Rheinische Friedrich-Wilhelms-Universität Bonn, Institut für Informatik III

Seminar „Intelligente Datenbanken“, Sommersemester 2005

Prof. Dr. R. Manthey
Andreas Behrend

SQL 2003 – Neue Konzepte

Gereon Schüller
14. Juni 2005

Inhalt

1 Einleitung

Mit SQL 2003 ist der bereits vier Jahre zuvor stark erweiterte Standard nochmals um viele Funktionen ergänzt worden.
Diese Ausarbeitung wird die neuen Funktionalitäten vorstellen, die mit Verabschiedung zu SQL hinzugefügt worden sind. Wer mit SQL vertraut ist, wird feststellen, dass SQL dem Prinzip treu geblieben ist, möglich „lesbar“ und an die natürliche Sprache angelehnt zu sein. Die Neuerungen werden hauptsächlich an kleinen konkreten Beispielen und Listings gezeigt, wobei an einigen Stellen zum Verständnis auch eine Syntaxübersicht gegeben wird.
Diese Ausarbeitung wird wohl nicht als Referenz geeignet sein, dafür reicht auch der Umfang nicht, dennoch wird der interessierte Leser mit den genannten Schlüsselwörtern sicherlich recht schnell detailliertere Informationen zu den einzelnen Konzepten finden können. Bereits an dieser Stelle sei hier besonders die Quelle [iAnywhere] empfohlen.

Bonn, im Juni 2005

2 Übersicht der SQL-Komponenten und deren Neuerungen in SQL 2003

SQL 2003 enthält zahlreiche Neuerungen, jedoch sind nur einige konzeptionell, sie werden im Folgenden näher erläutert und sind hier fettgedruckt dargestellt. Die „fehlenden“ Parts der ANSI-Norm wurden aus historischen Gründen nicht mehr belegt. Part 14 ist völlig neu hinzugekommen und wird daher auch besonders ausführlich behandelt werden.

1	Framework	Änderungen aufgrund der anderen Parts
2	Foundation	**BIGINT, Multimengen, Generierte Spalten, Sequenzgeneratoren und Identitätspalten, Merge, Funktionen, generierte Tabellen, Tabellensamples**
3	Call Level Interface	-
4	Persistent Stored Modules	CASE-Anweisung, Variablenbenennung
9	Management of External Data	Update-in-place, Projektion externer Daten
10	Object Language Binding	JDBC 3.0, Array-Typen, Multimengenunterstützung
11	Information and Definition Schema	**Sequenzgeneratoren**, neue techn. Views
13	SQL Routines and Type Using Java	**Tabellenfunktionen**, DATALINK-Ergebnisse
14	XML-related Specifications	**Völlig neuer Part**

3 Trivia: Neue Datentypen und Funktionen

3.1 Neue Datentypen in SQL 2003

3.1.1 BIGINT

Als neuer primitiver Datentyp ist der Typ „BIGINT“ hinzugekommen. Er ist –wie der Name vermuten lässt- ein Ganzzahlwert, der eine Länge von 8 Byte und mit Vorzeichen oder ohne verwendet werden kann. Sein Wertebereicht beträgt daher entweder -2^{63} bis $2^{63}-1$ oder 0 bis 2^{64}

3.1.2 XML, MULTISET

Um die neuen Konzepte von Part 14 und das der Multimenge zu nutzen, wurden diese beiden Datentypen ins Leben gerufen. Sie werden im entsprechenden Abschnitt erläutert.

3.2 Neue Funktionen in SQL 2003

Als neue Skalarfunktionen sind in SQL 2003 die Funktionen `LN, EXP, POWER, SQRT, FLOOR und CEIL[ING]` hinzugekommen. Sie erwarten jeweils einen numerischen Wert, die Funktion `POWER` erwartet hingegen zwei. Bereits die Namen lassen ihre Wirkungsweise vermuten und werden hier der Vollständigkeit halber aufgeführt. Sie dürften ohnehin in den meisten Implementierungen bereits enthalten gewesen sein.
Völlig neu –wenn auch durch vorhandene Funktionen zu ersetzen- sind Funktionen, um Regressionen zwischen Datenreihen zu berechnen. Wir wollen auf sie nicht weiter eingehen, verweisen jedoch auf [2], die ihre Funktionsweise beschreiben.

