einer pipelined Tabellenfunktion sehr einfach genutzt werden können. Bedenken Sie immer, dass SQL dann am effektivsten ist, wenn die Daten in einem Schritt – als ein Result Set – erstellt werden. Aber bevor Sie mit Zwischenergebnissen arbeiten und unnötigerweise Tabellen erzeugen, die nicht indiziert werden können, sind pipelined Tabellenfunktionen eine gute Alternative. Mit anderen Worten: Ob pipelined Tabellenfunktionen eine gute Wahl sind, kann nur kontextbezogen entschieden werden!

7.8.3
Seite 6

7.8.4 Abgeleitete Funktionen

Abgeleitete SQL-Funktionen erlauben uns die erneute Definition einer bereits bestehenden Funktion. Dabei ist es zunächst unwichtig, ob es sich um eine bestehende SQL-Built-in-Funktion oder um eine benutzerdefinierte SQL-Funktion handelt. Für abgeleitete Funktionen können wir andere Namen und/oder unterschiedliche Parametertypen verwenden. Eine Ausnahme bilden SQL-Tabellenfunktionen. Sie können nicht abgeleitet werden.

Abgeleitete Funktionen werden allgemein dann eingesetzt, wenn es darum geht, aufbauend auf einer bereits bestehenden Funktion ähnliche Lösungen zu realisieren. Unterschiede zu der bereits bestehenden Funktion und der neu abzubildenden Funktion können zum Beispiel unterschiedliche Datentypen sein, die alternativ verarbeitet werden sollen. Ein weiterer Einsatzbereich kann nicht nur funktionale Hintergründe haben, sondern auch auf Basis von ästhetischen Gesichtspunkten erforderlich sein – nämlich dann, wenn eine bereits bestehende Funktion schlicht umbenannt werden soll, weil der Name nicht wirklich passend gewählt wurde.

Das Umbenennen einer Funktion kann aber auch noch andere Gründe haben als die Angabe eines vielleicht aussagekräftigeren Funktionsnamens. Denken wir an die unterschiedlichen Anforderungen, die durch die verschiedenen SQL-Dialekte gestellt werden, wie zum Beispiel durch Microsoft SQL. Diese Funktionen werden dann entsprechend auf die Anforderungen der Gegenseite abgestimmt, also zum Beispiel auf die Microsoft-Spezifika.

Wie Sie sich vielleicht jetzt schon vorstellen können, gestaltet sich die Anweisung CREATE FUNCTION für solche abgeleiteten Funktionen wesentlich einfacher als wir es bei den zuvor erläuterten Funktionen kennengelernt haben:

```
CREATE FUNCTION VERFUEGBAR( ARTIKEL CHARACTER (5) )
RETURNS  INTEGER
SPECIFIC VERFUEGBAR   SOURCE  GetBestand ;
```

Sie sehen anhand des Beispiels, dass die Definition sehr einfach ist. Der Parameter SOURCE enthält den Funktionsnamen der Funktion, die als Pate für die neu zu erstellende Funktion dienen soll. Zusätzlich besteht die Möglichkeit, die Parameterdefinition zu ändern.

7.8.4
Seite 2

7.9 Datenbanktrigger

Datenbanktrigger (dt. Datenbankauslöser) werden meistens nur Trigger (Auslöser) genannt. Dabei handelt es sich um gespeicherte Programme, die immer dann aufgerufen werden, wenn eine bestimmte Datenbankoperation (INSERT, UPDATE, DELETE) ausgeführt wird. Trigger können die Ausführung dieser Datenbankoperation verhindern, erlauben oder auch ergänzen. Heute können Trigger nicht nur für die Manipulation ganzer Zeilen (Zeilentrigger) sondern auch für die Manipulation einzelner Zeilen (Spaltentrigger) definiert werden.

Maximal können heute 300 Datenbanktrigger für eine Tabelle definiert werden.

Vor- und Nachteile von Triggern
Trigger werden häufig eingesetzt, um die Datenkonsistenz zu wahren, können aber auch Protokollaufgaben wahrnehmen. Welche Zwecke die Trigger erfüllen, ist weitestgehend anwendungsspezifisch und abhängig vom Datenbankdesign, aber auch von den „Vorlieben" der Programmierer. So gibt es Häuser, die mit Hilfe von Datenbanktriggern komplexe Anwendungslogiken realisieren, andere Verzichten hingegen vollständig auf den Einsatz von Datenbanktriggern. Um zu entscheiden, ob der Einsatz von Triggern in Ihrem Umfeld sinnvoll ist, sollten Sie sich mit den Vor- und Nachteilen von Triggern auseinandersetzen:

- Trigger sind anwendungsunabhängig, das heißt, nicht der Programmierer bestimmt den Zeitpunkt des Aufrufs, sondern die jeweilige Datenbankoperation. Dabei spielt das Interface keine Rolle. Es ist also egal, ob der entsprechende DELETE-, INSERT- oder UPDATE-Befehl interaktiv, in einem HLL-Programm (zum Beispiel RPG) oder in einer SQL-Prozedur oder -Funktion ausgelöst wird.
- Leichte Pflege. Sollten sich einmal die Businessregeln oder das Datenbankdesign in Ihrem Haus ändern, lassen sich die neuen Regeln schnell und einfach implementieren, da es meist ausreichend ist, die Triggerprogramme zu modifizieren. Die Applikation bleibt unberührt.
- Wiederverwendbarkeit, das heißt, die Funktionalität der Trigger seht automatisch allen Applikationen und allen Interfaces zur Verfügung.
- In einer Client-Server-Umgebung können Trigger helfen, die Geschäftslogiken in Client- und in Serverlogik zu splitten.
- Der Trigger wird synchron zur eigentlichen Datenbankoperation ausgeführt und verlängert dadurch den I/O-Prozess.
- Die Anwendungslogik ist anhand der Programme nicht mehr nachvollziehbar, da die Trigger automatisch von der Datenbank und nicht aus den Programmen heraus aktiviert werden.

7.9
Seite 2

7.9.1 Datenbanktrigger – Eigenschaften

Bevor wir anfangen Trigger zu codieren, müssen wir uns zunächst einige grundlegende Eigenschaften anschauen.

Das Triggerevent
Soll ein Trigger erstellt werden, muss das Triggerevent für die Registrierung benannt werden. Hierbei unterscheiden wir heute vier verschiedene Events:

INSERT
Der Trigger wird beim Einfügen von Zeilen aufgerufen.
Der Insert-Trigger wird für eine INSERT-Anweisung/WRITE-Anweisung, aber auch für die Anweisung MERGE ausgeführt, wenn diese eine INSERT-Klausel enthält.

UPDATE
Der Trigger wird beim Aktualisieren von Zeilen aufgerufen.
UPDATE-TRIGGER werden aktiviert, wenn die Tabelle mit einem UPDATE oder mit einer MERGE-Anweisung mit UPDATE-Klausel modifiziert wird.

DELETE
Der Trigger wird beim Löschen von Zeilen aktiviert.
DELETE-TRIGGER reagieren auf die Anweisung DELETE oder MERGE mit DELETE-Klausel.

READ
Es gibt auch Lesetrigger. Dabei wird der Trigger beim Lesen einer Zeile ausgeführt. Hierbei handelt es sich jedoch nicht um SQL-Trigger, so dass diese Trigger auch nur mit den Befehlen ADDPFTRG (Add Physical File Trigger) registriert werden können. Außerdem können nur externe DB-Trigger registriert werden.

Der Triggerzeitpunkt
Neben dem Triggerevent spielt vor allen Dingen der Triggerzeitpunkt eine entscheidende Rolle. Hierbei können wir unterscheiden zwischen:

BEFORE-Trigger
Das Trigger-Programm wird aufgerufen, bevor die Zeile eingefügt, aktualisiert oder gelöscht wird. Dies ermöglicht eine Ergänzung oder auch eine Überprüfung der jeweiligen Aktion. Before-Trigger können so beispielsweise dazu verwendet werden, vor dem INSERT oder UPDATE die aktuelle Zeitmarke, einen laufenden Zähler oder den aktuellen Benutzer zu ergänzen.

AFTER-Trigger
Das Trigger-Programm wird aufgerufen, nachdem die Zeile eingefügt, aktualisiert oder gelöscht wurde. After-Trigger können zum Beispiel dazu verwendet werden, um Summen, die über die aktualisierten Datensätze ermittelt werden, in anderen Dateien fortzuschreiben.

INSTEAD OF-Trigger
Haben wir einen Instead-of-Trigger (V5R4), wird das eigentliche Datenbankereignis ersetzt. Eine Update-Anweisung könnte beispielsweise „umgewandelt" werden in eine INSERT-Operation. Für Instead-Of-Trigger existieren allerdings einige Restriktionen:

- Sie können nur mit dem Befehl CREATE TRIGGER implementiert werden.
- Sie können nur für SQL-Views verwendet werden.
- Es können nur SQL-Trigger verwendet werden. Externe Datenbanktrigger sind nicht erlaubt.

Die Granularität
Die Granularität bestimmt, wie oft ein Trigger ausgelöst wird. Man unterscheidet zwischen einem Zeilentrigger (FOR EACH ROW) und einem Anweisungstrigger (FOR EACH STATEMENT).

Zeilentrigger (FOR EACH ROW) werden für jede betroffene Zeile in der Tabelle einmal ausgeführt. Ist keine Tabellenzeile betroffen, wird der Trigger nicht ausgeführt. Sind 100 Zeilen betroffen, wird der Trigger 100-mal ausgeführt.

Ein Anweisungstrigger (FOR EACH STATEMENT) wird hingegen erst ausgeführt, wenn die Datenbankoperationen beendet sind. Daher wird der Anweisungstrigger immer genau einmal ausgeführt – egal ob die Tabelle modifiziert wurde oder nicht.

Ob Sie Zeilen- oder Statement-Trigger verwenden sollten, hängt stets von der Aufgabenstellung ab. Es gilt lediglich zu beachten, dass **BEFORE-Trigger immer Zeilentrigger** sein müssen und dass Anweisungstrigger nicht mit dem Modus DB2ROW verwendet werden können.

Der Triggermodus
Zusätzlich zur Granularität kennt die DB2 for i den Triggermodus. Auf der DB2 for i unterscheiden wir zwischen dem Modus DB2ROW und DB2SQL. Worin unterscheiden sich die beiden Modi? Beim Modus DB2ROW wird der Trigger unmittelbar nach jeder Zeilenoperation ausgeführt, das heißt, Datenbankoperation und Triggerauslösung finden im Wechsel statt.

Der Modus DB2SQL hingegen sorgt dafür, dass die Triggerauslösung durch die verwendete Granularität bestimmt wird.

Aus diesem Grund können Sie den Triggermodus auch nicht beliebig mit der Granularität kombinieren. Die nachfolgende Darstellung zeigt die Kombinationsmöglichkeiten und die damit verbundenen Einschränkungen:

Granularität FOR EACH ROW				
Modus	**Zeitpunkt**	**Operation**	**Korrelations-variablen**	**Transitionstabelle**
DB2ROW (Auslöser für jede Zeilenoperation aktiviert)	BEFORE	DELETE	REFERENCING OLD (oRow)	nicht erlaubt
	AFTER oder INSTEAD OF	INSERT	REFERENCING NEW (nRow)	
		UPDATE	REFERENCING NEW (nRow) OLD (oRow)	
DB2SQL (Auslöser nach Abschluss aller Zeilenoperationen aktiviert)	AFTER oder INSTEAD OF	DELETE	REFERENCING OLD (oRow)	OLD Table (oTab)
		INSERT	REFERENCING NEW (nRow)	NEW Table (nTab)
		UPDATE	REFERENCING NEW (nRow) OLD (oRow)	OLD Table (oTab) NEW Table (nTab)

Granularität FOR EACH STATEMENT				
DB2ROW nicht erlaubt bei Granularität FOR EACH STATEMENT				
DB2SQL (Auslöser nach Abschluss aller Zeilenoperationen aktiviert)	AFTER	DELETE	nicht erlaubt	OLD Table (oTab)
		INSERT		NEW Table (nTab)
		UPDATE		OLD Table (oTab) NEW Table (nTab)

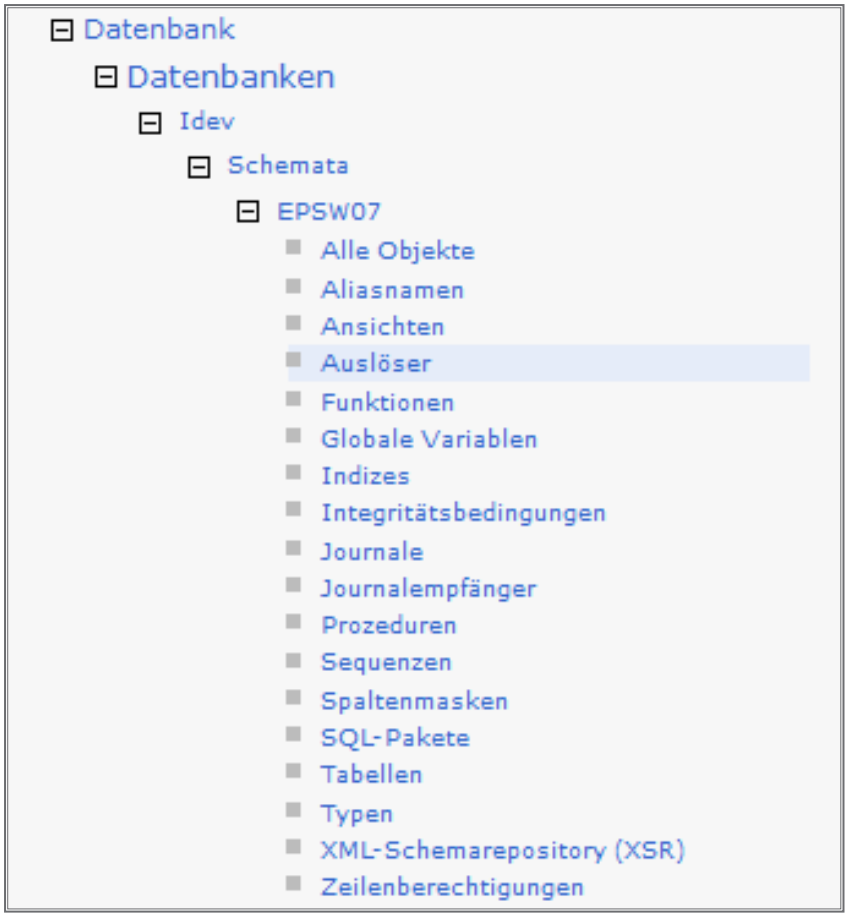
7.9.1
Seite 4

7.9.2 Datenbanktrigger erstellen – grafisches Interface

Jetzt wollen wir einen ersten Datenbanktrigger erstellen. In unserer Datenbank haben wir die Tabellen AUFTRAG und AUFTRAGSPOSITION. Da in der Tabelle AUFTRAG der Auftragswert gespeichert ist, wollen wir mit Hilfe eines Datenbanktriggers sicherstellen, dass dieser Wert automatisch modifiziert wird, wenn in der Positionstabelle eine Auftragsposition hinzugefügt wird.

Wir benötigen also einen INSERT-Trigger für die Positionstabelle, und wir nutzen zunächst einmal den Navigator for i, bevor wir uns mit der eigentlichen SQL-Syntax beschäftigen.

Hierzu öffnen wir den Navigator for i, stellen zunächst die Verbindung zum Server her und öffnen dann den Datenbankbereich:

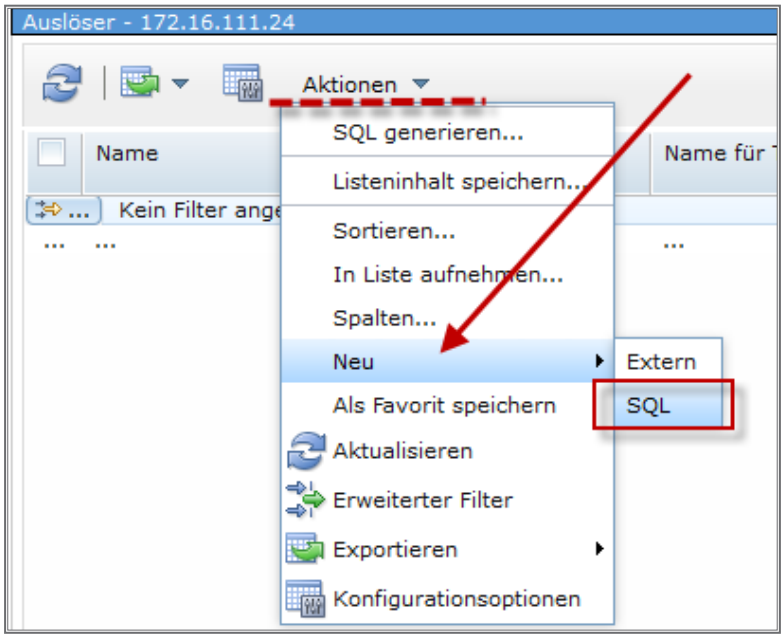


Datenbanktrigger im Navigator for i erstellen

Wenn Sie den Eintrag AUSLÖSER markiert haben, können Sie einen neuen SQL-Trigger erstellen. Wählen Sie hierfür den entsprechenden Eintrag in der Aktionsliste. Die folgende Abbildung zeigt, was gemeint ist.

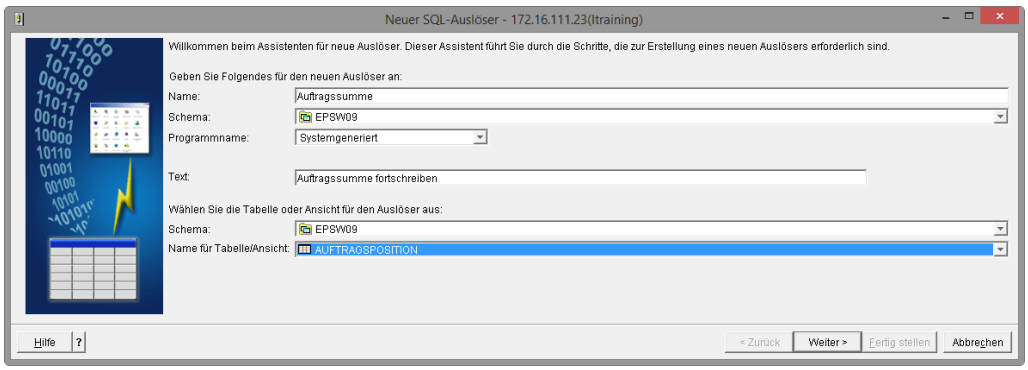
Ergänzung 38./2016

7.9.2
Seite 2



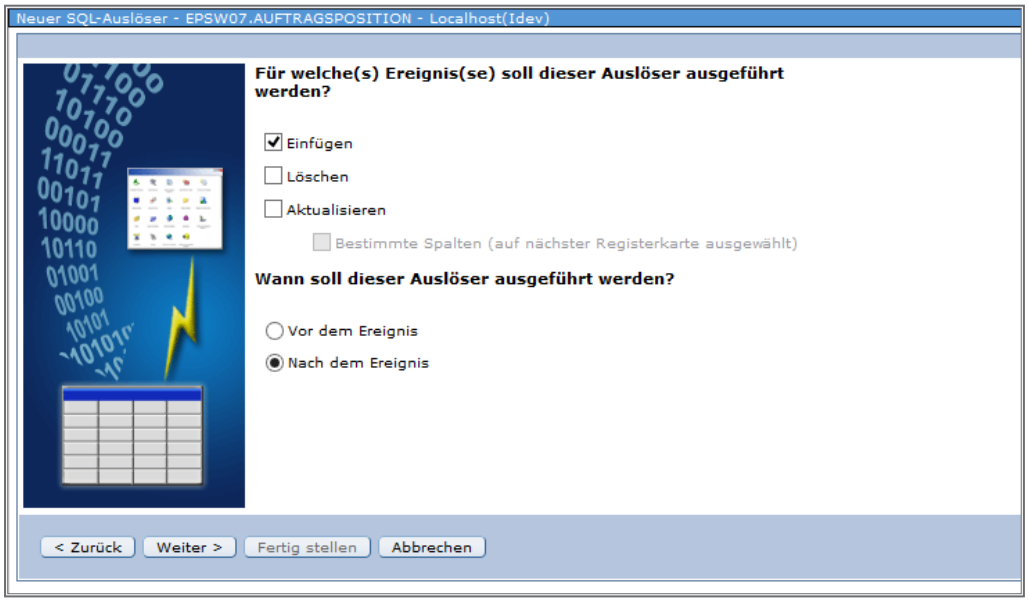
SQL-Trigger erstellen

Daraufhin öffnet sich ein Assistent, der Sie durch den Erstellungsvorgang führt. Auf der ersten Seite legen Sie lediglich den Triggernamen und die Triggertabelle fest. Beachten sollten Sie den Eintrag PROGRAMMNAME. Wie alle SQL-Objekte haben Datenbanktrigger zwei Namen, einen langen, den Sie im Feld NAME eintragen, und den systemspezifischen Kurznamen. Dies ist der PROGRAMMNAME. Wählen Sie also bitte einen „vernünftigen“ Kurznamen, wenn Sie sicherstellen wollen, dass Sie Ihre Trigger auch auf der Betriebssystemebene identifizieren können.



Grundparameter

Als Nächstes bestimmen Sie das Triggerevent und den Triggerzeitpunkt. Für meine Aufgabe entscheide ich mich für einen Insert-Trigger, der nach dem Insert in die Positionstabelle aufgerufen wird.