4 Identitätsspalten, Sequenzgeneratoren und generierte Spalten

4.1 Identitätsspalten

Mit Sequenzgeneratoren wird ein neues Konzept in SQL eingeführt, das sicherlich in den meisten DBMS schon vorhanden gewesen sein dürfte, wenngleich auch nicht zwingend so gut kontrollierbar.
Bei einer Sequenz handelt es sich um einen Zahlwert, welcher von hoch- oder heruntergezählt wird.
Zwei Sequenzgeneratoren sind möglich:

- Interne Sequenzgeneratoren per Identitätsspalte
- Externe Sequenzgeneratoren.

Eine Identitätsspalte nennt z.B. Microsoft Access bis dato „AutoWert“. Unter SQL2003 kann man eine Tabelle mit Vorlesungstitel und einer Vorlesungsnummer, beginnend mit 1000 und im 10-er-Intervall wie folgt definieren:

```
CREATE TABLE Vorlesungen(
     Vorlesungs_ID INTEGER GENERATED ALWAYS AS IDENTITY
               START WITH 1000
               INCREMENT 10
               MINVALUE 100
               NO MAXVALUE
               NO CYCLE,
     Titel VARCHAR(50))
```

Für jede Tabelle kann eine Identitätsspalte spezifiziert werden. Gestartet wird der Wertebereicht immer mit dem Argument von `START WITH`, bei jeder Operation wird um den Wert von `INCREMENT` hochgezählt, bis `MAXVALUE` erreicht ist. Ist `CYCLE` angegeben, so wird nach dem Maximalwert wieder von vorne angefangen, ansonsten tritt eine Ausnahme auf.
`MINVALUE` stellt den kleinsten Wert sicher, der auftreten darf. Bei den Werten darf auch jeweils `NO` angegeben werden, dann wird ein DBMS-Interner Wert benutzt.
Im Gegensatz zu einigen heutigen Implementierungen darf der Wert einer Identitätsspalte auch manuell gesetzt werden.
Die Auswirkungen der obigen Definition sollen noch verdeutlicht werden. Fügen wir also ein:

```
INSERT INTO Vorlesungen (Titel) VALUES ('Informationsysteme')
INSERT INTO Vorlesungen (Titel) VALUES ('Computergrafik')
```

So erhalten wir folgende Tabelle:

Vorlesungs_ID	**Titel**
1000	Informationssysteme
1010	Computergrafik

4.2 Sequenzgeneratoren

Was aber, wenn wir tabellenübergreifend hochzählen möchten? Dazu stellt uns SQL das neue Datenbankobjekt SEQUENCE zur Verfügung. Wir können es wie folgt erstellen:

```
CREATE SEQUENCE Vorgang_Seq AS INTEGER
     START WITH 1
     INCREMENT 1
     NO MINVALUE
     MAXVALUE 1000
     CYCLE
```

Der Sequenzgenerator soll also mit dem Wert 1 starten, immer um 1 hochzählen, solange bis 1000 erreicht ist und dann wieder bei 1 anfangen.
Haben wir nun eine Datenbank, in der Bestellungen und Reklamationen gespeichert sind, und möchten beiden eine Vorgangsnummer zuweisen, so erhalten wir den nächsten Sequenzwert mittels `NEXT VALUE FOR`:

```
INSERT INTO Bestellung(Ware, Preis, Vorgang_ID) VALUES
     ('Turingmaschinen', 13.54, NEXT VALUE FOR Vorgang_Seq)
INSERT INTO Reklamation(Ware, Defekt, Vorgang_ID) VALUES
     ('Endlicher Automat', 'terminiert nicht', NEXT VALUE FOR
     Vorgang_Seq)
```

Beim ersten Einfügen erhält man dann den Wert ‚1', beim zweiten ‚2' als Vorgangsnummer.
Natürlich kann auf `SEQUENCE` auch `ALTER` und `DROP` angewendet werden.

4.3 Generierte Spalten

Hat man eine Spalte, die nur aus anderen Spalten berechnet wird, so ist es (und so wurde es bisher auch stets gelehrt) unnütz, diese in der Tabellenrelation selbst zu speichern.
Es reicht auch vollkommen aus, generierte Werte per Abfrage zu gewinnen, z.B. den Bruttopreis eines Artikels aus dem Nettopreis durch Aufschlagen der Mehrwertsteuer:

```
SELECT ArtikelName, NettoPreis, Nettopreis * 1.16 AS
     BruttoPreis FROM Artikel
```

Braucht man diese Werte jedoch häufiger, so stellt sich die Frage, warum der Wert immer wieder neu berechnet werden soll. Eine Abspeicherung kommt aber wegen möglicher Änderungen auch nicht in Frage.
Kann man nicht beides dem DBMS mitteilen? Genau dies ist mit den generierten Spalten möglich:

```
CREATE TABLE Artikel(
     ArtikelName AS VARCHAR(50),
     NettoPreis DECIMAL(7,2),
     BruttoPreis ALWAYS GENERATED AS (NettoPreis * 1.16))
```

Das DBMS erhält damit die Möglichkeit, Werte genau zum richtigen Zeitpunkt zu ändern und Werte ggf. zwischenzuspeichern.
Der SQL-Standard macht dabei jedoch die Einschränkung, dass die Werte nur aus Spalten der selben Zeile (und somit der selben Relation) berechnet sein dürfen.

5 Automatische Tabellenschema-Erstellung

Bisher sah die DDL von SQL nur vollständig manuelle Erstellung von Tabellen vor. Oft kommt es jedoch vor, dass bestehende Tabellenstrukturen in neue Tabellen übernommen werden sollen oder dass Abfrageergebnisse dauerhaft gespeichert werden müssen, um die Daten später auszuwerten oder weiterzugeben. Hier bietet SQL 2003 mit zwei neuen Schlüsselwörtern Abhilfe

5.1 Der „LIKE“-Operator

Erstellt eine neue Tabelle aufgrund eines existierenden Schemas. Stellen wir uns folgende Schema vor:

```
CREATE TABLE Fahrzeuge (
     Nummer INTEGER GENERATED ALWAYS AS IDENTITY,
     Raeder INTEGER,
     Hersteller CHAR(50),
     Preis DECIMAL(7,2),
     Farbe CHAR(20),
);
```

Möchten wir nun eine neue Tabelle für Kraftfahrzeuge anlegen, so fehlt uns z.B. noch die Motorleistung. Wir können diese Tabelle dann wie folgt erstellen:

```
CREATE TABLE Kraftfahrzeuge (
     LIKE Fahrzeuge INCLUDING IDENTITY WITH NO DATA,
     Motorleistung INTEGER
);
```

Mittels `[INCLUDING|EXCLUDING] [IDENTITY|DEFAULTS|GENERATED]` können Identitäten, Defaultwerte und generierte Spalten ebenfalls übernommen werden.
`WITH [NO] DATA` gibt an, ob die Daten ebenfalls übernommen werden sollen.

5.2 Der „AS“-Operator

Erstellt eine neue Tabelle aufgrund einer Abfrage. Um aus einer vorhandenen Tabelle von Personen eine Tabelle „Professoren“ zu erstellen, die nur ProfessorInnen enthält, kann die folgende Anweisung verwendet werden:

```
CREATE TABLE Professoren (
     Name,
     Vorname)
AS
(SELECT Name,
     Vorname
FROM Personen
WHERE Personen.Beruf LIKE 'Professor%');
```

Zwar müssen die Spaltennamen explizit angegeben werden, der Spaltentyp wird allerdings automatisch übernommen. Die Spaltenzahl muss mit der in der Abfrage übereinstimmen. Auch hier sind die oben erwähnten Modifikatoren zugelassen.

Man muss bei beiden Befehlen allerdings bedenken, dass keine Anhängigkeit zu den ursprünglichen Daten besteht, so dass `CREATE TABLE LIKE` **keine** wirkliche Vererbung darstellt, sondern eher ein Hilfsmittel für den Datenbankentwickler.