Triggerevent und Zeitpunkt festlegen

Und jetzt fehlt noch die Granularität. Ich möchte meinen Trigger nur einmal ausführen, und zwar nachdem die Zeilen in die Positionstabelle eingefügt wurden. Denn ich weiß, dass es nicht vorkommen kann, dass eine leere Zeile eingefügt wird oder eine Zeile ohne Positionswert. Mein Trigger wird also nie unnötig ausgeführt. Dies spricht für die Granularität FOR EACH STATEMENT.

Aber ich benötige auch Zugriff auf die Auftragsnummer der Positionstabelle, und dafür brauche ich Korrelationsvariablen, die ich nicht erhalte, wenn ich zu diesem Zeitpunkt die Granularität FOR EACH STATEMENT wähle.

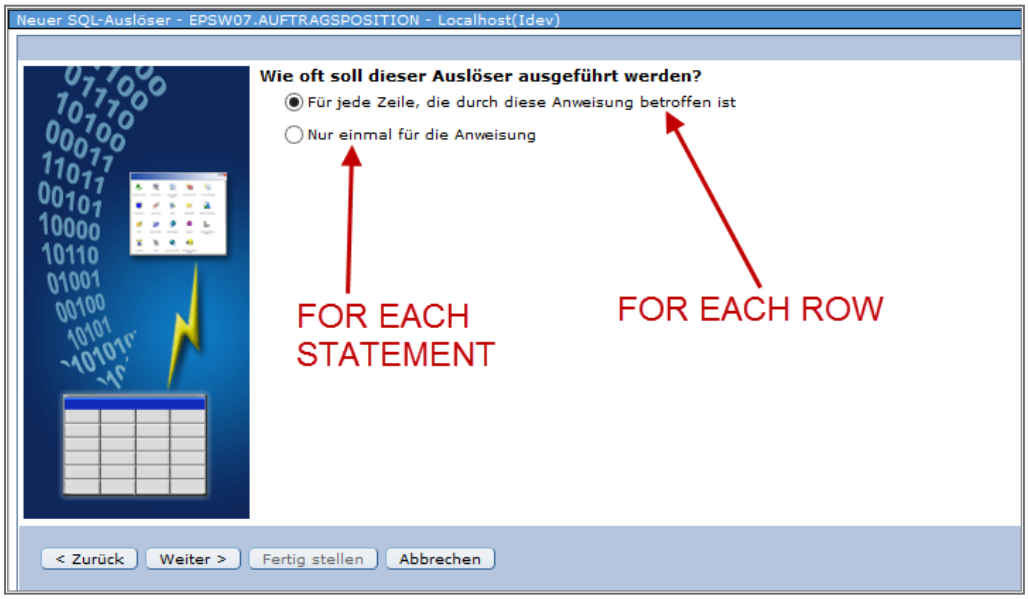
7.9.2
Seite 4

Schauen Sie sich noch einmal die nachfolgende Tabelle an:

Granularität FOR EACH ROW				
Modus	**Zeitpunkt**	**Operation**	**Korrelations-variablen**	**Transitionstabelle**
DB2ROW (Trigger wird für jede Zeile ausgeführt)	BEFORE	DELETE	REFERENCING OLD (oRow)	nicht erlaubt
	AFTER oder INSTEAD OF	INSERT	REFERENCING NEW (nRow)	
		UPDATE	REFERENCING NEW (nRow) OLD (oRow)	
DB2SQL (Auslöser nach Abschluss aller Zeilen-operationen aktiviert)	AFTER oder INSTEAD OF	DELETE	REFERENCING OLD (oRow)	OLD Table (oTab)
		INSERT	REFERENCING NEW (nRow)	NEW Table (nTab)
		UPDATE	REFERENCING NEW (nRow) OLD (oRow)	OLD Table (oTab) NEW Table (nTab)

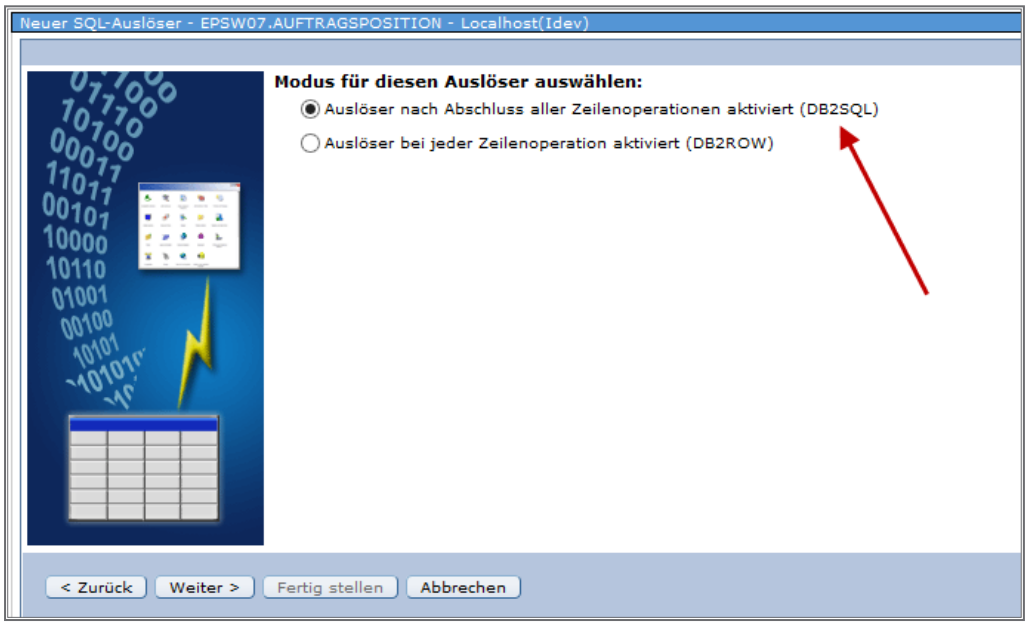
Granularität FOR EACH STATEMENT				
DB2ROW ist nicht erlaubt bei Granularität FOR EACH STATEMENT				
DB2SQL (Auslöser nach Abschluss aller Zeilen-operationen aktiviert)	AFTER	DELETE	nicht erlaubt	NEW Table (nTab)
		INSERT		OLD Table (oTab)
		UPDATE		NEW Table (nTab)

Ich entscheide mich daher an dieser Stelle für die Ausführung nach jeder Zeile…



Triggergranularität bestimmen

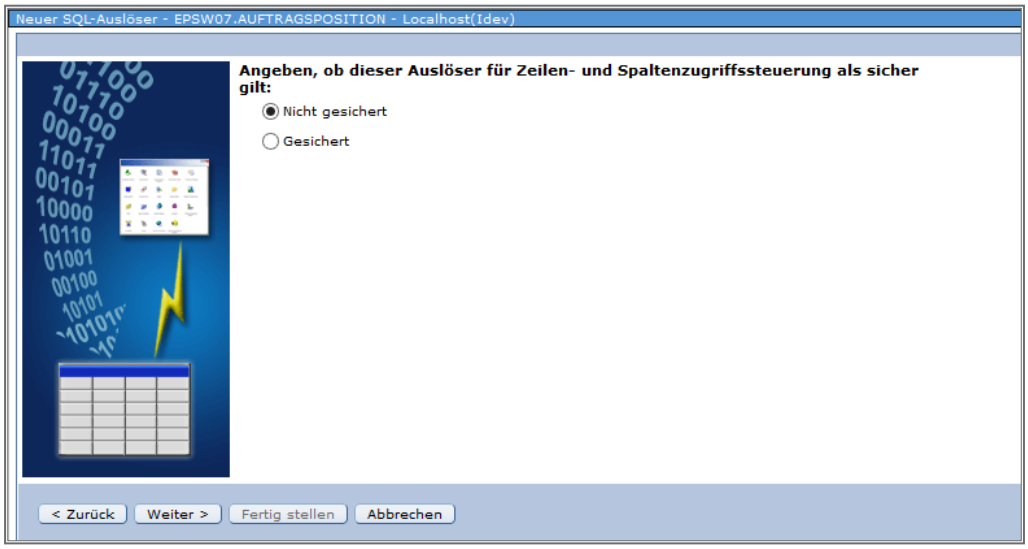
... und wähle dann im folgenden Dialog den Modus DB2SQL, um die Triggerausführung einmalig nach Abschluss der Positionseinfügungen auszulösen.



Modus DB2SQL oder DB2ROW

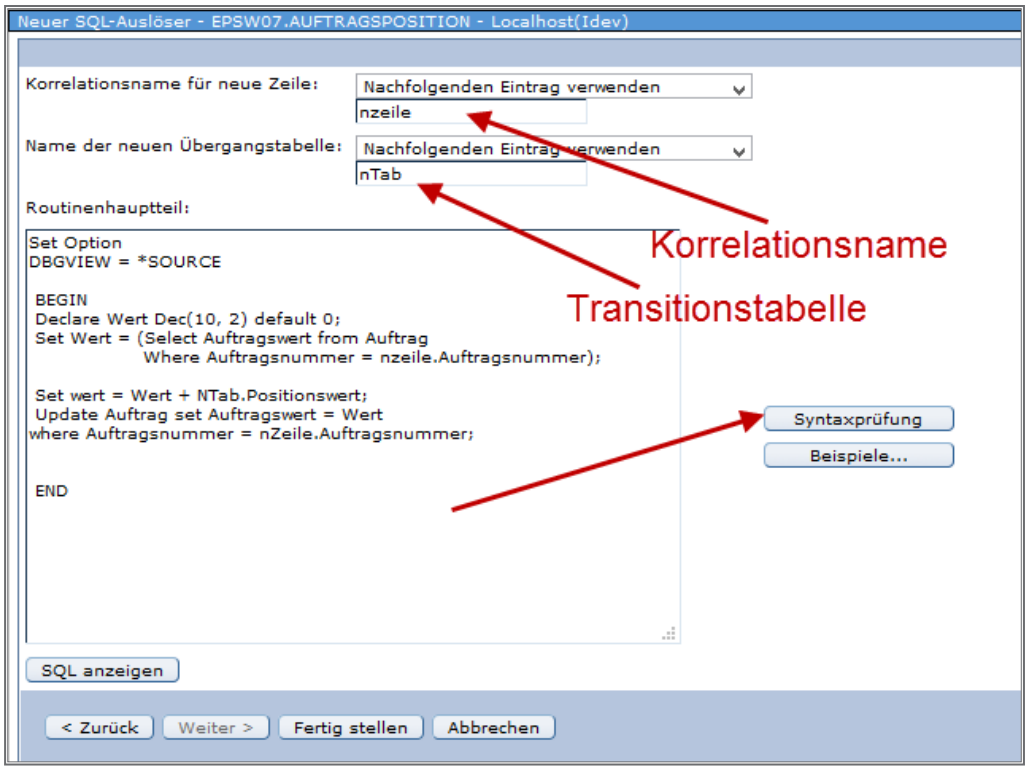
Falls Sie sich bereits in Release i7.2 befinden, erhalten Sie einen weiteren Dialog. Mit Hilfe dieses Dialogs spezifizieren Sie, ob der Trigger für Row and Column Access (RCAC) sicher ist. Die Eigenschaft SECURED muss festgelegt sein, wenn der Trigger auf Tabellen zugreift, für die RCAC-Bedingungen definiert wurden. Ist die Eigenschaft nicht gesetzt, wird der Trigger in diesen Fällen Fehler auslösen.

7.9.2
Seite 6



Die Eigenschaft SECURED i7.2

Jetzt fehlt noch ein wenig Programmlogik, dann ist der Trigger einsatzbereit.



Die Triggerlogik

Schauen wir uns die Logik etwas genauer an:

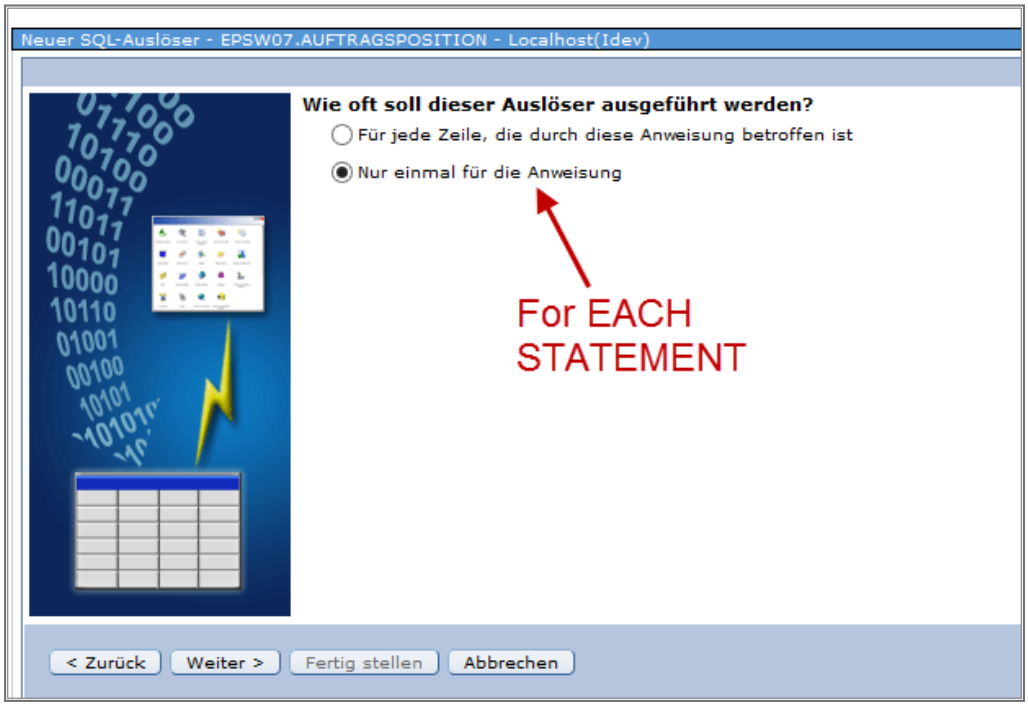
```
SET OPTION
DBGVIEW = *SOURCE

BEGIN
DECLARE WERT DEC(10, 2) DEFAULT 0;

SET WERT = (  SELECT AUFTRAGSWERT
              FROM AUFTRAG
              WHERE AUFTRAGSNUMMER = NZEILE.AUFTRAGSNUMMER);

SET WERT = WERT + NZEILE.POSITIONSWERT;

UPDATE AUFTRAG SET AUFTRAGSWERT = WERT
WHERE AUFTRAGSNUMMER = NZEILE.AUFTRAGSNUMMER;


END
```

In einem Trigger können Sie wie in Prozeduren und Funktionen eigene Variablen deklarieren. Hier im Beispiel wird zunächst der alte Auftragswert in der Auftragstabelle ermittelt, dann aktualisiert und anschließend zurückgeschrieben. Es geht mir nicht darum, zu beurteilen, ob man diese Logik nicht sehr viel besser hätte schreiben können. Vielmehr geht es mir darum, dass Sie Folgendes erkennen: Es ist problemlos möglich, die eingefügten Tabellenzeilen mit dem Korrelationsnamen NZEILE anzusprechen. Und noch etwas ist wichtig: In diesem Beispiel wird der **Trigger nur einmal** aufgerufen, aber der **UPDATE-Befehl wird n-Mal** ausgeführt!

7.9.2
Seite 8

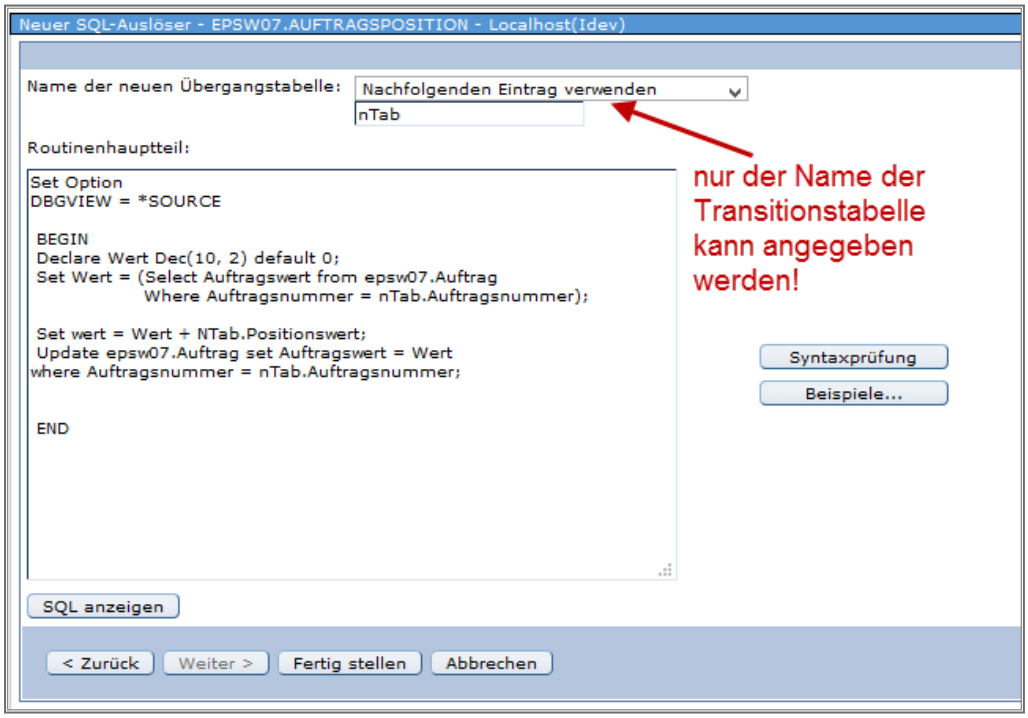
Schauen wir uns dieses Beispiel noch einmal an, nutzen dabei aber die Granularität FOR EACH STATEMENT.

Die ersten Eingabedialoge sind identisch: Wir vergeben den Triggernamen und benennen die Tabelle sowie das entsprechende Ereignis. Dieses Mal entscheiden wir uns aber für die Granularität FOR EACH STATEMENT:



Granularität FOR EACH STATEMENT

Der nachfolgende Dialog, der uns die Entscheidung zwischen DB2SQL und DB2ROW ermöglicht, entfällt. Nachdem wir den Dialog SECURED bzw. NOT SECURED bestätigt haben, gelangen wir sofort in das Skript-Fenster zur Eingabe unserer Programmlogik:



Triggerlogik FOR EACH STATEMENT

Es fällt sofort auf, dass Sie jetzt nur noch den Namen der Transitionstabelle verwenden können, und dieser Name ist nicht in der gleichen Weise nutzbar wie die Korrelationsnamen. Sie können in Ihrer Programmlogik daher mit dem Ausdruck nTab.Auftragsnummer die neue Auftragsnummer nicht adressieren. Mit Hilfe der Transitionstabellennamen können Sie nur alle eingefügten Zeilen in Ihrer Gesamtheit ansprechen, also als Result Set. Die Logik des Triggers muss also folgendermaßen geändert werden:

```
UPDATE AUFTRAG AFTRG SET AUFTRAGSWERT =

(SELECT SUM ( AP.POSITIONSWERT )
        FROM AUFTRAGSPOSITION AP
WHERE AP . AUFTRAGSNUMMER = AFTRG.AUFTRAGSNUMMER )

WHERE AFTRG.AUFTRAGSNUMMER IN
             (SELECT NTAB.AUFTRAGSNUMMER
              FROM NTAB) ;
```

7.9.2
Seite 10

Sie sehen, dass ich jetzt die Transitionstabelle in der FROM-Klausel eines SELECT-Statements verwende und auf diesem Weg alle Auftragsnummern selektiere, die eingefügt wurden. Für diese Auftragsnummern berechne ich mit Hilfe eines Korrelations-Subselects einfach den Auftragswert neu. Im Gegensatz zum zuvor gezeigten Beispiel wird das **UPDATE jetzt nur einmal** ausgeführt. Dafür werden aber alle Positionssätze der entsprechenden Auftragsnummer neu kumuliert. Auch diese Lösung ist gangbar! Welche Lösung die bessere Wahl ist, hängt – wie so oft – von der Datenlage ab.

7.9.3 Datenbanktrigger erstellen – CREATE TRIGGER

Verlassen wir jetzt das grafische Interface und schauen uns die entsprechende SQL-Syntax des Statements CREATE TRIGGER an:

```
CREATE or REPLACE TRIGGER INSERT_POSITION
       AFTER INSERT ON AUFTRAGSPOSITION
       REFERENCING NEW TABLE AS NTAB
       FOR EACH STATEMENT   MODE DB2SQL
PROGRAM NAME INSPOS
NOT SECURED
…CONCURRENT ACCESS RESOLUTION wait for outcome -- default
```

Wie bei allen anderen SQL-Objekten können wir auch bei Triggern die Klausel OR REPLACE verwenden und ersparen es uns auf diesem Weg, den Trigger zu löschen bevor er neu erstellt wird. Zudem bleiben so die Objektberechtigungen bestehen und müssen nicht neu vergeben werden. Anschließend folgen die Triggereigenschaften. Trigger haben sehr viel weniger Eigenschaften als Prozeduren und Funktionen. Für Trigger legen wir lediglich das Ereignis und die Tabelle fest:

```
AFTER INSERT ON AUFTRAGSPOSITION
```

Anschließend folgt der Transitionsname der Tabelle und/oder der Korrelationsname:

```
REFERENCING NEW nZeile NEW_TABLE nTab
```

Und die Granularität muss unter Berücksichtiung der dargestellten Kombinationsmöglichkeiten festgelegt werden:

```
FOR EACH STATEMENT    MODE DB2SQL
```

Jetzt kann noch der interne Kurzname benannt werden:

```
PROGRAM NAME INSPOS
```

7.9.3
Seite 2

Und es folgen die Eigenschaften SECURED bzw. NOT SECURED, wenn Sie mit RCAC arbeiten. Andernfalls können Sie es beim Standard NOT SECURED belassen und müssen die Eigenschaft nicht explizit benennen. Falls Sie mit Transaktionen arbeiten, können Sie dem Trigger mitteilen, wie er mit nicht festgeschriebenen Zeilen verfahren soll, indem Sie die Eigenschaft CONCURRENT ACCESS RESOLUTION verwenden. Arbeiten Sie nicht mit Transaktionen, so können Sie es beim Default WAIT FOR OUTCOME belassen und müssen die Eigenschaft nicht explizit verwenden.