6 Die dritte Dimension: Multimengen

6.1 Probleme mit Arrays in SQL 1999

In SQL 1999 wurde das Konzept von Arrays eingeführt. Somit war es möglich, z.B. folgende Tabelle zu erstellen:

```
CREATE TABLE Studenten (
     Name VARCHAR(15),
     …
     Sprachen VARCHAR(15) ARRAY[8]);
```

Hier könnten also objektorientiert jedem Studenten 8 Sprachen in einem 8-elementigen Array aus Strings zugewiesen werden.
Wie von Programmiersprachen bekannt, können Arrayelemente direkt angeprochen werden, ganze Arrays sind untereinander vergleichbar. Doch was passiert, wenn ein Student mehr als 8 Sprachen spricht, und wozu in jedem Datensatz 8 Plätze reservieren? Hier offenbart sich ein Nachteil des objektrelationalen Modells von SQL1999. SQL2003 bietet uns hierfür die Multimenge.

6.2 Multimengen

Zuerst eine kurze Definition, was (zumindest SQL) unter Multimengen versteht:
Eine Multimenge ist eine Sammlung gleichgetypter Objekte, ähnlich wie eine Menge, ohne bestimmte Ordnung, jedoch mit Duplikaten. So ist z.B.

$$\{1,2,3,4\} \neq \{1,2,2,3,3,4\}$$

da die Kardinalität nicht übereinstimmt.

6.3 Definition von Multimengen in DDL

In DDL lässt sich von jedem Datentyp durch den Modifikator MULTISET eine Multimenge erstellen. Die Studententabelle kann nun wie folgt definiert werden:

```
CREATE TABLE Studenten (
     Name VARCHAR(15),
     …
     Sprachen VARCHAR(15) MULTISET);
```

6.3.1 Erstellung per Aufzählung

Mittels der Funktion `MULTISET(Element`$_1$`, Element`$_2$`, …, Element`$_n$`)` lässt sich eine Multimenge durch Aufzählung erstellen (wobei auch eine leere Klammer erlaubt ist). Um dem Studenten „Henry“ die Spachen Englisch, Deutsch und Französisch zuzuweisen, lautet die `UPDATE`-Anweisung:

```
UPDATE Studenten
SET Sprachen = MULTISET('ENGLISCH','DEUTSCH','FRANZÖSISCH')
WHERE Name='Henry';
```

6.3.2 Erstellung aus Abfragen

Auch aus Abfragen lassen sich Multimengen erstellen. Sofern die Abfrage nur eine Spalte zurückgibt, wird eine Multimenge vom Typ dieser Spalte erstellt, andernfalls ist jedes Element der Multimenge wiederrum vom Typ `ROW(TYPE(Spalte`$_1$`),…,TYPE(Spalte`$_n$`)`[1].
Falls die Sprachen des Studenten Henry schon in einer Tabelle `Schueler` standen:

```
UPDATE Studenten
SET Sprachen = MULTISET(SELECT Sprachen FROM Schueler WHERE
Name='Henry') WHERE Name='Henry'
```

[1] Eine `ROW` fasst mehrere Objekte unterschiedlichen Typs zusammen, ähnlich wie eine Tabellenzeile oder ein Tupel in der Mathematik

6.4 Operationen auf Multimengen

SQL2003 bietet verschiedene Operatoren auf Multimengen an, die zumeist der Mengenalgebra entlehnt sind:

`CARDINALITY(M)`	`|M|`	Liefert die Kardinalität einer Multimenge (Duplikate mitgezählt)
`SET(M)`	`{M}`	Entfernt die Duplikate aus einer Multimenge
x `[NOT] MEMBER OF` M	`x∈M`	Prüft, ob eine Element in einer Multimenge enthalten ist
M `SUBMULTISET` N	`M ⊆ N`	Prüft, ob die Multimenge M in N enthalten ist, wobei jedes Element in der Obermenge auch oft genug enthalten sein muss
`IS A SET`		Prüft, ob die Multimenge duplikatfrei ist
M `MULTISET UNION` N	`M∪N`	Vereinigung zweier Multimengen
M `MULTISET EXCEPT` N	`M\N`	Bildet die Differenz zwei Multimengen
M `MULTISET INTERSECT` N	`M∩N`	Bildet den Schnitt zweier Multimengen
`ELEMENT(`M`)`		Liefert das einzige Element einer einelementigen Multimenge.

Bei `UNION`, `EXCEPT`, `INTERSECT` kann ferner `DISTINCT` oder `ALL` angegeben werden, was zu einer Duplikateliminierung führt oder nicht. Der Defaultwert ist `DISTINCT`.