Nach den Eigenschaften folgen wiederum die Optionen (SET OPTION). Hier sind einige Einschränkungen zu erwähnen:

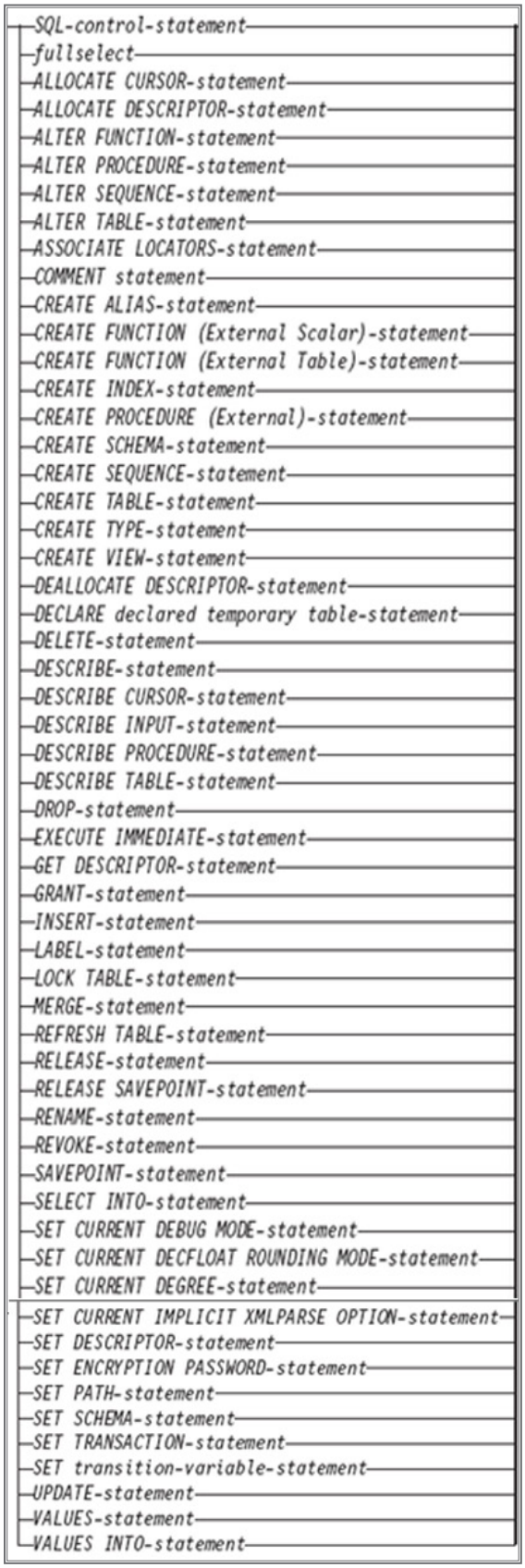
Die Optionen CNULIGN, CNULRQD, COMPILEOPT, EXTIND, NAMING und SQLCA sind in einem Trigger grundsätzlich nicht erlaubt. Die Optionen DATFMT, DATSEP, TIMFMT und TIMSEP können nicht verwendet werden, wenn Sie mit Korrelationsnamen (nZeile/oZeile) arbeiten. Die Option COMMIT können Sie zwar verwenden, sie wird aber ignoriert.

Anschließend folgen der BEGIN- und der END-Block:

```
cl: chgcurlib epsw07;

CREATE or REPLACE TRIGGER UpDateAuftragswert
    AFTER INSERT ON AUFTRAGSPOSITION
    REFERENCING NEW TABLE AS NTAB
    FOR EACH STATEMENT  MODE DB2SQL
    PROGRAM NAME UpdAFWert
    NOT SECURED

    SET OPTION
    DBGVIEW = *SOURCE

    BEGIN

    UPDATE AUFTRAG SET AUFTRAGSWERT =
            (   SELECT SUM ( AUFTRAGSPOSITION . POSITIONSWERT )
                FROM AUFTRAGSPOSITION
                WHERE  AUFTRAGSPOSITION . AUFTRAGSNUMMER = AUFTRAG . AUFTRAGSNUMMER )
    WHERE AUFTRAG . AUFTRAGSNUMMER IN ( SELECT NTAB . AUFTRAGSNUMMER FROM NTAB ) ;

END  ;
comment on Trigger  Insert_Position is 'Neuberechnung des Auftragswertes-Update auf Auftragstabelle';
```

Der Quellcode

In einem Trigger können allerdings nicht alle Anweisungen codiert werden. Nachfolgende Grafik zeigt Ihnen, welche Anweisungen Sie verwenden dürfen:



Erlaubte Trigger-Anweisungen

7.9.3
Seite 4

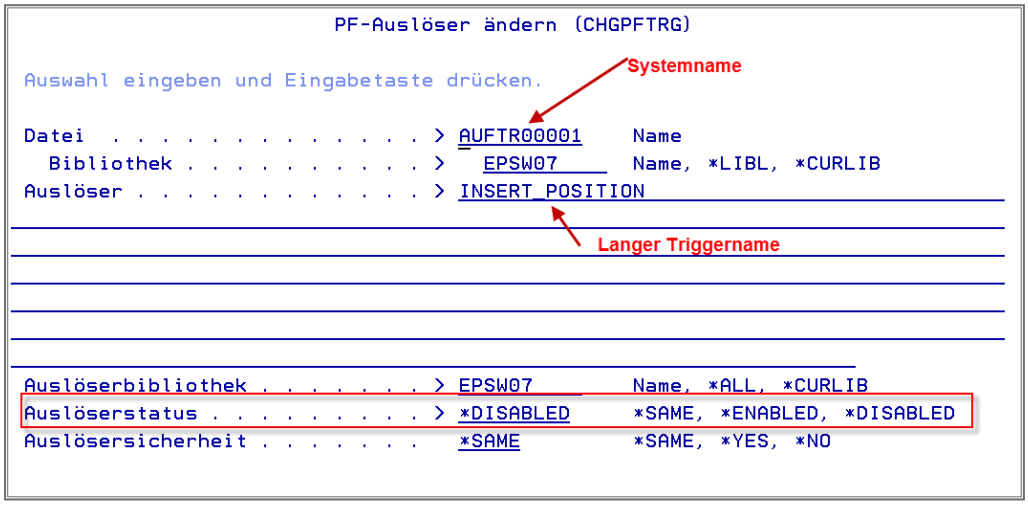
7.9.4 Triggeradministration

Trigger sind im Allgemeinen nützlich und behindern nicht die eigentlichen Prozesse. Dennoch können Situationen auftreten, in denen die Trigger den Businessablauf stören. Stellen Sie sich in Bezug auf mein Beispiel vor, wir würden in der Nacht Auftragspositionen von einem anderen System übernehmen. Die Nachtverarbeitung stellt ebenfalls sicher, dass die zugehörigen Auftragsköpfe entsprechend aktualisiert werden. In diesem Szenario wäre mein Trigger eher hinderlich. Aus diesem Grund können Trigger vorübergehend deaktiviert werden, und natürlich können sie zu einem späteren Zeitpunkt jederzeit wieder aktiviert werden.

Am einfachsten gelingt dies sicherlich mit dem SQL-Befehl ALTER TRIGGER:

```
ALTER TRIGGER INSERT_POSITION DISABLE;

ALTER TRIGGER INSERT_POSITION DISABLE;
```

Aber auch der CL-Befehl CHGPFTRG (Change Physical File Trigger) kann genutzt werden, um den Status des Triggers zu verändern:



Der Befehl CHGPFTRG

7.9.4
Seite 2

Bleibt nur noch die Frage zu klären, wie Sie sich einen Überblick über die Trigger auf Ihrem System verschaffen. Auch hier gibt es wieder verschiedene Alternativen:

a) **Der Datenbankkatalog**
Verschiedene Datenbankkataloge enthalten vielfältige Informationen über Trigger:

☑	SYSTRIGCOL	SYSTRIGCOL	26.01.15 15:22:04
☑	SYSTRIGDEP	SYSTRIGDEP	26.01.15 15:22:03
☑	SYSTRIGGERS	SYSTRIGGER	21.12.15 11:24:03
☑	SYSTRIGUPD	SYSTRIGUPD	26.01.15 15:22:04

Der Datenbankkatalog – Trigger

Dabei ist der Katalog SYSTRIGGERS sicherlich der Katalog, der Ihnen zunächst den besten Überblick über die Trigger Ihres Systems verschafft.

```
67 Select *
68 from qsys2/systriggers
69 where trigger_schema = 'EPSW07';
70
```

TRIGGER_SCHEMA	TRIGGER_NAME	EVENT_MANIPULATION	EVENT_OBJECT_SCHEMA	EVENT_OBJECT_TABLE
EPSW07	INSERT_POSITION	INSERT	EPSW07	AUFTRAGSPOSITION
EPSW07	UPDATEAUFTRAGSWERT	INSERT	EPSW07	AUFTRAGSPOSITION

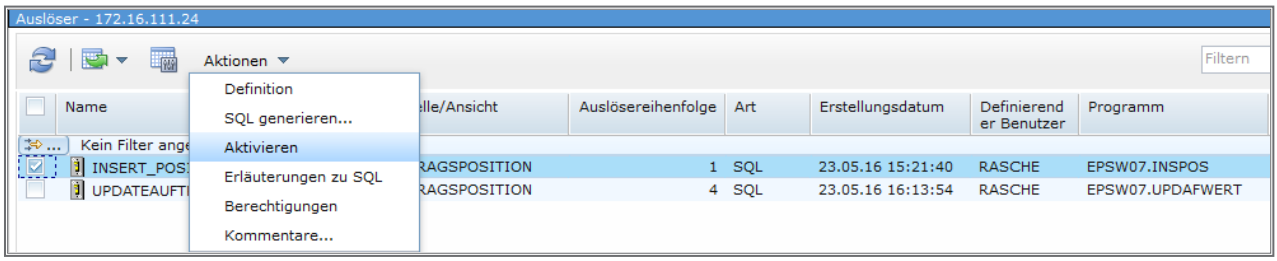
Der Datenbankkatalog SYSTRIGGERS

Die Kataloge SYSTRIGCOL, SYSTRIGDEP und SYSTRIGUPD enthalten hingegen spezifischere Informationen. So finden Sie im Katalog SYSTRIGUPD lediglich UPDATE-Trigger. Und den Katalog SYSTRIGCOL benötigen Sie, wenn Sie beispielsweise wissen möchten, welche Spalten in den verschiedenen Triggern angesprochen werden. Das folgende SQL-Statement zum Beispiel zeigt mir alle Trigger an, die die Spalte AUFTRAGSWERT der Tabelle AUFTRAG verwenden:

```
SELECT TRIGGER_NAME, TRIGGER_SCHEMA
FROM QSYS2/SYSTRIGCOL
WHERE COLUMN_NAME = 'AUFTRAGSWERT'
AND TABLE_NAME = 'AUFTRAG'
AND TABLE_SCHEMA = 'EPSW07';
```

b) Grafische Oberflächen

Auch grafische Oberflächen, wie der System i Navigator oder der Navigator for i, helfen bei der Triggerverwaltung. In diesen Oberflächen finden wir nicht nur viele Informationen zu den Triggern, sondern können zum Beispiel auch einzelne Trigger aktivieren bzw. deaktivieren oder uns die Triggerdefinitionen anzeigen lassen.



Grafische Oberfläche

c) CL-Befehle

Falls Sie lieber CL-Befehle einsetzen möchten, um vorhandene Trigger auf Ihrem System zu identifizieren, so können Sie sich einen Bericht mit dem Befehl PRTTRGPGM (Print Trigger Programs) erstellen:

```
                    Auslöserprogramme drucken (PRTTRGPGM)

Auswahl eingeben und Eingabetaste drücken.

Bibliothek . . . . . > *ALLUSR      Name, *LIBL, *USRLIBL...
Nur Änderungsbericht . . . . .  *NO         *NO, *YES
```

Der Befehl PRTTRGPGM

Mit dem Befehl PRTTRGPGM (Auslöserprogramme drucken) werden die Programme aufgelistet, die als Auslöserprogramme verwendet werden. Der Befehl erstellt zwei Berichte. Der erste Bericht enthält eine Übersicht aller Auslöserprogramme. Der zweite Bericht enthält nur die Auslöserprogramme, die seit der letzten Ausführung des Befehls PRTTRGPGM hinzugekommen sind (Darstellung des Deltas). Wenn der Bericht das erste Mal ausgerufen wird, erhalten Sie selbstverständlich zunächst nur eine Gesamtübersicht. Frühestens mit dem zweiten Aufruf wird der Änderungsbericht erstellt. Falls zwischenzeitlich keine Änderungen stattgefunden haben, wird der Änderungsbericht zwar gedruckt, ist jedoch leer. Beachten Sie bitte auch, dass eine Änderung der Erstellungszeit, des Auslöserereignisses und der Bedingung nicht im Änderungsbericht berücksichtigt werden. Die Datei QSECTRGOLD in Bibliothek QUSRSYS enthält übrigens Informationen über die letzte Ausführung des Befehls PRTTRGPGM. Für jede Bibliothek wird in der Datei ein gleichnamiges Member erstellt.

7.9.4 Seite 4

```
                                        Auslöserprogramm (Gesamtbericht)
5770SS1 V7R2M0  140418
 Angegebene Bibliothek  . . . . :   EPSW07

                                   ---------------------------------------Auslö
 Bibliothek Datei      ASP-Einh.   Name                 Art  Bibliothek Progra
 EPSW07     AUFTR00001 *SYSBAS     INSERT_POSITION      *SQL EPSW07     INSPOS
 EPSW07     AUFTR00001 *SYSBAS     UPDATEAUFTRAGSWERT   *SQL EPSW07     UPDAFW
```

Gesamtbericht PRTTRGPGM

7.9.5 Multi-Event-Trigger (i7.1 TR6)

Ein SQL-Trigger, das wissen Sie bereits, enthält eine oder mehrere SQL-Anweisungen, die ausgeführt werden, wenn die verknüpfte Datenbanktabelle modifiziert wird. Mit Hilfe eines Triggers definieren wir somit automatische Reaktionen auf Ereignisse, die durch Datenmanipulationen auf eine Tabelle ausgelöst werden.

Bislang hatte jeder Trigger einen eigenständigen Aktivierungszeitpunkt (INSERT/UPDATE/DELETE). Doch seit Release i7.1 Technology Refresh 6 kann ein Trigger für mehrere Ereignisse gleichzeitig zuständig sein.

Kommen wir auf unser Beispiel zurück. Eigentlich ist es nicht ausreichend, den Auftragswert in der Tabelle „Auftrag“ zu modifizieren, wenn Zeilen hinzugefügt werden. Auch das Löschen einer Auftragsposition und die Modifizierung des Artikelpreises oder der Positionsmenge wirken sich auf den Auftragswert aus. Es kann also durchaus sein, dass eine Auftragsposition nachträglich in der Menge oder im Preis korrigiert wird.

Wir benötigen somit noch einen UPDATE-Trigger.

```
CREATE or REPLACE TRIGGER UpDateAuftragswertU
        AFTER UPDATE ON AUFTRAGSPOSITION
        REFERENCING NEW NZEILE OLD OZeile NEW_TABLE NTAB Old_Table OTAB
FOR EACH ROW MODE DB2SQL
PROGRAM NAME UpdAFWertU
NOT SECURED
```

Damit der Trigger nur aufgerufen wird, wenn sich der Artikelpreis oder die Positionsmenge verändern, kann ich ihm noch eine Bedingung hinzufügen:

```
WHEN (oZeile.Artikelpreis <> nZeile.Artikelpreis
OR oZeile.Positionsmenge <> nZeile.Positionsmenge)
```

7.9.5
Seite 2

Alternativ zur Bedingung ist es auch möglich, einen Spaltentrigger zu verwenden. Bei einem Spaltentrigger wird der Trigger nicht ausgelöst, wenn die Zeile verändert wird, sondern der Trigger reagiert auf die Veränderung einzelner Spalten . In meinem Beispiel muss der Trigger beispielsweise nur aktiviert werden, wenn sich entweder die Spalte ARTIKELPREIS oder die Spalte POSITIONSMENGE ändert. Die Definition eines Spaltentriggers unterscheidet sich nicht wesentlich von der Definition des oben gezeigten UPDATE-Triggers:

```
CREATE or REPLACE TRIGGER UpDateAuftragswertU
AFTER UPDATE OF POSITIONSMENGE, ARTIKELPREIS ON AUFTRAGSPOSITION
REFERENCING NEW NZEILE OLD OZeile NEW_TABLE NTAB Old_Table OTAB
FOR EACH ROW MODE DB2SQL
PROGRAM NAME UpdAFWertU
NOT SECURED
```

Die nachfolgende Bedingung kann in meinem Beispiel jetzt natürlich entfallen.

```
WHEN (oZeile.Artikelpreis <> nZeile.Artikelpreis
OR oZeile.Positionsmenge  <> nZeile.Positionsmenge)
```

Auch dieser Trigger – als Zeilen- oder Spaltentrigger definiert – wird zuverlässig arbeiten.

Natürlich müssten wir jetzt auch noch berücksichtigen, dass Auftragspositionen storniert werden könnten. Es gibt also noch ein weiteres Szenario, das behandelt werden müsste. Deutlich wird hierbei, dass bislang für unterschiedliche Datenbankevents verschiedene Trigger eingesetzt werden mussten. Viel einfacher ist es allerdings, wenn unterschiedliche Triggerevents in einem Trigger abgebildet werden können. Und genau das passiert in einem Multi-Event-Trigger. Multi-Event-Trigger stehen seit Release i7.1 TR6 (Februar 2013) zur Verfügung.

Multi-Event-Trigger enthalten vordefinierte, logische Variablen, die wir verwenden können, um die verschiedenen Triggerevents zu unterscheiden. Folgende boolesche Variablen stehen in einem Multi-Event-Trigger zur Verfügung: UPDATING/DELETING/INSERTING.

Versuchen wir also, unsere Events in einem einzigen Trigger zu vereinen.

```
CREATE or REPLACE TRIGGER UpDateAuftragswertMulti
AFTER UPDATE OR INSERT OR DELETE ON AUFTRAGSPOSITION

  REFERENCING NEW NZEILE OLD OZeile NEW_TABLE NTAB Old_Table OTAB
  FOR EACH ROW MODE DB2SQL
  PROGRAM NAME UpdAFWertM
  NOT SECURED

SET OPTION DBGVIEW = *SOURCE

BEGIN

IF UPDATING THEN

  IF OZEILE.POSITIONSWERT <> NZEILE.POSITIONSWERT THEN
       UPDATE AUFTRAG SET AUFTRAGSWERT =
              (SELECT SUM ( AUFTRAGSPOSITION . POSITIONSWERT )
              FROM AUFTRAGSPOSITION
              WHERE AUFTRAGSPOSITION.AUFTRAGSNUMMER =
                    AUFTRAG.AUFTRAGSNUMMER )
              WHERE AUFTRAG . AUFTRAGSNUMMER IN
                  ( SELECT AUFTRAGSNUMMER
                    FROM NTAB);
  END IF;

END IF;


IF INSERTING OR DELETING THEN
       UPDATE AUFTRAG SET AUFTRAGSWERT =
              (SELECT SUM ( AUFTRAGSPOSITION . POSITIONSWERT )
              FROM AUFTRAGSPOSITION
              WHERE AUFTRAGSPOSITION.AUFTRAGSNUMMER =
                    AUFTRAG.AUFTRAGSNUMMER )
       WHERE AUFTRAG . AUFTRAGSNUMMER IN
                    ( SELECT AUFTRAGSNUMMER
                      FROM NTAB);
END IF;


END;
```

7.9.5
Seite 4

Mit Hilfe der booleschen Variablen INSERTING, DELETING und UPDATING können wir jetzt die Triggerevents im Triggerbody problemlos unterscheiden und so in Abhängigkeit des Triggerevents unterschiedliche Anweisungen ausführen. In der Verwendung unterliegen die vordefinierten booleschen Variablen keiner Restriktion, das heißt, sie können überall verwendet werden, wo mit Ausdrücken gearbeitet werden kann.

7.9.6 Instead-of-Trigger

Eine Insert-, Update- oder Delete-Anweisung auf eine logische Sicht mit verknüpften Dateien ist nicht zulässig! Diese Behauptung hatte ihre Gültigkeit bis Release V5R4. Bereits unter Release V5R3 über PTF SI16101 wurden INSTEAD-OF-Trigger eingeführt, die jedoch erst mit Release V5R4 ihre volle Funktionalität erhalten haben. Mit Hilfe dieser speziellen Trigger ist es möglich, SQL-Views mit verknüpften Tabellen (JOIN) zu manipulieren. INSTEAD OF-Trigger werden daher nicht für eine Tabelle, sondern immer für eine SQL-View definiert, wodurch eine View, die eigentlich nur READ-ONLY ist, „updatefähig“ wird. Das heißt, wir können SQL-Views verwenden, um in physischen Tabellen Datensätze einzufügen, zu löschen oder zu ändern. Man könnte auch sagen, ein INSTEAD-OF-Trigger übersetzt die Modifikationen, die an der View vorgenommen werden, auf die physischen Dateien. Erfolgt nach der Registrierung eines INSTEAD-OF-Triggers für die View das angegebene Ereignis/Event, also eine Insert-, Update- oder Delete-Operation, so wird der Trigger vom Datenbanken-Manager aufgerufen und ausgeführt. Wird für eine View nur ein INSTEAD-OF-Trigger registriert, etwa für das Ereignis INSERT, und anschließend auf die View eine Update- oder Delete-Operation ausgeführt, so erfolgt ein Abbruch mit der Fehlermeldung SQL0150 (Sicht oder logische Datei XXX in Bibliothek YYY schreibgeschützt). Um solche Probleme zu vermeiden, sollte man immer gleich für jedes der drei Ereignisse einen INSTEAD-OF-Trigger anlegen.

7.9.6
Seite 2

Schauen wir uns auch hierzu ein Beispiel an. Als Erstes definieren wir eine View, die die Daten der Tabellen AUFTRAG und AUFTRAGSPOSITION vereint:

```
CREATE OR REPLACE VIEW AUFTRAG_MIT_POSITIONEN
FOR SYSTEM NAME AUFTRMPOS
(AFAUFTRAGSNUMMER, KUNDENNUMMER, AUFTRAGSDATUM,
AUSLIEFERUNGSDATUM, AUFTRAGSWERT, VERKAEUFER, RABATT, SOLL_AUSLIEFE-
RUNGSDATUM, AFSTATUS, ARTIKELNUMMER, POSITIONSMENGE, POSITIONSWERT,
ARTIKELPREIS, APSTATUS)
AS
(
SELECT
AF.AUFTRAGSNUMMER, KUNDENNUMMER, AUFTRAGSDATUM, AUSLIEFERUNGSDATUM,
AUFTRAGSWERT, VERKAEUFER, RABATT, SOLL_AUSLIEFERUNGSDATUM, AF.STATUS,
ARTIKELNUMMER, POSITIONSMENGE, POSITIONSWERT, ARTIKELPREIS,
AP.STATUS

FROM AUFTRAG AF JOIN AUFTRAGSPOSITION AP
ON AP.AUFTRAGSNUMMER = AF.AUFTRAGSNUMMER
);
```

Nun wollen wir einen INSTEAD-OF-Trigger definieren, der es uns ermöglicht, einen Auftrag über die SQL-View zu erfassen:

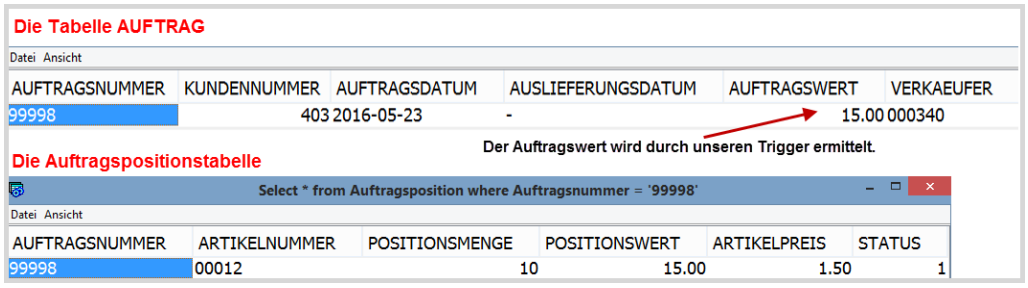
```
CREATE OR REPLACE TRIGGER INSERTAUFTRAG
INSTEAD OF INSERT ON AUFTRAG_MIT_POSITIONEN
REFERENCING NEW NZEILE NEW_TABLE NTAB
FOR EACH ROW MODE DB2SQL

SET OPTION DBGVIEW = *SOURCE

BEGIN
INSERT INTO AUFTRAG
(AUFTRAGSNUMMER, KUNDENNUMMER, AUFTRAGSDATUM,
AUSLIEFERUNGSDATUM, VERKAEUFER, RABATT, SOLL_AUSLIEFERUNGSDATUM, STATUS)

VALUES(NZEILE.AFAUFTRAGSNUMMER, NZEILE.KUNDENNUMMER, CURRENT DATE, NULL,
NZEILE.VERKAEUFER, NZEILE.RABATT, CURRENT DATE + 8 DAYS,
1);

INSERT INTO AUFTRAGSPOSITION
(AUFTRAGSNUMMER, ARTIKELNUMMER,
POSITIONSMENGE, POSITIONSWERT,
ARTIKELPREIS, STATUS)

VALUES(NZEILE.AFAUFTRAGSNUMMER, NZEILE.ARTIKELNUMMER,
NZEILE.POSITIONSMENGE, (NZEILE.POSITIONSMENGE * NZEILE.ARTIKELPREIS),
NZEILE.ARTIKELPREIS, 1);

END;
```

Anschließend können wir die Triggerfunktion sehr einfach testen, indem wir eine neue Zeile einfügen:

```
INSERT INTO AUFTRAG_MIT_POSITIONEN
(AFAUFTRAGSNUMMER, KUNDENNUMMER, VERKAEUFER, RABATT,
ARTIKELNUMMER, POSITIONSMENGE, ARTIKELPREIS)
VALUES('99999', 00990, '000340', 0.00, '00010', 100, 3.80);
```

7.9.6
Seite 4

Nun prüfen wir, ob die Tabellen „Auftrag“ und „Auftragspositionen“ jeweils eine Zeile für die Auftragsnummer 99999 enthalten.



Das Ergebnis des INSTEAD-OF-Triggers

Über das Fehlerhandling müssten wir uns sicherlich noch ein wenig Gedanken machen, aber die Grundfunktion ist einwandfrei umgesetzt.

Fassen wir abschließend die Eigenschaften eines INSTEAD-OF-Triggers zusammen:

Nur SQL-Trigger zulässig:
Der Aktivierungszeitpunkt INSTEAD OF kann nur für SQL-Trigger angegeben werden. Externe Trigger, die in RPG oder Cobol geschrieben wurden, können nicht direkt als INSTEAD-OF-Trigger registriert werden. Es ist jedoch möglich im Trigger eine externe Prozedur aufzurufen.

Keine bedingte Ausführung über WHEN-Klausel:
Die Angabe einer WHEN-Anweisung ist in einem INSTEAD-OF-Trigger nicht zulässig. Das heißt jedoch nicht, dass es nicht möglich wäre, durch entsprechende IF- oder CASE-Anweisungen bedingte Ausführungen zu steuern. Das erfordert lediglich ein wenig mehr Programmieraufwand.

Maximal ein INSTEAD-OF-Trigger kann je Triggerevent definiert werden. So können für eine View maximal drei INSTEAD-OF-Trigger registriert werden.

Ausführungsreihenfolge:
Während BEFORE- und AFTER-Trigger mit einer Tabelle verbunden werden, werden

INSTEAD-OF-Trigger Views zugeordnet. Erst innerhalb des INSTEAD-OF-Trigger-Programms wird dann auf die Zeilen in den Tabellen zugegriffen und erst zu diesem Zeitpunkt können BEFORE- und AFTER-Trigger aktiviert werden. INSTEAD-OF-Trigger werden daher immer vor den BEFORE- und AFTER-Triggern aktiviert.

Satzsperren:
Bei der Verwendung von Views und INSTEAD-OF-Triggern ist es nicht möglich, die Zeilen bereits beim Lesen zu sperren. Erst innerhalb des Triggers, der vom Datenbanken-Manager bei einer Insert-, Update- oder Delete-Operation aktiviert wird, erfolgt der Zugriff auf die eigentlichen Tabellenzeilen, und erst zu diesem Zeitpunkt kann auch die Satzsperre erfolgen.

7.10 Fehlerhandling in Prozeduren / Funktionen / Trigger

Es wird sich wohl kaum ein Unternehmen in der heutigen Zeit finden, dass folgender Aussage widersprechen würde: „Datenqualität ist wichtig!“ Jedes Unternehmen weiß, dass die Datenqualität maßgeblich zu verbesserten Prozessabläufen und Unternehmensergebnissen beiträgt. Aber wie sieht es mit dem praktischen und vor allem konsequenten Handeln auf Basis dieser Erkenntnis aus? Wann und wie haben Sie die Qualität Ihrer Daten im Unternehmen zum letzten Mal geprüft? Von der unternehmerischen Bedeutung der Daten her ist genau dies notwendig, nämlich die Datenqualität regelmäßig zu prüfen, denn jeder im Unternehmen wird behindert, wenn die Daten, mit denen er arbeiten muss, nicht korrekt sind. Und es geht sogar noch weiter: Verschiedene Studien belegen, dass schlechte Datenqualität Unternehmen mehrere Millionen Euro pro Jahr kosten. Ein europäisches Unternehmen stellte bei einer Datenqualitätsanalyse fest, dass 4 Prozent der angefallenen Rechnungen erst gar nicht ausgestellt wurden. Bei einem Umsatz von 2 Mrd. Euro bedeutet dies 80 Mio. Euro unbezahlter Rechnungen.

Natürlich kümmern wir uns indirekt um die Qualität unserer Daten, da wir stets für den schnellen und reibungslosen Betrieb unserer IT-Systeme sorgen. Schließlich sind instabile Systeme Gift für die Datenqualität. Aber stellen wir dabei die Qualität unserer Daten in den Mittelpunkt, oder geht es eher um einen reibungslosen 7x24-Stundenprozess und eine entsprechend hohe Produktivität? Die Qualität der Daten wird nicht gezielt gesteuert, sondern ist oftmals ein Nebenprodukt oder noch schlimmer, wird häufig als gegeben vorausgesetzt. Diese Haltung wäre vielleicht ausreichend, wenn hohe Datenqualität „nice to have“ wäre. Dem ist aber nicht so! Eine hohe Qualität der Daten ist für fast alle Unternehmen in der heutigen Zeit ein absolutes „Muss“.

Diese Herausforderung wird in Zukunft nicht kleiner, da sich laut entsprechender Studien die Datenvolumen alle zwei Jahre verdoppeln werden. Schon längst sprechen wir von Exa-, Zetta- oder gar Yottabytes und nicht mehr von Giga- und Terabytes. Ein Zettabyte ist übrigens eine 1 mit 21 Nullen. Eine für mich unvorstellbar große Zahl.

Angesichts dieser Größenordnung, dem rasanten Wachstum und der immensen Bedeutung unserer Unternehmensdaten, reicht es heute nicht mehr aus, Daten einfach nur zur Verfügung zu stellen.

Wie reagieren wir auf diese Herausforderungen? Betrachten wir doch einmal kurz die Vergangenheit der AS/400. Vor 25 Jahren war die immense Bedeutung der Daten in ihrer heutigen Ausprägung noch nicht vorhersehbar. Der Schwerpunkt der IT war ausgerichtet auf ausfallsichere Systeme und hier hat sich die AS/400 ohne jeden Zweifel bewährt. Gleichzeitig ging und geht es um den Datenschutz. Auch in diesem Punkt wurde nachgelegt. Vielleicht nicht immer mit Bordmitteln des Servers, aber fast jedes Unternehmen hat entsprechende Maßnahmen eingeleitet, um seine Daten zu schützen. Doch Datenschutz alleine macht noch keine Datenqualität aus. Stellen wir doch mal die Frage nach der Datenintegrität.

Das Wort Integrität stammt aus dem Lateinischen und bedeutet so viel wie „Unversehrtheit“ oder „Reinheit“. Die Datenintegrität gehört neben der Verfügbarkeit und Vertraulichkeit von Daten zu den drei klassischen Zielen der Informationssicherheit. Die Datenintegrität kann und sollte heute durch referenzielle Integritätsbedingungen sichergestellt werden. Jede Datenbank verfügt über diese Mechanismen – so auch die DB2 for i. Haben Sie Integritätsbedingungen in Ihrer Datenbank implementiert? Und wenn nicht, warum verzichten Sie auf dieses Instrument der Datensicherheit? Häufig liegt die Antwort in der Historie begründet:

- Zum einen setzt die Implementierung der referenziellen Integritätsbedingungen (vgl. Kapitel 7.2.4.1) eine konsistente Datenbasis voraus. Man kann auf diesem Weg sehr schnell feststellen, wie „gut“ die über Jahrzehnte gewachsene Datenbasis wirklich ist. Und falls Sie feststellen, dass es Inkonsistenzen gibt, dann sollten Sie nicht nach dem Prinzip „Was ich nicht weiß, macht mich nicht heiß“ agieren. Angesichts der enormen Bedeutung Ihrer Daten, sollten Sie zumindest versuchen die Fehler zu beheben.
- Ein anderer Grund liegt meist in der Programmierung. Wenn referenzielle Integritätsbedingungen aktiviert werden, müssen die Programme die Regeln entsprechend berücksichtigen. Der Kopfsatz kann dann halt nicht vor dem Positionssatz gelöscht werden.

Mir ist bewusst, dass die vielen tausend Zeilen Quellcode nicht alle überarbeitet und umgeschrieben werden können, aber wer oder was hindert Sie daran, referenzielle Integritätsbeziehungen für neue Tabellen zu nutzen?

Bleiben wir noch einen Augenblick in der Vergangenheit und werfen einen weiteren Blick auf die Programmierung. Aufgrund der hohen Stabilität der AS/400 haben viele Programmierer auf ein ausgeprägtes Fehlerhandling und Transaktionssicherheit in ihren HLL-Programmen verzichtet. Vielleicht kann man in einem HLL-Programm damit leben, denn der Anwender wird sich sicherlich irgendwann melden, wenn das Programm abstürzt oder die Liste nicht gedruckt wird. Was passiert aber, wenn Sie Ihre Business-Logik von der Anwendungsschicht trennen? Ein SQL-Statement, das eine Fehlersituation auslöst, bringt die Client-Applikation nämlich nicht zum Absturz! Und was passiert, wenn in einer SQL-Prozedur drei aufeinanderfolgende und voneinander abhängige UPDATES laufen und nur das dritte Statement läuft auf einen Fehler? Wie wollen Sie die zuvor gemachten Änderungen rückgängig machen, um letztendlich die Datenkonsistenz Ihrer Datenbank zu gewährleisten? Fehlerhandling und Transaktionssicherheit sind zwei Schlagworte, die in neuen modernen Applikationen zunehmend an Bedeutung gewinnen, wenn Sie über den Stellenwert unserer Daten in der heutigen Zeit nachdenken!

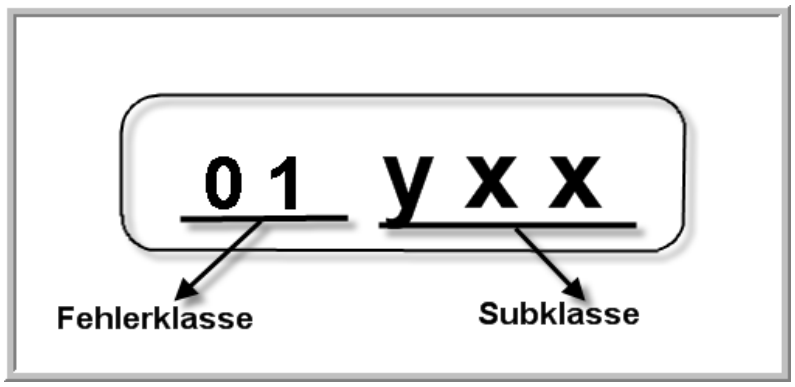
7.10
Seite 4

7.10.1 Fehlermeldungen durch die Datenbank

Das DBMS nutzt im SQL-Umfeld zwei Felder, um Fehler und Warnungen zu melden. Der SQLCODE als Integer-Wert ist die ursprüngliche Variable, die von der DB2 genutzt wurde. Mit dem SQL-Standard99 wurden die SQL-Errorcodes jedoch für alle Datenbankhersteller standardisiert: Diese standardisierten Fehlerbedingungen werden durch die Variable SQLSTATE CHAR(5) abgebildet. SQLSTATE basiert heute auf dem ISO/ANSI SQL 1999 CORE-Standard als plattformunabhängige Fehlercodestruktur. Für die DB2 for i gibt es ein eigenes Handbuch, dass alle SQL-Fehlercodes beinhaltet (IBM i Database SQL messages and codes).

Der SQLCODE ist ein Returncode, der vom DBMS nach der Ausführung jedes SQL-Statements gesendet wird. Jeder SQLCODE hat eine korrespondierende Fehlermeldung in der Nachrichtendatei QSQLMSG. Die Message-ID in QSQLMSG besteht in der Regel aus dem SQLCODE selbst sowie dem Präfix SQL. Ist der SQLCODE 551, heißt die zugehörige MSGID SQL0551. Ist der SQLCODE negativ – also zum Beispiel -0204 – bleibt die Message-ID natürlich trotzdem positiv und heißt SQL0204. Eine Ausnahme gibt es jedoch: Der SQLCODE kann fünfstellig sein. In diesen Fällen wird das Präfix auf SQ verkürzt. Aus dem SQLCODE 30070 wird also die Nachrichten-ID SQ30070.

Der SQL-Status ist ebenfalls ein Returnwert, der nach der Ausführung eines SQL-Statements gesendet wird. Allerdings ist der Statuscode ein fünf Byte großes Zeichenfeld (CHAR(5)). Wobei die ersten beiden Bytes die Fehlerklasse repräsentieren und die nachfolgenden drei Bytes die Subklasse darstellen. Die Klasse 01 repräsentiert alle Warnungen, während die Klasse 02 anzeigt, dass keine Daten gefunden wurden. Alle anderen Klassen sind Fehlerklassen.



Aufbau SQLSTATUS

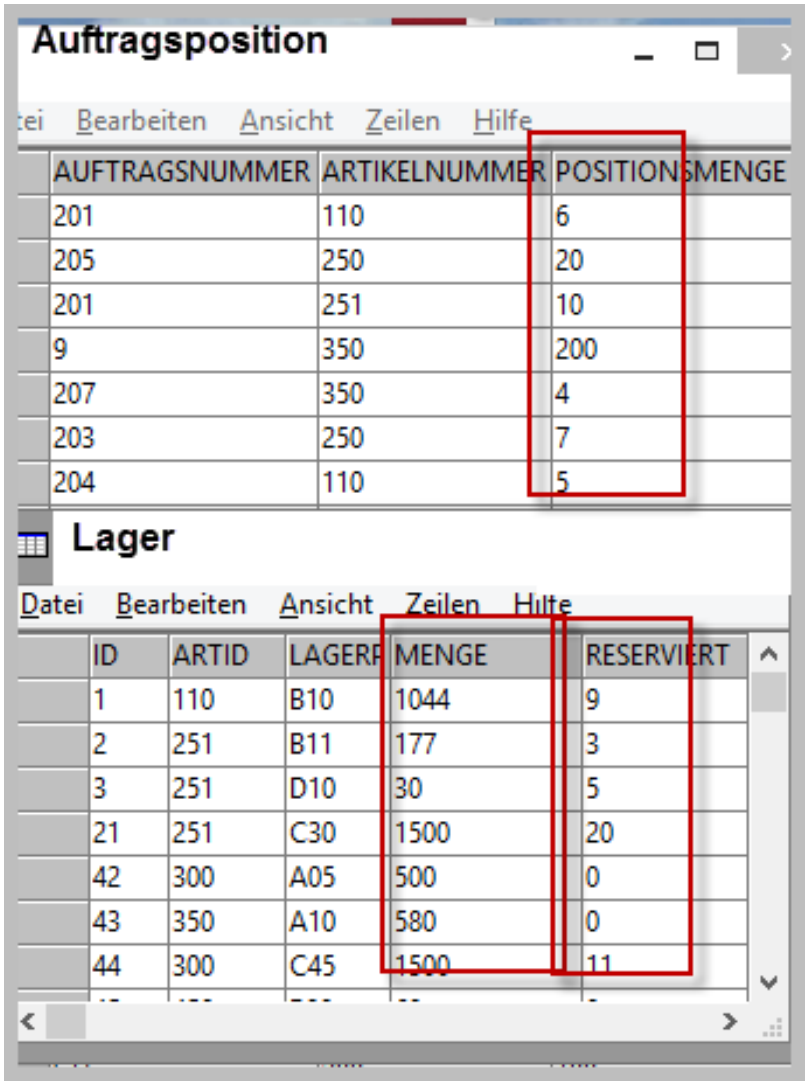
Falls Sie sich entscheiden mit dem SQL-Status zu arbeiten, können Sie auch eigene Fehlerklassen definieren. Dafür stehen eigens bestimmte Wertebereiche zur Verfügung. Dazu aber später mehr.

Um den SQL-STATUS zu verwenden beziehungsweise auszuwerten, stehen verschiedene Möglichkeiten zur Verfügung. In einem Java-Programm kann beispielsweise die Methode getSQLState() verwendet werden, im HLL-Programm, aber auch in den SQL-Prozeduren ist es ausreichend, eine entsprechende Variable mit dem Namen zu deklarieren. Auch die SQL-Anweisung GET DIAGNOSTICS kann genutzt werden, um den SQL-Status zu ermitteln.