6.4.1 Spaltenfunktionen

Ferner ist es möglich, aus den Spalten einer Tabelle Multimengen zu generieren. Folgende Funktionen stehen dabei zur Verfügung:

`COLLECT(`Spalte`)`	Erzeugt eine Multimenge aus einer Spalte
`FUSION(`Multimengen-Spalte`)`	Vereinigt die Multimengen einer Spalte zu einer neuen
`INTERSECTION(`Multimengen-Spalte`)`	Bildet den Schnitt der Multimengen einer Spalte

6.4.2 UNNEST

Die Funktion UNNEST(Multimenge) bildet eine Multimenge auf eine Tabelle ab, so dass diese als Tabellenreferenz gebraucht werden kann, wie im folgenden Beispiel die Multimenge {4,7,11}. Es sei dabei noch auf die explizite Benennung der resultierenden Tabelle hingewiesen, die nur die Spalte „DM“ enthält:

```
SELECT Preise.DM, Preise.DM/1.95583 AS Preise.EUR
FROM UNNEST(MULTISET(4,7,11)) AS Preise(DM);
```

7 Tabellenwertige Funktionen

7.1 Was verstehen wir unter tabellenwertigen Funktionen?

„Eine tabellenwertige Funktion ist eine Abbildung von beliebigen Werten auf eine Multimenge von Zeilen (`ROW`s).“[2]

Der Leser wird sich vielleicht an die Definition einer Tabelle in relationalen Datenbanksystemen erinnern, und genauso ist es: Eine tabellenwertige Funktion liefert im Endeffekt eine (virtuelle) Tabelle, ggf. basierend auf Eingabeparametern.

[2] “For specification purposes in the standard, the return type is equivalent to a multiset of rows (*i.e.*, a `MULTISET` type whose element type is a `ROW` type) and not a real table, but it can be queried like a table“, Eisenberg [1], p. 121

Trotzdem unterscheidet der SQL-Standard hier, um Verwechslungen mit physischen Tabellen zu vermeiden.
Überall dort, wo in Anfragen Tabellen erwartet werden, kann stattdessen auch eine tabellenwertige Funktion eingesetzt werden.

7.2 Interne SQL-Funktionen

Den einfachsten Fall bilden hier die internen SQL-Funktionen. Angenommen, wir möchten alle Studenten mit Matrikelnummer, Name und Vorname eines bestimmten Fachsemesters als Tabelle erhalten, dann können wir uns hierfür selbstverständlich eine Abfrage definieren. Brauchen wir diese Abfrage jedoch häufig und mit wechselnden Semesterzahlen, so stellt eine tabellenwertige Funktion eine sinnvolle Lösung dar:

```
CREATE FUNCTION StudentenImFachsemester(Semesterzahl INTEGER)
     RETURNS TABLE (
          MatrNr INTEGER,
          Name VARCHAR(15),
          Vorname VARCHAR(15))
     LANGUAGE SQL
     READS SQL DATA
     DETERMINISTIC
     RETURN TABLE(
          SELECT Matrikelnummer, Name, Vorname
          FROM Studenten
          WHERE Studenten.Fachsemester=Semesterzahl);
```

Zuerst werden also der Funktionsname festgelegt, dann die Eingabeparameter und sodann die Ausgabeparamter. Es folgen einige Zusicherungen (die unten besprochen werden) und dann die eigentliche Anfrage, in der die Eingabeparameter wieder verwendet werden können.
Möchte man dann die Matrikelnummern der Studenten im 6. Fachsemester, die mit „S“ anfangen, abfragen, so können wir schreiben:

```
SELECT MatrNr
FROM StudentenImFachsemester(6)
WHERE Name LIKE 'S%';
```