Jeder SQLSTATUS korrespondiert mit einem oder mehreren SQLCODES. Auch ist es möglich, dass ein SQLSTATUS für verschiedene SQLCODES verwendet wird. Grundsätzlich lässt sich aber sagen, dass der SQLSTATUS analog zum SQLCODE den Wert 00000 beziehungsweise 0 enthält, wenn das Statement korrekt ausgeführt wurde. Falls Sie einen SQLCODE größer 0 erhalten, wurde das Statement mit Warnungen ausgeführt. Dies entspricht der Fehlerklasse 01 beziehungsweise 02 des SQL-Statusfelds; Wobei die Klasse 02 eine Sonderstellung einnimmt und über fehlende Daten informiert. Eine Warnung erhalten Sie zum Beispiel, weil Daten gekürzt wurden, die Zielvariable zu kurz für die Daten ist. Eine Warnung wird aber auch ausgegeben, wenn Sie ein DELETE-Statement ohne WHERE-Klausel absetzen.

Schauen wir uns jetzt eine einfache Prozedur an, die mit dem SQL-Status arbeitet.

Zuvor müssen wir die Ausgangsbasis betrachten: Die nachfolgend gezeigte Prozedur arbeitet mit zwei Tabellen: Auftragsposition und Lager. Die Prozedur soll die Positionsmengen in der Auftragspositionstabelle ändern und analog dieser Veränderungen die Lagerbestände in der Tabelle „Lager“ korrigieren. Dabei müssen sowohl der reservierte als auch der verfügbare Bestand berücksichtigt werden.



Die Beispieltabellen

Die Prozedur ist schnell geschrieben, wobei wir zunächst auf eine komplizierte Business-Logik verzichten wollen.

7.10.1
Seite 4

```
CREATE or REPLACE PROCEDURE ChangeAuftragsMenge
(
IN I_Auftragsnummer smallInt default 0,
IN I_Artikelnummer char(5) default ' ',
IN I_Menge dec(5,0) default 0,
OUT MSG CHAR(50)
)
LANGUAGE SQL
SPECIFIC CHGAUFMG
MODIFIES SQL DATA
PROGRAM TYPE SUB
NOT DETERMINISTIC
DYNAMIC RESULT SETS 0
CALLED ON NULL INPUT

        SET OPTION
        COMMIT = *NONE ,
        DBGVIEW = *SOURCE ,
        DATFMT = *EUR,
        TIMFMT = *EUR

BEGIN
DECLARE SQLSTATE CHAR(5);

UPDATE Auftragsposition
SET Positionsmenge = Positionsmenge + I_Menge,
Positionswert = (Positionsmenge + i_menge) * Artikelpreis
WHERE Auftragsnummer = I_Auftragsnummer
AND Artikelnummer = i_artikelnummer;

IF SQLSTATE = '00000' THEN

        CASE
-- Menge wird erhöht
        WHEN i_Menge > 0 THEN
        UPDATE Lager set reserviert = (reserviert + i_menge),
         menge = (menge - i_menge)
        WHERE artid = i_Artikelnummer
        AND Lagerplatznummer = (Select Lagerplatznummer
                    From Lager
                    where artid = I_artikelnummer
                    and menge > i_menge
                    fetch first 1 row only);
                    IF SQLSTATE <> '00000' Then
                                SET MSG = SQLSTATE || ': Fehler bei Update Lager (Plus Auftragsmenge) ' ||
I_Artikelnummer;
                    END IF;
-- Menge wird reduziert
        WHEN i_Menge < 0 THEN
                    UPDATE     Lager set reserviert = (reserviert + i_menge),
                    menge = (menge - i_menge)
                    WHERE artid = i_Artikelnummer
                    AND        Lagerplatznummer = (SELECT Lagerplatznummer
                                FROM Lager
                                WHERE artid = I_artikelnummer
                                AND reserviert > (i_menge * -1)
                                FETCH FIRST 1 ROW ONLY);
                    IF SQLSTATE <> '00000' Then
                                SET MSG = SQLSTATE || ': Fehler bei Update Lager (Minus Auftragsmenge) ' ||
I_Artikelnummer;
                    END IF;
        END CASE;
ELSE
        SET MSG = SQLSTATE || ,: Fehler bei Update Position , || I_Auftragsnummer || ';' || i_Artikelnummer;
END IF;
```

Jetzt wollen wir unsere Prozedur testen:

```
CALL ChangeAuftragsMenge (111, '110', 5, ?);
```

Als Ergebnis erhalten wir folgende Ausgabe:

```
Ausgabeparameter 4 = 00000: Fehler bei Update Position 111;110
```

Einerseits ist das Ergebnis korrekt, da die Auftragsnummer 111 nicht existiert. Andererseits überrascht der SQL-Status, der den Wert 00000 enthält. Wie kommt das? Sie haben gelesen, dass jedes SQL-Statement den SQLSTATUS verändert. Dabei übersieht man schnell, dass auch eine IF-Anweisung den SQL-STATUS verändert. Wenn wir also – wie in der Prozedur geplant – den SQL-STATUS weiterhin verwenden wollen, müssen wir ihn nach der Update-Anweisung in einer Variablen sichern.

```
UPDATE Auftragsposition
SET Positionsmenge = Positionsmenge + I_Menge,
    Positionswert = (Positionsmenge + i_menge) * Artikelpreis
WHERE Auftragsnummer = I_Auftragsnummer
AND Artikelnummer = i_artikelnummer;
SET #Status = SQLSTATE;
```

Die Variable müssen Sie natürlich deklarieren und in Ihrer Ausgabe verwenden, dann erhalten Sie auch das gewünschte Ergebnis:

```
Ausgabeparameter 4 = 02000: Fehler bei Update Position 111;110
```

7.10.1
Seite 6

7.10.2 Die Anweisung GET DIAGNOSTICS

Nun wollen wir einen Schritt weitergehen und nicht nur den Fehlercode, sondern auch die Originalfehlermeldung in der Output-Variablen verarbeiten. Wie können wir uns diese Information beschaffen?

Die Datenbank verwendet eine sogenannte Diagnostic Area (DA), um Informationen über das zuletzt ausgeführte SQL-Statement zu speichern. Diese DA wird vor der Ausführung eines SQL-Statements stets bereinigt, so dass dieser Bereich die Informationen über das zuletzt ausgeführte SQL-Statement beinhaltet. Eine Ausnahme hiervon stellt die Anweisung GET DIAGNOSTICS dar. Diese Anweisung verändert weder den SQLSTATUS noch die DA. Mit Hilfe der Anweisung GET DIAGNOSTICS sind Sie in der Lage Informationen der DA zu lesen.

Tritt zur Laufzeit unseres SQL-Statements ein Fehler auf, löst die Datenbank eine Exception oder Warnung aus, die wir problemlos über Variablen SQLCODE beziehungsweise SQLSTATE auswerten können. Doch wie gelangen wir an die Nachrichtentexte. Oder wie reagieren wir, wenn keine Datenbank-Exception ausgelöst wird, weil das SQL-Statement fehlerfrei ausgeführt wurde, wir aber dennoch Information über unser SQL-Statement benötigen? Der SQLSTATUS ‚02000' greift bei Cursormengen nicht! Wie erfahren wir, ob unser Cursor Daten enthält? Oder wie ermitteln wir, wie viele Zeilen bei einem Update verändert wurden? Die Antwort lautet häufig: GET DIAGNOSTICS.

Die SQL-Anweisung GET DIAGNOSTICS erlaubt das Abrufen unterschiedlicher Informationen. Dabei beziehen sich die Informationen stets auf das zuletzt ausgeführte SQL-Statement. Das Problem mit der Anweisung GET DIAGNOSTICS ist ihr Umfang. Ganze 26 Seiten umfasst die Erläuterung im IBM Referenzhandbuch „DB2 for i SQL Reference i7.1". Hinzu kommen unterschiedliche Formate und zu guter Letzt die kleinen Tücken, die nicht im Referenzhandbuch erwähnt werden und oftmals dazu führen, dass in Foren Bugs vermutet werden, wo keine Fehler sind. Schauen wir uns also die Anweisung GET DIAGNOSTICS etwas genauer an!

Mit der Anweisung GET DIAGNOSTICS können Sie

- Informationen zu Ihrem SQL-Statement abrufen (Statement-Informationen)
- Informationen über Ihre aktuelle Datenbankverbindung ermitteln (Connection-Informationen)
- SQL-Fehlersituationen analysieren (Condition-Informationen)

Ergänzung 32 / 2015

7.10.2
Seite 2

Je nachdem, welche Informationen benötigt werden, stehen zwei unterschiedliche Formate zur Verfügung:

Syntax: Statement-Informationen

```
GET CURRENT / STACKED DIAGNOSTICS
variable1 = statement-information, variable2 = statement-information, ...
```

Syntax: Condition-Informationen / Connection-Informationen

```
GET CURRENT / STACKED DIAGNOSTICS CONDITION 1
variable1 = condition-information, variable2 = condition-information, ...
```

Zunächst müssen Sie festlegen, welche Diagnostic Area Sie lesen wollen. GET CURRENT DIAGNOSTICS zeigt stets auf die DA des unmittelbar vorausgegangenen SQL-Statements. GET STACK DIAGNOSICS darf nur in einem Handler – einer speziellen „Fehlerroutine“ in SQL-Prozeduren, Funktionen oder Triggern – verwendet werden. Dazu später mehr. Mit der Anweisung GET STACK DIAGNOSTICS lesen Sie die DA der letzten außerhalb des Handlers ausgeführten SQL-Anweisung. Ist die GET DIAGNOSTICS-Anweisung die erste im Handler, so werden Sie zwischen den beiden DAs keine Unterschiede feststellen.

Als Nächstes müssen Sie überlegen, welche Informationen benötigt werden. Brauchen sie Statement-Informationen, folgen jetzt die Variablen und die Statement-Information-Items. Sie können mit einer GET DIAGNOSTICS-Anweisung mehrere Informationen abrufen.

Etwas anders ist die Syntax, wenn Sie Condition- oder Connection-Informationen benötigen. In diesem Fall nutzen Sie das Schlüsselwort CONDITION gefolgt von einem Integer-Wert. In der Regel wird dies der Wert 1 sein.

```
GET CURRENT DIAGNOSTICS CONDITION 1
variable1 = condition-information, variable2 = condition-information, ...
```

Welche Informationen stehen zur Verfügung?
Über die **Command_Function** können Sie beispielsweise ermitteln, welche SQL-Anweisung soeben ausgeführt wurde. Dies kann sinnvoll sein, wenn Sie Fehlersituationen protokollieren wollen, um zu analysieren, welche Anweisung den Fehler erzeugt hat. Alternativ steht hierfür die Information **Command_Function_Code** zur Verfügung. Eine dynamisch aufgebaute Fetch-Anweisung wird beispielsweise durch den Code 39 repräsentiert, während statische Fetch-Anweisungen den Code 45 liefern. Muss tatsächlich ermittelt werden, welches Statement zuletzt ausgeführt wurde, sind die Codes natürlich sehr viel einfacher zu handhaben als die entsprechenden Befehlsnamen.

Auch die Information **DB2_LAST_ROW** kann hilfreich sein, wenn Sie beispielsweise mit einem Multi-Fetch-Befehl Daten sukzessive selektieren und feststellen müssen, ob das Dateiende erreicht wurde. DB2_LAST_ROW ist 0, solange weitere Daten vorhanden sind. Aber Vorsicht! Die Information wird nur bei einem Multi-Fetch-Select-Statement gesetzt. Und genau hier liegt die Schwierigkeit. GET DIAGNOSTICS kann sehr nützliche Informationen liefern, kennt aber auch eine Menge Ausnahmen und spezifische Einschränkungen für die diversen Items. Sie werden daher nicht umhin kommen, sich sehr genau die Einschränkungen im Referenzhandbuch anzuschauen!

Kommen wir nun auf unser Ausgangsproblem zurück. Wir benötigen den Originalnachrichtentext. Es handelt sich hierbei um die Condition-Information MESSAGE_TEXT, und auch den SQLSTATUS kann die Anweisung uns zur Verfügung stellen, wenn wir das Item RETURNED_SQLSTATE verwenden.

```
UPDATE Auftragsposition
SET Positionsmenge = Positionsmenge + I_Menge,
Positionswert = (Positionsmenge + i_menge) * Artikelpreis
WHERE Auftragsnummer = I_Auftragsnummer
AND Artikelnummer = i_artikelnummer;

GET DIAGNOSTICS CONDITION 1 #status = RETURNED_SQLSTATE, msg = MESSAGE_TEXT;
:
:
ELSE
      SET MSG = #Status || ' ' || Trim(MSG) || ':' || I_Auftragsnummer || ';' || I_Artikelnummer;
END IF;
```

7.10.2
Seite 4

7.10.3 Globale und lokale Fehlerroutinen

Etwas störend an der gezeigten Lösung ist sicherlich, dass Sie die GET DIAGNOSTICS-Anweisung hinter jedem SQL-Statement codieren müssen. Keine sonderlich gute Lösung. Innerhalb der SQL-Prozeduren und Funktionen können wir stattdessen mit Handlern arbeiten – vergleichbar mit einer globalen MONMSG-Anweisung in einem CL-Programm.

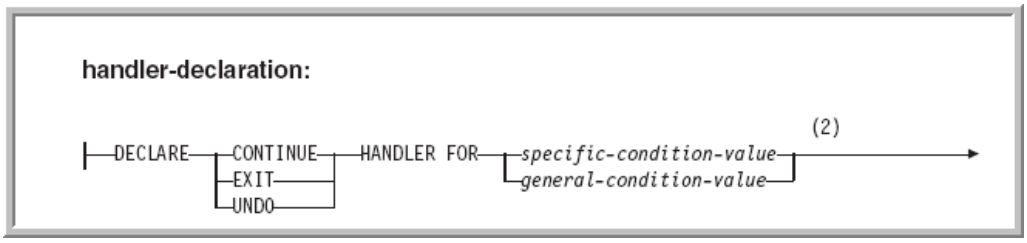
Grundsätzlich muss man zunächst einmal zwischen drei Handler-Typen unterscheiden:

CONTINUE-Handler
Bei einem CONTINUE-Handler wird die Fehlerroutine ausgeführt und anschließend die Prozedur fortgesetzt. Die Steuerung wird an das SQL-Statement übergeben, das dem Statement folgt, welches den Fehler ausgelöst hat. Sollte der Fehler durch eine IF- oder Schleifenbedingung ausgelöst worden sein, wird die Anweisung nach dem zugehörigen END IF, END CASE, END WHILE oder END REPEAT fortgesetzt.

EXIT-Handler
Bei einem EXIT-Handler wird die Prozedur nach erfolgreicher Ausführung der Fehlerroutine beendet.

UNDO-Handler
Auch bei einem UNDO-Handler wird zunächst die Fehlerroutine durchlaufen, anschließend wird die Prozedur beendet. Allerdings wird beim Beenden der Prozedur ein implizites Rollback ausgeführt. Daher muss bei einem UNDO-Handler auch BEGIN ATOMIC verwendet werden.

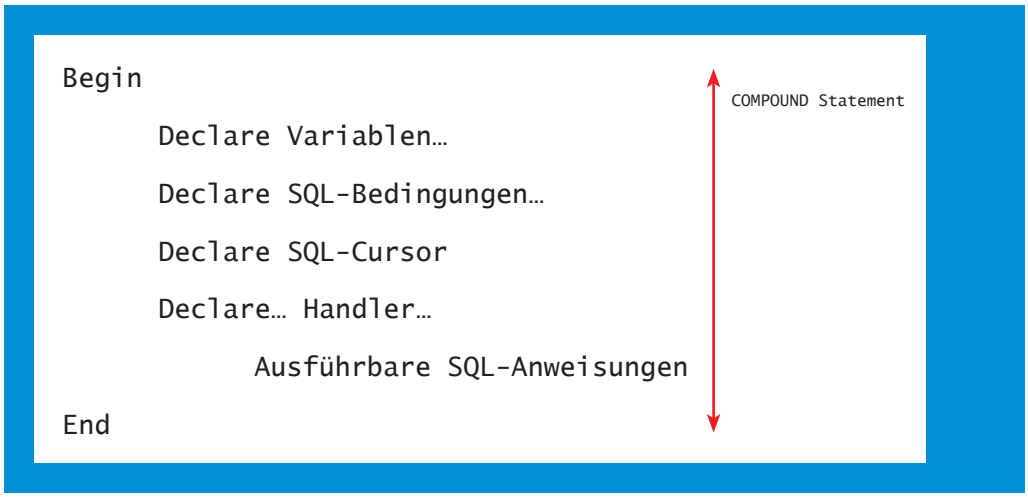
Die Deklaration der Handler ist nicht schwierig:



Handler-Deklaration

7.10.3
Seite 2

Sie müssen lediglich auf die Reihenfolge Ihrer Deklarationen achten:



Es ist durchaus erlaubt, mehrere Compound-Statements zu verschachteln. Dadurch haben Sie die Möglichkeit, lokale Variablen und lokale Handler zu definieren.

```
Begin
      Declare…
            Begin
                  Declare…

            End
End
```

Für unser Beispiel ist es sinnvoll, die Verarbeitung zu beenden, falls das UPDATE fehlschlägt. Also werden wir einen EXIT-Handler verwenden müssen. Als Nächstes müssen wir dem Handler mitteilen, für welche SQL-Stati er verantwortlich ist. Dabei können wir die SQL-Stati direkt verwenden oder aber SQL-Bedingungen. Drei SQL-Bedingungen sind bereits definiert und stehen uns unmittelbar zur Verfügung. Schauen Sie sich die nachfolgenden Definitionen an. Sie werden sehr schnell verstehen, was gemeint ist:

```
DECLARE CONTINUE HANDLER FOR     SQLSTATE '23505',
                      SQLSTATE '23510',
                      SQLSTATE '23511'
                Begin
                …
                End;


DECLARE EXIT HANDLER FOR NOT FOUND
                Begin
                ….
                End;


DECLARE … HANDLER FOR SQLEXCEPTION, SQLWARNING
                Begin
                …
                End;
```

Die Bedingung NOT FOUND steht für die Fehlerklasse 02. Alle Warnungen werden in der Fehlerklasse 01 erfasst, und alle anderen Klassen werden als SQL-Exception verstanden. Wir können aber auch eigene SQL-Bedingungen definieren, falls dies erforderlich wird:

```
DECLARE   check_constraint_error  CONDITION FOR      '23513';

DECLARE Exit Handler FOR check_constraint_error
```

7.10.3
Seite 4

Nachdem wir uns jetzt mit der Syntax vertraut gemacht haben, wollen wir die Handler in unsere Prozedur integrieren. Ich möchte zunächst, falls ein Satz für das UPDATE nicht gefunden wird, lediglich eine Nachricht an den Client zurückgeben. Die Umsetzung ist nicht schwer:

```
BEGIN

DECLARE  SQLSTATE CHAR(5);
DECLARE  #STATUS CHAR(5);

DECLARE EXIT HANDLER FOR NOT FOUND
          Begin
             GET DIAGNOSTICS CONDITION 1 #status = RETURNED_SQLSTATE, MSG = MESSAGE_TEXT;
             SET MSG = #Status || ' ' || TRIM(MSG) || ':' || I_Auftragsnummer || ';' || I_Artikelnummer;
          End;


update Auftragsposition
SET Positionsmenge = Positionsmenge + I_Menge, Positionswert = (Positionsmenge + i_menge) * Artikelpreis
WHERE Auftragsnummer = I_Auftragsnummer AND Artikelnummer = i_artikelnummer;

CASE
-- Menge wird erhöht
WHEN i_Menge > 0 THEN
         UPDATE Lager SET reserviert = (reserviert + i_menge), menge = (menge - i_menge)
         WHERE artid = i_Artikelnummer AND Lagerplatznummer =
                                                  (SELECT Lagerplatznummer
                                                  FROM Lager
                                                  WHERE artid = I_artikelnummer
                                                  AND menge > i_menge
                                                  FETCH FIRST 1 ROW ONLY
                                                  );

         -- Menge wird reduziert
WHEN i_Menge < 0 THEN
         UPDATE Lager SET reserviert = (reserviert + i_menge), menge = (menge - i_menge)
WHERE artid = i_Artikelnummer AND Lagerplatznummer =
                                             (SELECT Lagerplatznummer
                                             FROM Lager
                                             WHERE artid = I_artikelnummer
                                             AND reserviert > (i_menge * -1)
                                             FETCH FIRST 1 ROW ONLY);

END CASE;
END;
```

Sie können schnell erkennen, dass die Prozedur kompakter wird. Ich muss nicht mehr nach jedem SQL-Statement den SQL-Status abfangen, und ich muss mir den SQLSTATUS auch nicht in einer Variablen grundsätzlich sichern. Der EXIT HANDLER übernimmt jetzt global diese Aufgabe.

Bleibt nur noch folgende Frage zu klären: Wie reagiere ich auf allgemeine Fehlersituationen? Dafür integrieren wir jetzt in einem zweiten Schritt einen weiteren Handler, der sowohl für die Ausnahmesituationen als auch für Warnungen verantwortlich ist. Dieser Handler soll alle Warnungen und Ausnahmesituationen in einer Tabelle protokollieren. Die Tabelle LOG wurde dafür folgendermaßen definiert:

```
CREATE TABLE LOG
(
DATUM_ZEIT FOR COLUMN DATZEIT TIMESTAMP
GENERATED ALWAYS FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP NOT NULL ,

        BENUTZER CHAR(18 ) NOT NULL DEFAULT USER ,

        PROZEDURNAME FOR COLUMN PROCNAME CHAR(128) NOT NULL DEFAULT '' ,
        TABELLE CHAR(128) DEFAULT NULL ,
        ANWEISUNG CHAR(100) NOT NULL DEFAULT '' ,
        MSGTEXT CHAR(1000) NOT NULL DEFAULT '' ,
        STATUS CHAR(5 DEFAULT NULL ,
        ALLEXCEPTION FOR COLUMN ALLEXC VARCHAR(10000) ALLOCATE(1000) DEFAULT NULL
)

        RCDFMT LOGFMT ;
```

7.10.3
Seite 6

Der Zeitstempel und der Benutzername müssen nicht eingefügt werden. Diese Spalten werden automatisch vom DBMS befüllt. Alle anderen Spalten übernimmt die Prozedur beziehungsweise mein Handler:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
        BEGIN
            GET DIAGNOSTICS #Anweisung = COMMAND_FUNCTION;
            GET DIAGNOSTICS       Condition 1
                           MSG = Message_text,
                           #status = RETURNED_SQLSTATE,
                           #tabName = TABLE_NAME;
            GET DIAGNOSTICS #Info = All ;
            INSERT INTO Log (Prozedurname, MSGTEXT, Anweisung, Status, Tabelle, Allexception )
            VALUES(#Procname, MSG, #Anweisung, #Status, #Tabname, #Info );

        End;
```

Mit Hilfe der ersten GET DIAGNOSTICS-Anweisung ermittle ich das Statement, welches den Fehler verursacht hat. Die nachfolgende GET DIAGNOSTICS-Anweisung ist für den Nachrichtentext, den SQLSTATUS und den Tabellennamen zuständig. Als Drittes sehen Sie eine Sonderform der Anweisung GET DIAGNOSTICS. Hier handelt es sich um eine kombinierte Information oder anders gesagt: Diese Anweisung liefert alle zur Verfügung stehenden Diagnoseinformationen für das fehlerhafte Statement. Mit Hilfe der Protokolldatei gehen mir keine Informationen verloren und ich bin jederzeit in der Lage, nachzuvollziehen, welche Fehler in den verschiedenen Prozeduren zum Abbruch geführt haben. Im günstigsten Fall bleibt die LOG-Datei leer und ich muss keine Fehlersituationen in den Prozeduren ermitteln.

Abschließend wollen wir noch einen weiteren Schritt gehen: Falls die Zeile für das UPDATE in meiner Lagertabelle nicht gefunden wird, möchte ich die Verarbeitung nicht sofort abrechen, sondern die Artikelnummer soll in einer Dispositiontabelle erfasst werden. Wir brauchen also zwei unterschiedliche Verfahren – bei gleichem Status. Auch das ist möglich! Wir müssen lediglich unsere Compound-Statements verschachteln:

```
GLOBAL:BEGIN
DECLARE ….
-------------------------------------------------------------------------------------------
LOKAL:BEGIN
        DECLARE EXIT HANDLER FOR NOT FOUND
        BEGIN
           SET MSG =      'ARTIKEL: ' || I_ARTIKELNUMMER ||
                   ' IM LAGER NICHT VERFÜGBAR! ZUR DISPOSITION VERMERKT!';
           INSERT INTO DISPOSITION(ARTID, MENGE) VALUES(I_Artikelnummer, I_Menge);
        END;

CASE

-- Menge wird erhöht
WHEN i_Menge > 0 THEN
update Lager set reserviert = (reserviert + i_menge), menge = (menge – i_menge)
where artid = i_Artikelnummer
        and Lagerplatznummer = (Select Lagerplatznummer
           From Lager
           where artid = I_artikelnummer
           and menge > i_menge
           fetch first 1 row only);

        -- Menge wird reduziert
WHEN i_Menge < 0 THEN
update  Lager set reserviert = (reserviert + i_menge),
           menge = (menge – i_menge)
           where artid = i_Artikelnummer
           and Lagerplatznummer = (Select Lagerplatznummer
              From Lager
              where artid = I_artikelnummer
              and reserviert > (i_menge * -1)
              fetch first 1 row only);

END CASE;
END;
-------------------------------------------------------------------------------------------
END;
```

7.10.3
Seite 8

7.10.4 Benutzerdefinierte Meldungen

In den vorangegangenen Kapiteln haben Sie verschiedene Lösungsansätze kennengelernt, um Fehlermeldungen der Datenbank zu behandeln. Aber Sie werden sicherlich auch eigene Geschäftsregeln in Ihre Applikationen implementieren, deren Einhaltung Sie überwachen wollen. Zum Beispiel darf ein Preisnachlass nicht dazu führen, dass der Artikel kostenlos abgegeben wird, und Sie wollen einen Fehler ausgegeben bekommen, falls der Bestand durch einen Auftrag Null wird, oder Sie möchten bei der Erfassung eines Kunden eine Fehlermeldung angezeigt bekommen, wenn der Kunde nicht volljährig ist. Da die Szenarien dieser Beispiele für die Datenbank keine Fehler darstellen, wird das DBMS auch nicht reagieren. Sie könnten diese Meldungen natürlich in die Client-Applikationen integrieren, aber es besteht auch die Möglichkeit, diese Meldungen innerhalb der Prozeduren zu verarbeiten. Vermutlich werden Sie zunächst einfach mit Ausgabeparametern arbeiten, so wie wir es ja auch beim Aufruf unserer HLL-Programme jahrelang getan haben. Diese Lösung ist tatsächlich einfach und sähe vermutlich folgendermaßen aus:

```
CREATE OR REPLACE PROCEDURE DUMMYPROC
                  (OUT MSG VARCHAR(1000) ,
                   OUT STATUS CHAR(5) )
        LANGUAGE SQL
        SPECIFIC DUMMYPROC
        NOT DETERMINISTIC
        READS SQL DATA
        CALLED ON NULL INPUT
        SET OPTION COMMIT = *NONE ,
        DATFMT = *EUR,
        TIMFMT = *EUR,
        DBGVIEW = *SOURCE
        BEGIN
/* eine business-regel wird verletzt */
        SET MSG = 'DER PREIS IST NICHT GÜLTIG!', STATUS = '88I01';

        END ;
```

Das aufrufende Programm kann anschließend die Variablen STATUS und MSG auswerten und entsprechend darauf reagieren.

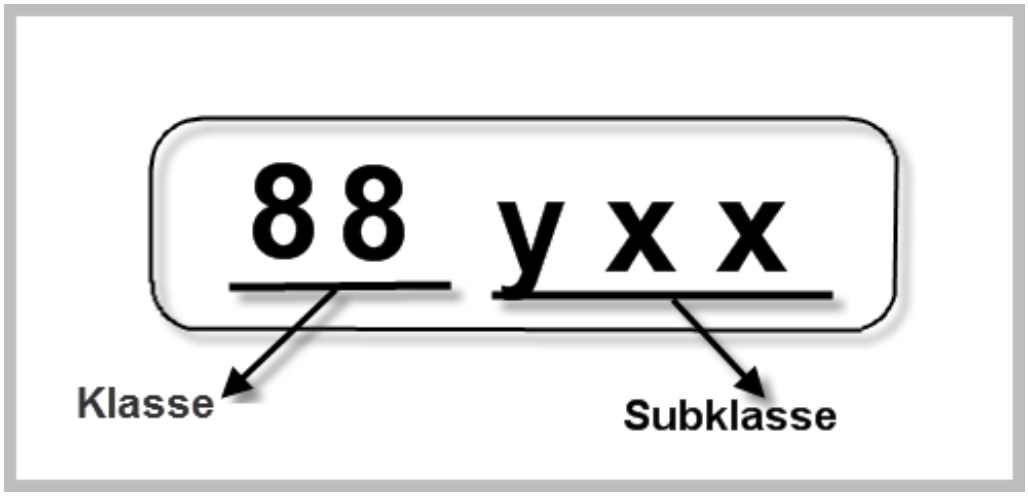
7.10.4
Seite 2

7.10.4.1 Benutzerdefinierte Meldungen auslösen

Innerhalb der SQL-Programmierung können wir aber auch unsere Geschäftslogik auf die Datenbankebene verlagern und mit dem Aufrufer kommunizieren, ohne hierfür Parameter zu verwenden. Voraussetzung dafür sind eigene SQLSTATI.

Bereits von Beginn an sollten Sie sich dabei an die folgenden Regeln halten: Natürlich könnten Sie jeden beliebigen SQLSTATUS für Ihre Meldungen nutzen, aber um Konflikte mit bestehenden SQLSTATI und zukünftigen Releases zu vermeiden, hat IBM bereits Wertebereiche für Ihre Fehlermeldungen reserviert und garantiert diese auch in der Zukunft. Daher sollten Sie den Bereich 01Hxx nutzen, wenn Warnungen ausgegeben werden. Die Buchstaben xx repräsentieren hierbei beliebige Großbuchstaben oder Zahlen. Alle SQLSTATI, die dieser Klasse und der Subklasse Hxx angehören, werden als Warnung interpretiert und als SQLCODE +438 umgesetzt.

Soll hingegen ein Fehler signalisiert werden, sollten Sie die Klasse aus dem Wertebereich 7–9 oder I–Z wählen. Den SQLSTATUS 88I01 könnte man beispielsweise problemlos verwenden. Ein SQLSTATUS, der der Klasse 7x bis 9x oder Ix bis Zx angehört, wird immer als SQLCODE -438 umgesetzt.



Wertebereiche eigener SQLSTATI

Nachdem wir jetzt wissen, welche Wertebereiche uns für die eigenen Fehlercodes zur Verfügung stehen, fehlt noch die entscheidende Anweisung: Mit Hilfe der Anweisung SIGNAL kann ein Fehler initiiert werden. Alternativ kann hierfür auch die Anweisung RAISE_ERROR verwendet werden. RAISE_ERROR kennen Sie vielleicht, wenn Sie bislang mit einem Microsoft SQL-Server gearbeitet haben. Die Anweisung RESIGNAL hingegen repliziert einen ursprünglichen Fehler und kann nur in einem Handler genutzt werden.

7.10.4
Seite 2

Schauen wir uns im ersten Schritt Syntax und Wirkung einer SIGNAL-Anweisung an. Dazu nutzen wir folgende Dummy-Prozedur:

```
CREATE OR REPLACE PROCEDURE DUMMYPROC ( )
       DYNAMIC RESULT SETS 0
       LANGUAGE SQL
       SPECIFIC DUMMYPROC
       NOT DETERMINISTIC
       READS SQL DATA
       CALLED ON NULL INPUT
       SET OPTION COMMIT = *NONE ,
       DATFMT = *EUR,
       TIMFMT = *EUR,
       DBGVIEW = *SOURCE
       BEGIN

       SIGNAL SQLSTATE '88I01'
       SET MESSAGE_TEXT = 'DER PREIS IST NICHT GÜLTIG!';

       END ;
```

Die Anweisung SIGNAL können Sie an einer beliebigen Stelle Ihrer Prozedur verwenden – auch innerhalb einer Handler-Deklaration, falls dies erforderlich sein sollte. Der Statuswert muss nicht zwingend als Konstante angegeben werden, genauso gut könnten Sie eine Variable verwenden. Nun bestimmen Sie, welche Informationen die Signal-Anweisung liefern soll. In meinem Beispiel setze ich zunächst lediglich einen individuellen Nachrichtentext. Auf weitere Informationen verzichte ich.

Schauen wir uns die Wirkung an:

```
> call dummyProc()

SQL-Status: 88I01
Vendorencode: -438
Nachricht: [SQL0438] Der Preis ist nicht gültig!
```

Wirkung der Signal-Anweisung

Ein Web-, C#- oder Java-Client könnte jetzt mit der entsprechenden Methode diese Informationen lesen und weiterverarbeiten.

Innerhalb der SIGNAL-Anweisung können wir aber noch weitere Informationen verwenden:

SIGNAL-Information	Beschreibung
CATALOG_NAME	Name der Datenbank, die die Fehlerbedingung ausgelöst hat
CLASS_ORIGIN	Fehlerklasse
COLUMN_NAME	Name der Spalte, die den Fehler verursacht hat
CONSTRAINT_NAME	Constraint-Name, der den Fehler ausgelöst hat
CONSTRAINT_ SCHEMA	CONSTRAINT-Schema, das die RI-Bedingung enthält
CURSOR_NAME	Name des Cursors, der den Fehler verursacht hat
MESSAGE_TEXT	der zu sendende Nachrichtentext
TABLE_NAME	Name der Tabelle, die durch die Fehlersituation betroffen ist

Oft werde ich gefragt, wofür diese weiteren Informationen sinnvoll sind, denn zunächst scheinen sie ohne Bedeutung zu sein, wenn man sich das nachfolgende Beispiel anschaut:

```
CREATE OR REPLACE PROCEDURE DUMMYERRORPROC ( )
        DYNAMIC RESULT SETS 0
        LANGUAGE SQL
        SPECIFIC DUMERRPROC
        NOT DETERMINISTIC
        READS SQL DATA
        CALLED ON NULL INPUT
        SET OPTION COMMIT = *NONE ,
        DATFMT = *EUR,
        TIMFMT = *EUR,
        DBGVIEW = *SOURCE
        BEGIN

        SIGNAL SQLSTATE '88I01'
        SET MESSAGE_TEXT = 'DER PREIS IST NICHT GÜLTIG!',
        COLUMN_NAME = 'VKPRICE'; TABLE_NAME = 'AUFTRAGSPOSITION';

        END ;
```

Sie sehen, ich habe meine SIGNAL-Anweisung erweitert, doch die Ausgabe bleibt unverändert:

```
> call dummyErrorProc()

SQL-Status: 88I01
Vendorencode: -438
Nachricht: [SQL0438] Der Preis ist nicht gültig!
```

Wirkung der Signal-Anweisung mit zusätzlichen Informationen

Wofür also kann ich diese Informationen nutzen? Um diese Frage zu klären, gehen wir einige Schritte zurück. In den vorangegangenen Kapiteln haben wir uns stets mit Fehlermeldungen der Datenbank beschäftigt und haben in diesem Zusammenhang Exception-Handler sowie die Anweisung GET DIAGNOSTICS kennengelernt. Greifen wir dieses Wissen jetzt noch einmal auf.

Manchmal entsteht der Wunsch, globale Fehlerroutinen zu implementieren und ein zentrales Fehlerprotokoll zu schreiben. Denn anders als in den HLL-Programmen stürzen unsere Prozeduren bei einem SQL-Fehler nicht ab und wir erhalten auch keine Joblogs, da die Client-Jobs nicht beendet werden, sondern diverse Anfragen beantworten. Administratoren sprechen in diesem Zusammenhang immer von den PJ-Jobs (Prestarted Jobs). Um hier Abhilfe zu schaffen, könnten wir eine zentrale LOG-Datei aufbauen:

```
CREATE TABLE EPSW09/GLOBAL_ERROR_SQL_TABLE FOR SYSTEM NAME GERSQLTAB
(
        PKID INTEGER GENERATED ALWAYS AS IDENTITY (
        START WITH 1 INCREMENT BY 1
        NO MINVALUE NO MAXVALUE
        NO CYCLE NO ORDER
        CACHE 20 ) ,
        ZEITSTEMPEL FOR COLUMN ZEITSTEMP TIMESTAMP
                            DEFAULT CURRENT_TIMESTAMP ,
        PROCEDURNAME FOR COLUMN PNAME CHAR(128) DEFAULT NULL ,
        VARIABLENINHALTE FOR COLUMN VARVALUE CHAR(1000) DEFAULT NULL ,
        COMMAND_FUNCTION FOR COLUMN COMFUNC VARCHAR(128) DEFAULT NULL ,
        …..
        ….

        RETURNED_SQLSTATE FOR COLUMN RSQLSTATE CHAR(5) DEFAULT NULL ,
        ROUTINE_NAME FOR COLUMN PROCNAME VARCHAR(128) DEFAULT NULL ,
        ROUTINE_SCHEMA FOR COLUMN PROCSCHEMA VARCHAR(128) DEFAULT NULL ,
        SCHEMA_NAME FOR COLUMN TABSCHEMA VARCHAR(128) DEFAULT NULL ,
        SERVER_NAME FOR COLUMN SERVNME VARCHAR(128) DEFAULT NULL ,
        SPECIFIC_NAME FOR COLUMN SPECNAME VARCHAR(128) DEFAULT NULL ,
        TABLE_NAME FOR COLUMN TABNAME VARCHAR(128) DEFAULT NULL ,
        TRIGGER_NAME FOR COLUMN TRGNAM VARCHAR(128) DEFAULT NULL ,
        TRIGGER_SCHEMA FOR COLUMN TRIGSCHE VARCHAR(128) DEFAULT NULL )

        RCDFMT GERSQLTABF ;
```

Wenn sie sich die Tabellendefinition genauer ansehen, können Sie erkennen, dass meine Tabelle eine fortlaufende ID, einen Zeitstempel, eine Spalte für den Prozedurnamen und eine Spalte für die Variableninhalte enthält. Diese Spalten sind völlig individuell gestaltet. Die folgenden Spalten erinnern an die Anweisung GET DIAGNOSTICS (beispielsweise RETURNED_SQLSTATE, TRIGGER_NAME usw.) Mit Hilfe der GET DIAGNOSTICS-Anweisung waren wir in der Lage, diverse Informationen über unser zuletzt ausgeführtes SQL-Statement zu ermitteln. Welche dieser Informationen in der Tabelle GLOBAL_ERROR_SQL_TABLE gespeichert werden sollen, bleibt Ihnen überlassen.

Im nächsten Schritt benötigen wir eine universelle Prozedur, die im Fehlerfall grundsätzlich aufgerufen wird und die Aufgabe hat, die entsprechenden Informationen mit der GET DIAGNOSTICS-Anweisung zu ermitteln und anschließend in die Tabelle GLOBAL_ERROR_SQL_TABLE zu speichern. Diese Prozedur enthält nur eine Besonderheit:

```
CREATE PROCEDURE GLOBAL_SQL_ERR_HANDLER
       (
       IN I_SQLANW VARCHAR(128) ,
       IN I_VARIABLEN VARCHAR(1000) ,
       )
….

BEGIN
/* LOKALE VARIABLEN */

…

/***********************PGM STARTET***********************
GET STACKED DIAGNOSTICS
V_COMMAND_FUNCTION = COMMAND_FUNCTION ,
…

GET STACKED DIAGNOSTICS CONDITION 1
V_CATALOG_NAME = CATALOG_NAME ,
V_COLUMN_NAME = COLUMN_NAME ,
…

INSERT INTO GLOBAL_ERROR_SQL_TABLE
(
PROCEDURNAME , VARVALUE , COMMAND_FUNCTION , DB2_LAST_ROW , …
);
```

Da die externe Prozedur Global_SQL_ERR_Handler stets die Diagnostic Area der aufrufenden Prozedur lesen soll, müssen wir jetzt das Schlüsselwort STACKED verwenden. Dadurch lesen wir die Informationen des SQL-Statements, das als letztes außerhalb eines Handlers ausgeführt wurde. Anschließend schreiben wir diese Informationen in unsere Tabelle GLOBAL_ERROR_SQL_TABLE.

Schauen wir uns jetzt an, wie die Prozedur GLB_SQL_ERR_HANDLER arbeitet. Dazu benutzen wir wieder unsere Dummy-Prozedur. Innerhalb der Prozedur soll die Tabelle Auftragsposition gelesen werden.

```
CREATE OR REPLACE PROCEDURE DUMMYERRORPROC
       (
       IN I_AUFNR SMALLINT DEFAULT NULL ,
       OUT O_ANZAHL SMALLINT
       )
       DYNAMIC RESULT SETS 0
       LANGUAGE SQL
       SPECIFIC DUMERRPROC
       NOT DETERMINISTIC
       MODIFIES SQL DATA
       CALLED ON NULL INPUT
       PROGRAM TYPE SUB
       SET OPTION ….
BEGIN
/* LOKAL VARIABLES */
DECLARE #SQLANW VARCHAR ( 128 ) DEFAULT 'TESTERRORHANDLING' ;
DECLARE #VARIABLEN VARCHAR ( 1000 ) DEFAULT NULL ;

/*********** GLOBAL EXCEPTION HANDLING ****************************/

DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
       SET #VARIABLEN = '#' || TRIM(CHAR ( I_AUFNR ) ) || '#';
       CALL GLOBAL_SQL_ERR_HANDLER
       ( #SQLANW , #VARIABLEN) ;
END ;
```

Schauen Sie sich die Fehlerroutine an: In der Routine rufe ich die externe Prozedur GLOBAL_SQL_ERR_HANDLER im Fehlerfall auf und übergebe die Variableninhalte und den Prozedurnamen an meine Fehlerroutine. Die Prozedur DUMMYERRPROC hat ansonsten keine große Aufgabe, wie Sie nachfolgend erkennen können. Sie ermittelt lediglich die Positionsanzahl für einen Auftrag. Damit die Prozedur aber einen Fehler erzeugt und wir unsere globale Fehlerroutine testen können, wurde der Tabellenname falsch geschrieben.

7.10.4
Seite 8

```
-------------------------PROGRAMM START-------------------------

SELECT COUNT(*) INTO O_ANZAHL

FROM AUFTRAGPOSITION WHERE AUFTRAGSNUMMER = I_AUFNR;

-- TABELLENNAME IST FALSCH!!!