7.3 Externe Funktionen in anderen Programmiersprachen

Datenbanken sind auf Eingaben aus der Außenwelt angewiesen. Sofern diese nicht vom Benutzer per Hand erfolgen, müssen sie aus anderen externen Quellen übernommen werden. Wie aber kann man Messwerte etc. effektiv in Datenbanken halten, insbesondere dann, wenn sich diese beständig ändern?
Eine Möglichkeit besteht sicherlich darin, externe Programme aufzurufen, die beständig Daten in Tabellen schreiben, wie dies bei vielen DBMS möglich ist, doch wäre es einfacher, die Tabellen würden automatisch von externen Programmen dann dynamisch erstellt, wenn sie gebraucht werden. Genau dies können Tabellenfunktionen bewerkstelligen, wenn sie externe Funktionen aufrufen.
Nehmen wir an, wir haben ein Kraftwerk mit mehreren Blöcken, dessen Status unsere Datenbank überwachen soll. Wir definieren uns dazu folgende Funktion:

```
CREATE FUNCTION BlockStatus()
     RETURNS TABLE (
          Block_Nummer BYTE,
          Leistung_KW INTEGER,
          Temperatur_C DECIMAL(3,2),
          FehlerCode INTEGER)
     NOT DETERMINISTIC
```

```
    NO SQL
    LANGUAGE JAVA
    EXTERNAL
    PARAMETER STYLE SQL;
```

Beim Referenzieren der Funktion liefert diese uns eine Relation mit Block_Nummer, Leistung_KW, Temperatur_C und Fehlercode als Spalten.
Das DBMS muss dabei selbst dafür sorgen, dass die externe Funktion ordnungsgemäß aufgerufen wird, denn SQL definiert hier nur den Funktionsrumpf.

7.4 Zusicherungen von Funktionen

Neben den Parametern bilden die Zusicherungen einen weiteren wichtigen Bestandteil des Kontrakts zwischen Anwender und Programmierer. In SQL sind dazu folgende Angaben möglich:

- → `[NOT] DETERMINISTIC`: Die Funktion arbeitet entweder deterministisch, d.h. in diesem Falle liefert sie für gleiche Parameter *bei gleichem Datenbankzustand* die gleiche Ausgabe. Da im zweiten Beispiel die Ausgabe aber von dem Zustand des Kraftwerks abhängt, arbeitet diese Funktion nicht-deterministisch
- → `LANGUAGE {ADA|C|COBOL|FORTRAN|MUMPS|PASCAL|PLI|SQL|JAVA}` gibt die Sprache an, in der die externe Funktion geschrieben wurde
- → `PARAMETER STYLE {SQL|GENERAL|JAVA}` gibt an, welchen Parameterstil die externe Funktion erwartet. Wir möchten an dieser Stelle nicht weiter darauf eingehen.
- → `NO SQL|CONTAINS SQL|READS SQL DATA|MODIFIES SQL DATA` gibt an, ob die Funktion keine SQL-Anweisungen enthält, zwar SQL-Anweisungen enthält, aber keine Daten liest, Daten liest oder welche verändert
- → `RETURNS NULL ON NULL INPUT|CALLED ON NULL INPUT` gibt an, ob die Funktion mit `NULL` als Argument `NULL` zurückgibt oder trotzdem ausgeführt wird.

Die externen Funktionen stellen eine wichtige Erweiterung von SQL in Richtung aktive Datenbanken dar. Der Standard stellt hier nur die Schnittstellendefinition dar, den Rest übernimmt die jeweilige Implementierung. Einige größere DBMS wie ORACLE und DB/2 unterstützten schon vorher solche Funktionen, aus denen u.a. die Konstrukte übernommen wurden.

8 Fensterfunktionen

8.1 Funktionsweise von Fensterfunktionen

Fensterfunktion legen ein „Fenster“ über eine Tabelle und verarbeiten dessen Inhalt, d.h. sie operieren auf bestimmten Spalten.
Neu in SQL 2003 sind hier die Funktion `RANK(), DENSE_RANK(), ROW_NUMBER(), CUME_DIST() [=PERCENT_RANK()]`
Die Funktion `RANK` gibt dabei den Rang an, den der jeweilige Datensatz in der Tabelle bezüglich der Spalte „im Fenster“ einnimmt. Gleichstände werden dabei zugunsten eines höheren Wertes entschieden, bei `DENSE_RANK` zugunsten eines niedrigeren Ranges. `ROW_NUMBER` hingegen wählt zufällig einen der in Frage kommenden Ränge aus. `CUME_DIST` ordnet eine Zahl zwischen 0 und 1 entsprechenden der relativen Position zu.
Das Fenster selbst wird mittels des Schlüsselwortes `OVER` und dem Konstrukt `ORDER BY...[DESC]` ausgewählt.
Haben wir beispielsweise eine Tabelle der 2. Bundesliga mit Vereinsnamen und Punkten, dann könnten wir das Fenster über die Spalte `Punkte` legen und die Funktion wie folgt anwenden:

```
SELECT Verein, Punkte,
     DENSE_RANK() OVER (ORDER BY Punkte DESC) AS myDENSE_RANK,
     RANK() OVER (ORDER BY Punkte DESC) AS myRANK,
     ROW_NUMBER() OVER (ORDER BY Punkte DESC) AS myROW_NUMBER,
     CUME_DIST() OVER (ORDER BY Punkte DESC) AS myCUME_DIST
FROM ZweiteLiga;
```