-----------------------------------------------------------------

END ;
```

Jetzt rufen wir die Prozedur DummyErrorProc auf und erhalten das folgende Ergebnis:

```
call DummyErrorProc ( 1, ?);
```

```
[ Thu Feb 12 16:06:18 CET 2015 ]  Auswahl ausführen

> call DummyErrorProc ( 1, ?)

Rückkehrcode = 0

Ausgabeparameter 2 = <NULL>

Anweisung wurde erfolgreich ausgeführt  (78 ms)
```

Aufruf der Testprozedur

Schnell erkennt man, dass wir aufgrund unseres Handlers keine Exception mehr bekommen. Fast scheint es, als sei die Prozedur fehlerfrei gelaufen. Hat unsere zentrale Fehlerroutine denn wenigstens reagiert?

```
select  Zeitstempel, procedurname, variableninhalte,
command_function, message_text, returned_sqlstate,
Table_name, schema_name, Db2_authorization_id,  db2_module_detecting_error
from global_Error_sql_Table;
```

ZEITSTEMPEL	PROCEDURNAME	VARIABLE...	COMMAN...	MESSAGE_TEXT	RETURNED_SQL...	TABLE_NAME
2015-02-12 15:59:41....	TestErrorHandling ...	#1# ...	SELECT	AUFTRAGPOSITION der Art *FILE in EPSW09 ni...	42704	AUFTRAGPOSITION
2015-02-12 16:01:25....	TestErrorHandling ...	#1# ...	SELECT	AUFTRAGPOSITION der Art *FILE in EPSW09 ni...	42704	AUFTRAGPOSITION
2015-02-12 16:06:28....	TestErrorHandling ...	#1# ...	SELECT	AUFTRAGPOSITION der Art *FILE in EPSW09 ni...	42704	AUFTRAGPOSITION

Inhalt der zentralen LOG-Datei

Die Fehlersituation wurde ordnungsgemäß erkannt und in der Tabelle GLOBAL_ERROR_SQL_TABLE dokumentiert. Jetzt muss nur noch die Client-Applikation über die Fehlersituation informiert werde. Das geschieht ganz einfach mit der Anweisung RESIGNAL.

```
CREATE OR REPLACE PROCEDURE DUMMYERRORPROC
       (
       IN I_AUFNR SMALLINT DEFAULT NULL ,
       OUT O_ANZAHL SMALLINT
       )
       DYNAMIC RESULT SETS 0
       LANGUAGE SQL
       SPECIFIC DUMERRPROC
       NOT DETERMINISTIC
       MODIFIES SQL DATA
       CALLED ON NULL INPUT
       PROGRAM TYPE SUB
       SET OPTION ….
BEGIN
/* LOKAL VARIABLES */
DECLARE #SQLANW VARCHAR ( 128 ) DEFAULT 'TESTERRORHANDLING' ;
DECLARE #VARIABLEN VARCHAR ( 1000 ) DEFAULT NULL ;

/******* GLOBAL EXCEPTION HANDLING ******************************/

DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
       SET #VARIABLEN = '#' || TRIM(CHAR ( I_AUFNR ) ) || '#';
       CALL GLOBAL_SQL_ERR_HANDLER
       ( #SQLANW , #VARIABLEN) ;
       RESIGNAL;
END ;
```

7.10.4
Seite 10

Ich füge in meiner Prozedur DUMMYERRPROC lediglich die Anweisung RESIGNAL hinzu und veranlasse damit, dass der zuletzt aufgetretene Fehler erneut gesendet wird. Jetzt passt das Ergebnis:

```
call DummyErrorProc ( 1, ?);

SQL-Status: 42704
Vendorencode: -204
Nachricht: [SQL0204] AUFTRAGPOSITION der Art *FILE in *LIBL nicht gefunden. Ursache . . . . :
AUFTRAGPOSITION der Art *FILE in *LIBL wurde nicht gefunden. Ist der Teildateiname *ALL, ist die
Tabelle nicht partitioniert. Handelt es sich um eine Anweisung ALTER TABLE und ist die Art *N, wurde
```

Die Fehlermeldung wurde ausgelöst

Und gleichzeitig wurde der Fehler in der LOG-Datei protokolliert:

ZEITSTEM...	PROCEDURNAME	VARIA...	COMMAN...	MESSAGE_TEXT	RETU...	TABLE_NAME
2015-02-13 ...	TestErrorHandling ...	#1# ...	SELECT	AUFTRAGPOSITION der Art *FILE in *LIBL nicht ...	42704	AUFTRAGPOSITION
2015-02-13 ...	TestErrorHandling ...	#1# ...	SELECT	AUFTRAGPOSITION der Art *FILE in *LIBL nicht ...	42704	AUFTRAGPOSITION

Der Fehler steht auch in der LOG-Datei

Jetzt kommen wir zurück zu unseren Business-Fehlern. Wenn wir auch diese Fehler in der LOG-Datei dokumentieren wollen, benötigen wir auch hier Informationen für die Diagnostic-Area, andernfalls bleibt die GET DIAGNOSTICS-Anweisung in der globalen Fehlerroutine teilweise wirkungslos.

Schauen wir uns das nächste Beispiel an:

```
CREATE OR REPLACE PROCEDURE EPSW09/DUMMYERRORPROC
       (
       IN I_AUFNR SMALLINT DEFAULT NULL ,
       IN I_ARTNR CHAR (5) DEFAULT NULL,
       IN I_RABATT DEC(4,2) DEFAULT 0
       )
       …
       …

BEGIN
/* LOKALE VARIABLEN */
DECLARE #SQLANW VARCHAR ( 128 ) DEFAULT 'TESTERRORHANDLING' ;
DECLARE #VARIABLEN VARCHAR ( 1000 ) DEFAULT NULL ;

/*********** GLOBAL EXCEPTION HANDLING ***************************/

DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
       SET #VARIABLEN = '#' || TRIM(CHAR ( I_AUFNR ) ) || '#' ||
I_ARTNR || '#' ||
                CHAR ( I_RABATT ) || '#';

       CALL GLOBAL_SQL_ERR_HANDLER ( #SQLANW , #VARIABLEN ) ;
       RESIGNAL;

END ;

….
….

IF     (SELECT ARTIKELPREIS – I_RABATT FROM AUFTRAGSPOSITION
       WHERE AUFTRAGSNUMMER = I_AUFNR AND ARTIKELNUMMER = I_ARTNR)

>= 0 THEN

UPDATE AUFTRAGSPOSITION SET ARTIKELPREIS = ARTIKELPREIS – I_RABATT,
POSITIONSWERT = ARTIKELPREIS – I_RABATT * POSITIONSMENGE
WHERE AUFTRAGSNUMMER = I_AUFNR AND ARTIKELNUMMER = I_ARTNR;

ELSE

SIGNAL SQLSTATE '88I01' SET
MESSAGE_TEXT = 'VK-PREIS DARF NICHT KLEINER ODER GLEICH 0 SEIN!';

END IF;
```

7.10.4
Seite 12

Wie bereits angedeutet, habe ich nun eine Preisprüfung implementiert, so dass ein Fehler ausgegeben wird, falls der Preis kleiner oder gleich 0 ist. In unserer LOG-Tabelle stehen jetzt weniger Informationen zur Verfügung:

MESSAGE_TEXT	RETURNE...	TABLE_NAME	SCHEMA...	DB2_AUT...
VK-Preis darf nicht kleiner oder gleich 0 sein!	88I01			RASCHE
VK-Preis darf nicht kleiner oder gleich 0 sein!	88I01			RASCHE
VK-Preis darf nicht kleiner oder gleich 0 sein!	88I01			RASCHE

Tabellen und Spaltenname stehen nicht zur Verfügung

Benötigen Sie auch diese Informationen, um die Business-Fehler zu dokumentieren, müssen Sie lediglich die SIGNAL-Anweisung ergänzen:

```
VALUES( CURRENT SCHEMA) INTO #SCHEMA_NAME ;

SIGNAL SQLSTATE '88I01' SET
MESSAGE_TEXT = 'VK-PREIS DARF NICHT KLEINER ODER GLEICH 0 SEIN!',
COLUMN_NAME = 'ARTIKELPREIS' ,
TABLE_NAME = 'AUFTRAGSPOSITION',
SCHEMA_NAME = #SCHEMA_NAME;
```

Nachfolgende Tabelle zeigt, welche Informationen Sie in einer SIGNAL-Anweisung verwenden können.

SIGNAL-Information	Beschreibung
CATALOG_NAME	Name der Datenbank, die die Fehlerbedingung ausgelöst hat
CLASS_ORIGIN	Fehlerklasse
COLUMN_NAME	Name der Spalte, die den Fehler verursacht hat
CONSTRAINT_NAME	Constraint-Name, der den Fehler ausgelöst hat
CONSTRAINT_SCHEMA	CONSTRAINT-Schema, das die RI-Bedingung enthält
CURSOR_NAME	Name des Cursors, der den Fehler verursacht hat
MESSAGE_TEXT	der zu sendende Nachrichtentext
TABLE_NAME	Name der Tabelle, die durch die Fehlersituation betroffen ist

Bleiben wir noch ein klein wenig bei dem eben gezeigten Beispiel. Meine Auftragspositionstabelle steht in Beziehung mit der Tabelle AUFTRAG.

```
select * from auftrag
where Auftragsnummer = 201 ;
```

AUFT...	KUNDEN...	AUFTRAG...	AUSLIEF...	AUFTRAGSWERT	VE...	R...	S...
00201	70000	21.04.2010	30.11.2010	100.00	170	3.20	01.0...

Die Tabelle AUFTRAG und der Auftragswert

Ganz gleich, ob wir dieses Datenbankdesign gut heißen oder nicht, ich muss den Auftragswert aktualisieren, wenn ich für eine der Auftragspositionen einen Rabatt gewähre. Die Anforderung ist schnell umgesetzt:

```
IF (SELECT ARTIKELPREIS - I_RABATT FROM AUFTRAGSPOSITION
WHERE AUFTRAGSNUMMER = I_AUFNR AND ARTIKELNUMMER = I_ARTNR)
>= 0 THEN

UPDATE AUFTRAGSPOSITION
          SET ARTIKELPREIS = ARTIKELPREIS - I_RABATT,
          POSITIONSWERT = (ARTIKELPREIS - I_RABATT) * POSITIONSMENGE
          WHERE AUFTRAGSNUMMER = I_AUFNR AND ARTIKELNUMMER = I_ARTNR;

          UPDATE AUFTRAG
          SET AUFTRAGSWERT = (  SELECT SUM(POSITIONSWERT)
                                FROM AUFTRAGSPOSITION
                   WHERE AUFTRAGSNUMMER = I_AUFNR)
          WHERE AUFTRAGSNUMMER = I_AUFNR;
ELSE

          VALUES( CURRENT SCHEMA) INTO #SCHEMA_NAME ;
          SIGNAL SQLSTATE '88I01'
          SET MESSAGE_TEXT = 'VK-PREIS DARF NICHT KLEINER ODER GLEICH 0 SEIN!',
          COLUMN_NAME = 'ARTIKELPREIS',
          TABLE_NAME = 'AUFTRAGSPOSITION',
          SCHEMA_NAME = #SCHEMA_NAME;

END IF;
```

Unsere globale Fehlerroutine wird auch weiterhin zuverlässig arbeiten und jeden Fehler dokumentieren, der in der Prozedur auftritt. Aber was passiert, wenn beim UPDATE der Auftragstabelle ein Fehler auftritt?

VARIABLENINHALTE	COMMAND_FUNCTION	MESSAGE_TEXT
#201#110 #10.00 # ...	SIGNAL	VK-Preis darf nicht kleiner oder gleich 0 sein!
#201#110 #56.00 # ...	SIGNAL	VK-Preis darf nicht kleiner oder gleich 0 sein!
#201#110 #56.00 # ...	SIGNAL	VK-Preis darf nicht kleiner oder gleich 0 sein!
#201#110 #56.00 # ...	SIGNAL	VK-Preis darf nicht kleiner oder gleich 0 sein!
#201#110 #56.00 # ...		Spalte oder globale Variable #CURRENT_SCHEMA nicht gefunden.
#201#110 #56.00 # ...	SIGNAL	VK-Preis darf nicht kleiner oder gleich 0 sein!
#201#110 #.10 # ...	UPDATE WHERE	Zeile oder Objekt AUFTRAG der Art *FILE in EPSW09 wird verwendet.

Der Fehler ist ordnungsgemäß dokumentiert

Aber unsere Datenbasis ist fehlerhaft! Und hier ist die globale Fehlerroutine nicht mehr ausreichend. Wir benötigen Transaktionen, um diese Fehler auch auf der Datenbasis zu korrigieren.

7.10.4
Seite 14

7.10.6 Transaktionssicherung im SQL-Umfeld

Sicherlich wissen Sie bereits, welch mächtige Möglichkeiten Ihnen SQL bietet, um Daten zu selektieren. Und wahrscheinlich haben Sie auch festgestellt, dass es sehr einfach ist, mit entsprechenden INSERT-, UPDATE- und DELETE-Anweisungen Daten zu manipulieren. Wenn Sie nur hin und wieder mit SQL arbeiten, dürfte dieses Wissen ausreichen. Haben Sie allerdings vor, eine professionelle SQL-Anwendung zu entwickeln, sollten Sie sich irgendwann mit dem Begriff der Transaktionssicherung auseinandersetzen. Der Begriff Transaktionssicherung (= Transaktionsmanagement) beschreibt die Fähigkeit eines relationalen Datenbankmanagementsystems (DBMS), verschiedene SQL-Anweisungen zu einer Arbeitseinheit zu bündeln und diese Arbeitseinheit ganz oder gar nicht auszuführen. Meist spricht man in diesem Zusammenhang besser nicht von einer Arbeitseinheit, sondern von einer „Logical Unit of Work“ (LUW). Ganz gleich, wie Sie es nennen, eine LUW hat immer einen definierten Anfang und ein definiertes Ende, und wenn irgendetwas während der Transaktion schiefgeht, lässt sich die LUW einfach abbrechen. Läuft hingegen alles zur Zufriedenheit, kann man die gesamte LUW endgültig speichern. Diese Eigenschaft der Transaktionen gehört zu den sogenannten ACID-Eigenschaften. ACID (deutsch AKID) ist eine Abkürzung für erwünschte Eigenschaften in Datenbankmanagementsystemen (DBMS) und verteilten Systemen. ACID steht für Atomicity (Atomarität), Consistency (Konsistenz), Isolation (Isoliertheit) und Durability (Dauerhaftigkeit). Diese Eigenschaften gelten als Voraussetzungen für die Verlässlichkeit von IT-Systemen.

Einige Leser werden sich jetzt fragen, was hierbei neu ist? Denn eigentlich gibt es schon seit langem – auch im RPG-Umfeld – den Begriff der Transaktionen und wir konnten auch in diesem Umfeld mit ROLLBACK- (ROLBK) und COMMIT-Befehlen arbeiten. Nur haben wir es meistens nicht getan und wir kannten auch nur zwei Zustände unserer Datensätze – nämlich mit gesperrten und mit nicht gesperrten Zeilen. Mit SQL ändert sich dieses Umfeld erheblich. Nicht nur, dass wir es mit Mengenverarbeitung im Gegensatz zur Einzelsatzverarbeitung zu tun bekommen, es gibt auch immer mehr Webportale, über die beispielsweise Aufträge abgewickelt werden. Und wie reagieren Sie in diesem Umfeld, wenn sich der Anwender erst auf der letzten Seite der Auftragserfassung dazu entschließt den Button „Abbrechen“ zu benutzen? Und ist die Verbindung über Internet wirklich genauso stabil wie unser internes Firmennetz? Hinzu kommt, dass wir im SQL-Umfeld verschiedene Sperrstufen kennen, die kleine aber feine Unterschiede aufweisen.

Lassen Sie uns das Problem der Transaktionssicherung an dem vorangegangenen Beispiel erarbeiten:

Wir gewähren zunächst den Rabatt in der Auftragsposition, indem wir den Artikelpreis und den Positionswert reduzieren. Anschließend korrigieren wir den Auftragswert in der Kopfdatei.

Sicherlich würde das Programm auch in Multiuser-Umgebungen korrekt arbeiten – auch ohne Transaktionssicherung. Schließlich legt der SQL-Standard fest, dass das folgerichtige Lesen auf Anweisungsebene durch das DBMS unterstützt werden muss. Das bedeutet aber nur, dass die Daten für die Zeit des Datenabrufs konsistent bleiben. Wir benötigen aber in den meisten Fällen – so auch in unserem Beispiel – die Datenkonsistenz über einen längeren Zeitraum oder anders gesagt über mehrere Programmanweisungen hinweg.

Es wird immer Programmierer geben, die an dieser Stelle die Frage aufwerfen: „Wann kommt es denn überhaupt zu Fehlersituationen?“ Diese Frage können nur Sie beantworten! Wenn die Wahrscheinlichkeit gegen Null geht, können Sie vielleicht auf Transaktionen verzichten. Sie sollten jedoch berücksichtigen, dass unsere Umgebungen immer komplexer werden und nicht immer sind es Programmfehler, die Probleme verursachen:

Die Aktualisierung der Auftragskopfdatei wird möglicherweise verhindert, weil der erforderliche Satz in einem anderen Anwendungsprozess gesperrt ist. Oder der Satz kann nicht aktualisiert werden, weil gegen eine Datenbankbedingung verstoßen wird. Es gibt viele denkbare Szenarien, die Probleme verursachen könnten.

Jeder Programmfehler und die damit verbundenen Buchungsfehler führen stets zu einer inkonsistenten Datenbasis und die hat dann ggf. Auswirkungen auf Bestellmengen im Einkauf oder auf die Kundenzufriedenheit.

7.10.6.1 Transaktionen starten und beenden

Ich werde jetzt meine Aktualisierungen durch Transaktionen absichern. Voraussetzung für die Arbeit mit Transaktionen ist die Journalisierung der Datenbankdateien. Ohne Journalisierung können Sie nicht mit Transaktionen arbeiten.

Bevor ich die Transaktion integriere, stellt sich zunächst die Frage: „Wie starten wir eine Transaktion?“

Grundsätzlich können wir die LUW implizit am Programmanfang starten, beim Ausführen eines SQL-Skripts oder wenn ein Datenbank-Connect stattfindet. Allerdings ist es ebenfalls möglich, die Transaktion explizit zu öffnen, indem wir die Anweisung SET TRANSACTION verwenden.

Ich bevorzuge grundsätzlich die explizite Variante. So habe ich die vollständige Kontrolle über die Transaktion. Ich entscheide, wann die Transaktion beginnt und kann die Transaktion so kurz wie möglich aufbauen.

Merke:

- Jede Transaktion wird nur so lange wie nötig, aber so kurz wie möglich aufgebaut. Halten Sie sich nicht an diese Regel, werden Sie sehr schnell Sperrkonflikte und „Dirty Reads“ bekommen.
- Eine Transaktion sollte niemals einen Anwenderdialog beinhalten. Sie können nicht wissen, wann der Anwender den Dialog quittiert. Vielleicht kommt ein Telefonanruf dazwischen oder gar die Mittagspause! Auch in diesem Fall wären Sperrkonflikte vorprogrammiert.

Beenden Sie die Transaktion – egal wie sie gestartet wurde – explizit mit den SQL-Anweisungen COMMIT bzw. ROLLBACK. Vergessen Sie, eine Transaktion explizit zu beenden, sind auch hier Vorkehrungen getroffen. Endet das Programm, der Job oder die Datenbankverbindung normal oder abnormal ohne explizite COMMIT-Anweisung, findet standardmäßig immer eine ROLLBACK-Operation statt. Es könnte daher möglich sein, dass Sie viele Stunden in der Prozedurumgebung des System i Navigators arbeiten, um Daten zu erfassen oder zu manipulieren und dann den Navigator schließen, OHNE die LUW zu beenden. Ihre Arbeit war vergebens! Standardmäßig wird in diesen Fällen ein ROLLBACK ausgelöst. Dieses Szenario ist natürlich in jedem Umfeld denkbar und beschränkt sich nicht nur auf System i Navigator. So ärgerlich dieses Phänomen sein kann, wenn der implizite ROLLBACK unbeabsichtigt stattfindet, weil Sie vergessen haben, die Transaktion zu beenden, so positiv ist dieses Verhalten, wenn Programme abstürzen oder die Datenbankverbindung plötzlich abbricht: Ihre Datenbank bleibt stets konsistent.

Wir wollen aber weder die Transaktionen implizit starten noch sie implizit beenden. Unsere Transaktion soll mit der SQL-Anweisung SET TRANSACTION gestartet werden, bevor wir unseren Rabatt buchen.

Schauen wir uns dazu wieder die Prozedur DUMMYERRORPROC an:

```
CREATE OR REPLACE PROCEDURE DUMMYERRORPROC
       (
       IN I_AUFNR SMALLINT DEFAULT NULL ,
       OUT O_ANZAHL SMALLINT
       )
       DYNAMIC RESULT SETS 0
       LANGUAGE SQL
       SPECIFIC DUMERRPROC
       NOT DETERMINISTIC
       MODIFIES SQL DATA
       CALLED ON NULL INPUT
       PROGRAM TYPE SUB
       SET OPTION COMMIT = *NONE, ….

BEGIN
```

Auch wenn ich Transaktionen verwenden will, behalte ich die Option COMMIT = *NONE bei. Die Transaktion soll schließlich nicht starten, wenn die Prozedur startet, sondern erst dann, wenn tatsächlich ein UPDATE ausgeführt wird.

```
SET TRANSACTION ISOLATION LEVEL XXXX;

UPDATE AUFTRAGSPOSITION……;
UPDATE AUFTRAG ……..;

COMMIT;
```

Unmittelbar vor der Aktualisierung der Positionsdatei öffne ich die Transaktion, führe die Aktualisierungen durch und schreibe die Satzänderungen fest, wenn keine Fehler aufgetreten sind.

7.10.6.2 Die verschiedenen Isolationsstufen

Bei SQL müssen wir aber auch die Isolationsstufe der Transaktion benennen. Grundsätzlich haben wir hier fünf verschiedene Stufen zur Auswahl, die sich jeweils durch ihren Isolationsgrad unterscheiden.

```
SET TRANSACTION ISOLATION LEVEL NO COMMIT;
```

Die Isolationsstufe NO COMMIT ist System i-spezifisch und daher auf anderen Datenbankservern nicht vorhanden. Wir benötigen diese Isolationsstufe nur, um Datenmanipulationen an Tabellen vorzunehmen, die nicht journalisiert sind. Anders ausgedrückt: Wenn Sie die Isolationsstufe NO COMMIT wählen, gibt es keine Transaktionssicherheit. Alle Datenbankänderungen werden unmittelbar festgeschrieben. **Die Anweisungen COMMIT und ROLLBACK sind wirkungslos, führen jedoch nicht zu Fehlermeldungen.**

Da IBM bereits sehr früh Transaktionslevels eingeführt hat, ist die Benennung leider nicht durchgängig und häufig auch nicht ANSI-konform. Alternativ zur Bezeichnung NO COMMIT müssen Sie in anderen Umfeldern mit dem Synonym *NC bzw. *NONE arbeiten. In einer SET OPTION-Anweisung beispielsweise verwenden wir *NONE anstatt NO COMMIT.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Die Isolationsstufe READ UNCOMMITTED gehört zum ANSI-Standard und ist auch auf anderen Datenbankservern vorhanden. Die Isolationsstufe READ UNCOMMITTED bietet die schwächste Abschirmung. Nun könnte man sagen, dass Transaktionssicherung in diesem Level eigentlich gar nicht benötigt wird, denn Datenmanipulationen sind ohnehin nicht erlaubt, da diese Isolationsstufe automatisch die Transaktion in den Modus READ ONLY versetzt. Sie würden dann sofort folgende Fehlermeldung erhalten, wenn Sie dennoch versuchen Daten zu manipulieren:

```
SQL-Status: 25006
Vendorencode: -817
Nachricht: [SQL0817] Aktualisierungsoperation nicht zulässig. Ursache . . . . :  Es wurde versucht, eine
SQL-Anweisung auszuführen, die nach Ausführung von  SET TRANSACTION READ ONLY den Inhalt
einer Tabelle ändern oder ein Datenbankobjekt erstellen oder löschen würde. Fehlerbeseitigung:  SET
TRANSACTION READ WRITE angeben oder die Aktualisierungsoperation aus dem Programm entfernen
und die Anforderung wiederholen.
```

Read Committed – Fehlermeldung

ROLLBACK und COMMIT-Operationen sind in dieser Isolationsstufe daher überflüssig, aber es gibt ein anderes Phänomen, das auftreten könnte: DIRTY READ. Wir lesen in dieser Transaktionsstufe ggf. Daten anderer Anwender, die noch nicht mit COMMIT bestätigt wurden. Falls die Transaktionen in den anderen Applikationen später mit ROLLBACK zurückgesetzt werden, haben wir möglicherweise Daten gelesen, die für uns nie hätten existieren dürfen.

Aus den oben genannten Gründen sind die beiden genannten Level eher von untergeordneter Bedeutung. Spannend wird es erst, wenn wir die Isolationsstufe READ COMMITTED verwenden:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

UPDATE AUFTRAGSPOSITION SET ARTIKELPREIS = ARTIKELPREIS - I_
RABATT,
POSITIONSWERT = (ARTIKELPREIS - I_RABATT) * POSITIONSMENGE
WHERE AUFTRAGSNUMMER = I_AUFNR AND ARTIKELNUMMER = I_ARTNR;

UPDATE AUFTRAG SET AUFTRAGSWERT =
(SELECT SUM(POSITIONSWERT)
FROM AUFTRAGSPOSITION
WHERE AUFTRAGSNUMMER = I_AUFNR);

COMMIT;
```

Jetzt können wir unsere Aktualisierungen durchführen und am Ende festschreiben. Kommt es zu einer Fehlersituation, nehmen wir unsere Änderungen mit der Anweisung ROLLBACK in der Fehlerroutine zurück.

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
      SET #VARIABLEN = '#' || TRIM(CHAR ( I_AUFNR ) ) || '#'
|| I_ARTNR || '#' || CHAR ( I_RABATT ) || '#';
      CALL GLOBAL_SQL_ERR_HANDLER ( #SQLANW , #VARIABLEN ) ;
      ROLLBACK;
      RESIGNAL;

END ;
```

Und unsere Daten bleiben trotz Fehlersituation konsistent:

```
> call DummyErrorProc ( 201, '110' , 3.00)

SQL-Status: 57033
Vendorencode: -913
Nachricht: [SQL0913] Zeile oder Objekt AUFTRAG der Art *FILE in EPSW09 wird verwendet. Ursache . . .
:  Das angeforderte Objekt AUFTRAG der Art *FILE in der Bibliothek EPSW09 wird gerade von einem
anderen Anwendungsprozeß verwendet, oder eine Zeile im Objekt wird von einem anderen
```

Fehlersituationen lösen den Rollback aus

AU...	ARTIKELN...	ARTIKEL...	POSITION...	AUFTRAG...
201	110	0.80	8.00	80.00
201	251	3.20	32.00	80.00
201	108	1.00	40.00	80.00
201	45	1.00	0.00	80.00

Die Datenbasis ist korrekt

Dirty Reads kann es innerhalb dieser Isolationsstufe beim lesenden Zugriff übrigens nicht geben, da wir innerhalb einer Transaktion, die im Level Read Committed gestartet wurde, nur mit Commit bestätigte Zeilen lesen. Dennoch ist ein anderes Phänomen möglich. Stellen Sie sich Folgendes vor:

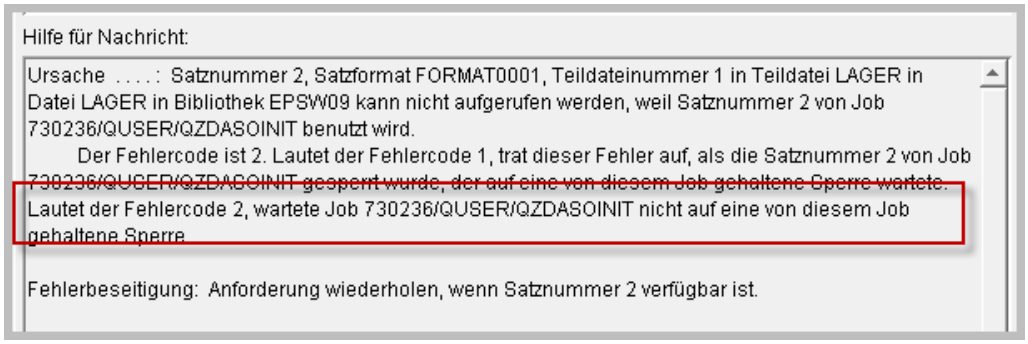
Am Programmanfang wird zunächst ein Wert ermittelt sowie ggf. die verfügbare Bestandsmenge. Wir erhalten also zum Beispiel den Wert 100 für den Artikel XY. Parallel zu unserer Bestandsermittlung beginnt eine zweite Transaktion in einem anderen Programm die Bestandsdaten dieses Artikels zu verändern. Denn eine zweite Transaktion kann uneingeschränkt UPDATE, DELETE und ROLLBACK auf die von uns gelesenen Zeilen anwenden, ohne dass unser Programm davon etwas merkt. Wenn unser Programm anschließend erneut auf die Bestandswerte zugreift, etwa um Reservierungen vorzunehmen, könnten sich die Bestände geändert haben. Wir können also die zu Beginn gelesene Artikelmenge nicht erneut lesen. Dieses Phänomen bezeichnet man als **Non Repeatable Read.** Auch wenn alle Daten gültig waren – also mit Commit bestätigt waren – und auch existieren, könnte – je nach Programmlogik und Anwendungsfall – unser Programm mit zwei verschiedenen Datenbeständen arbeiten. Das Problem DIRTY READ ist zwar durch die Isolationsstufe READ COMMITTED gelöst, aber das Phänomen NON REPEATABLE READ könnte auftreten. Ob die Transaktionsstufe READ COMMITTED ausreichend ist, bestimmt daher die jeweilige Situation.

Erst die Isolationsstufe REPEATABLE READ stellt sicher, dass gelesene Daten wiederholt gelesen werden können oder anders ausgedrückt: Gelesene Daten der Transaktion 1 können von anderen Anwendungsprozessen nicht mehr manipuliert werden und sind bereits nach dem Lesezugriff gesperrt. Versucht ein zweiter Anwendungsprozess, die gelesenen Daten zu verändern, erhält der Anwendungsprozess die folgende Fehlermeldung:

```
SQL-Status: 57033
Vendorencode: -913
Nachricht: [SQL0913] Zeile oder Objekt LAGER der Art *FILE in EPSW09 wird verwendet. Ursache . . . . :
Das angeforderte Objekt LAGER der Art *FILE in der Bibliothek EPSW09 wird gerade von einem anderen
Anwendungsprozeß verwendet, oder eine Zeile im Objekt wird von einem anderen Anwendungsprozeß oder
```

Bereits durch den lesenden Zugriff werden die Zeilen für andere Prozesse gesperrt

Im Joblog findet man detaillierte Informationen:



Das Joblog zeigt die Satzsperre

Alle Datenmanipulationen, die den Satz 2 in der entsprechenden Tabelle betreffen, werden unterbunden. Der lesende Zugriff oder Aktualisierungen anderer Zeilen der Tabelle sind problemlos möglich. Natürlich können auch Zeilen eingefügt werden.

Erst die Isolationsstufe **SERIALIZABLE** bietet die vollständige Isolation der Transaktion. Die Isolationsstufe SERIALIZABLE ist die sicherste, aber auch die langsamste Isolationsstufe. Der System i Server realisiert die Isolationsstufe durch Teildateisperren. Aus diesem Grund sollten sie diese Isolationsstufe wirklich nur verwenden, wenn sie die Datei vollständig sperren müssen.

Isolationsstufe	Dirty Read	Non Repeatable Read	Änderung gelesener Zeilen durch anderen Prozess	Einfügen neuer Zeilen
NO COMMIT	Ja	Ja	Ja	Ja
READ UNCOMMITTED	Ja	Ja	Ja	Ja
READ COMMITTED	Nein	Ja	Ja	Ja
REPEATABLE READ	Nein	Nein	Nein	Ja
SERIALIZABLE	Nein	Nein	Nein	Nein

7.10.6.3 Transaktionen identifizieren

Sicherlich wird es von Zeit zu Zeit erforderlich sein, zu prüfen, welcher Job eine Objekt- oder Satzsperre ausgelöst hat. Bekannt sind hier sicherlich die Befehle WRKOBJLCK (Work with Object Lock) und DSPRCDLCK (Display Record Lock).

```
                    Mit Objektsperren arbeiten (WRKOBJLCK)

Auswahl eingeben und Eingabetaste drücken.

Objekt . . . . . > LAGER_____    Name
  Bibliothek . . . . . >   EPSW09____  Name, *LIBL, *CURLIB
Objektart  . . . . . > *FILE__    *ALRTBL, *AUTL, *BNDDIR...
Teildatei  . . . . . .  *NONE_____  Name, *NONE, *FIRST, *ALL
ASP-Einheit  . . . . . .  *_________  Name, *, *SYSBAS
Ausgabe  . . . . . .  *_____    *, *PRINT
```

Der Befehl WRKOBJLCK

Anschließend müssen Sie mit dem Job arbeiten:

```
                         Mit Objektsperren arbeiten
                                                          System:   ITRAINNG
Objekt . . . . :   LAGER                 Typ  . . . . . :   *FILE-PHY
  Bibliothek . :     EPSW09              ASP-Einheit  . :   *SYSBAS

Auswahl eingeben und Eingabetaste drücken.
  4=Job beenden  5=Mit Job arbeiten  8=Mit Jobsperren arbeiten

Aus-
wahl   Job          Benutzer     Sperre    Status          Bereich   Thread
 _     QZDASOINIT   QUSER        *SHRRD     HELD           *JOB
 =     QZDASOINIT   QUSER        *SHRRD     HELD           *JOB
 _                               *SHRRD     HELD           *JOB
 _                               *SHRRD     HELD           *JOB
```

Mit einem Job arbeiten

7.10.6
Seite 10

Und wenn Sie dann im nachfolgenden Bildschirm die Option 12 „Mit Sperren arbeiten, falls aktiv“ verwenden, sind Sie am Ziel und sehen die Datensperren:

```
Auswahl eingeben und Eingabetaste drücken.
  5=Mit Jobsperren für gemeinsam benutzte Teildateien arbeiten
  8=Mit Teildateisperren arbeiten

Ausw  Teildatei   Sperrart  Sperre   Status  Gemeins.  Bereich  Thread
 =      LAGER       MBR     *SHRRD    HELD             *JOB
 _                  MBR     *SHRRD    HELD             *JOB
 _                  MBR     *SHRRD    HELD             *JOB
 _                  DATEN   *SHRRD    HELD             *JOB
 _                  DATEN   *SHRRD    HELD             *JOB
 _                  DATEN   *SHRUPD   HELD             *JOB
```

Mit Sperren arbeiten

Einfacher geht es mit dem Befehl DSPRCDLCK:

```
                    Satzsperren anzeigen (DSPRCDLCK)

  Auswahl eingeben und Eingabetaste drücken.

  Physische Datei  . . . . . > LAGER       Name
    Bibliothek . . . . . . . > EPSW09      Name, *LIBL, *CURLIB
  Teildatei  . . . . . .       *FIRST      Name, *FIRST
  Satznummer . . . . . .       *ALL        1-4294967288, *ALL
  Ausgabe  . . . . . .         *          *, *PRINT
```

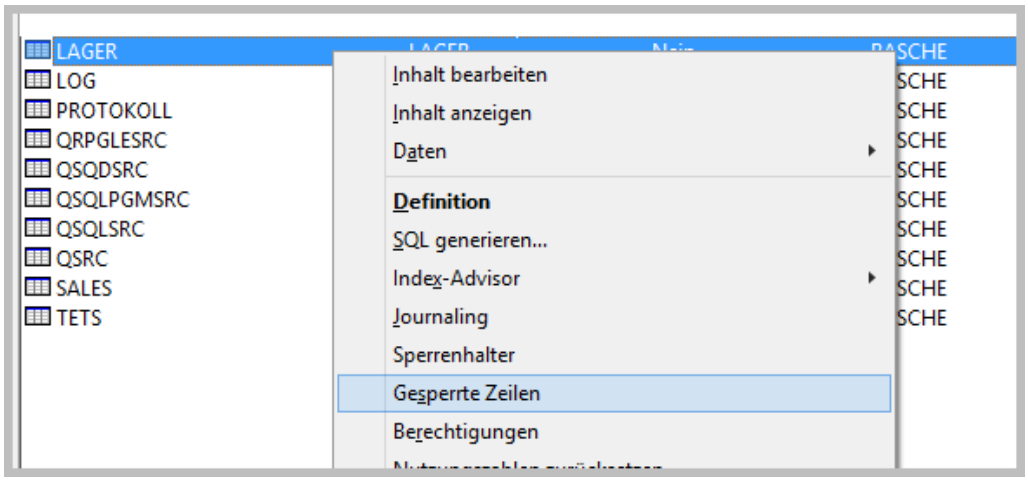
Mit Satzsperren arbeiten

Im nachfolgenden Bildschirm können Sie jetzt analysieren, welche Sätze derzeit für ein Update gesperrt sind:

```
_                      Teildateisatzsperren anzeigen
                                                        System:   ITRAINNG
 Datei  . . . . . :   LAGER          Teildatei  . . . . . :   LAGER
   Bibliothek . . . :     EPSW09

     Satznummer  Job         Benutzer    Nummer  Status    Sperrart
              2  QZDASOINIT  QUSER       730236  ANGEHALT  LESEN
              3  QZDASOINIT  QUSER       730236  ANGEHALT  LESEN
              4  QZDASOINIT  QUSER       730236  ANGEHALT  LESEN
             13  QZDASOINIT  QUSER       730236  ANGEHALT  LESEN
             15  QZDASOINIT  QUSER       730236  ANGEHALT  LESEN
```

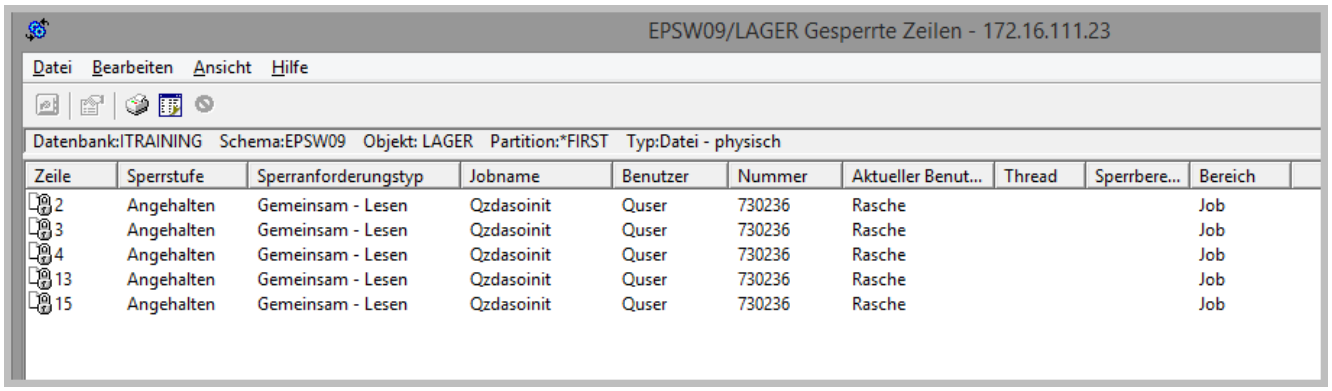
Satzsperren der Datei EPSW09/LAGER

Natürlich können Sie diese Informationen auch im System i Navigator finden:



Satzsperren im System i Navigator

Wie so oft, finde ich die Informationen im grafischen Interface sehr viel übersichtlicher dargestellt:



Zeile	Sperrstufe	Sperranforderungstyp	Jobname	Benutzer	Nummer	Aktueller Benut...	Thread	Sperrbere...	Bereich
2	Angehalten	Gemeinsam - Lesen	Qzdasoinit	Quser	730236	Rasche			Job
3	Angehalten	Gemeinsam - Lesen	Qzdasoinit	Quser	730236	Rasche			Job
4	Angehalten	Gemeinsam - Lesen	Qzdasoinit	Quser	730236	Rasche			Job
13	Angehalten	Gemeinsam - Lesen	Qzdasoinit	Quser	730236	Rasche			Job
15	Angehalten	Gemeinsam - Lesen	Qzdasoinit	Quser	730236	Rasche			Job

Informationen über die Satzsperren einer Tabelle/Datei

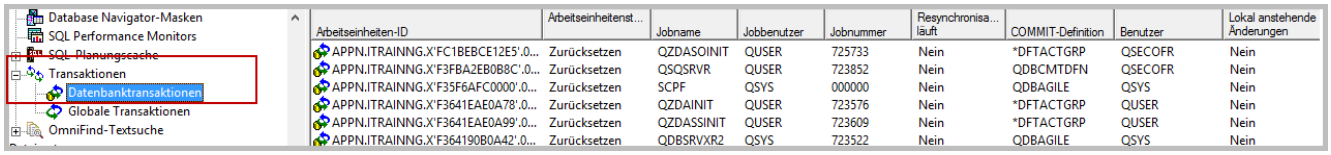
7.10.6
Seite 12

Abschließend bleibt noch die Frage zu klären, ob wir auch eine Übersicht aller bestehenden Transaktionen erzeugen können. Der Befehl WRKCMTDFN (Work with Commit Definition) liefert folgende Übersicht:

```
                                Job auswählen                          ITRAINNG
                                                          14.02.15  00:03:23
Auswahl eingeben und Eingabetaste drücken.
  1=Auswählen

                                                                      Im
                                                                    System
Ausw.  Job         Benutzer    Nummer  Art      -----Status-----  eingegang.
  _    QZDASOINIT  QUSER       730248  PJ       ACTIVE             13.02.15
  _    QZDASOINIT  QUSER       730247  PJ       ACTIVE             13.02.15
  _    QZDASOINIT  QUSER       730246  PJ       ACTIVE             13.02.15
  _    QZDASOINIT  QUSER       730245  PJ       ACTIVE             13.02.15
  _    QZDASOINIT  QUSER       730244  PJ       ACTIVE             13.02.15
  _    QZDASOINIT  QUSER       730243  PJ       ACTIVE             13.02.15
```

Mit Commit-Definitionen arbeiten

Auch hier wieder der direkte Vergleich mit dem System i Navigator:



System i Navigator – Datenbanktransaktionen

Schauen Sie sich die Informationen in den unterschiedlichen Oberflächen in Ruhe an und entscheiden Sie, welche Oberfläche in Ihrem Umfeld wirklich geeignet ist.