Verein	Punkte	myDENSE_RANK	myRANK	myROW_NUMBER	myCUME_DIST
1. FC Köln	58	1	1	1	1
MSV Duisburg	56	2	2	2	0,66
Eintracht Frankfurt	52	3	4	4	0
TSV 1860 München	52	3	4	3	0

9 Stichproben aus Tabellen

9.1 Probleme bei der Auswertung großer Relationen

Datenbanken sind hervorragende Grundlagen zur Erhebung von Statistiken. Mittels der Aggregatfunktionen wie `AVG()` könnte man zum Beispiel große Mengen an Messwerten etc. mitteln. Doch stößt man bei sehr vielen Datensätzen an die Grenzen von Datenbanksystemen. In der Statistik ist es üblich, mit Stichproben aus Datenmengen zu arbeiten.
Diese Funktionalität stellt SQL jetzt mit dem Modifikator `TABLESAMPLE {SYSTEM|BERNOULLI}(p) [REPEATABLE (s)]` bereit.
Es werden jeweils p% der Datensätze ausgewählt, doch stehen zwei Möglichkeiten zur Auswahl zur Verfügung:

- BERNOULLI: Jede Zeile wird mit einer Wahrscheinlichkeit von p/100 ausgewählt und mit 1-(p/100) verworfen.
- SYSTEM: Die Auswahl wird hier nicht zwangsläufig unabhängig voneinander ausgewählt

REPEATABLE(s) schließlich gibt einen beliebigen Startwert für den Zufallszahlengenerator vor, um reproduzierbare Ergebnisse erhalten zu können.
Wir könnten also das Durchschnittsgehalt der Bundesbürger in den Bundesländern berechnen (so wir denn die entsprechenden Datensätze hätten):

```
SELECT land, AVG(Gehalt)
FROM bundesbuerger TABLESAMPLE BERNOULLI(1) REPEATABLE(17)
GROUP BY land
```

Wir würden also nur 1% der Datensätze berücksichtigen, und zwar bei jedem Aufruf die selben.

10 Merging

10.1 Ein Drahtseilakt – UPDATE oder INSERT?

Möchte man Tabellen zusammenfügen, dann kann zwischen folgenden drei Fällen unterschieden werden:

1. Die Datensätze sind disjunkt, d.h. die Tabellen müssen nur zusammengefügt werden.
2. Die Datensätze stimmen alle in ihren Schlüsselwerten überein, die gewünschten Spalten müssen nur per `UPDATE` aktualisiert werden.
3. Eine Mischung aus den beiden vorigen Fällen, bzw. die Gegebenheiten sind unbekannt

Gerade der dritte Fall ist natürlich der schwierigste und vielleicht auch der häufigste.

Stellen wir uns folgendes Szenario vor: Ein Händler hat einen Warenbestand und erhält Lieferungen mitsamt Lieferschein als Tabelle. Alle Waren seien durch eine Artikelnummer identifiziert. Es können nun also auftreten:

1. Eine Menge des Artikels ist noch im Lager vorhanden, die Anzahl der Artikel aus der Lieferung ist also zu den vorhandenen aufaddieren
2. Ein Artikel ist neu im Sortiment und muss als neuer Datensatz eingefügt werden.

Die bisher benutzte Möglichkeit bestünde also darin, erst alle schon vorhandenen Artikel zu aktualisieren und dann alle neuen Waren per `INSERT` einzufügen. Damit aber in der Zwischenzeit keine Artikel gelöscht werden können, ist das ganze als Transaktion zu implementieren.

10.2 Eine mögliche Lösung

Da diese Aufgabe quasi als Standardaufgabe vorkommt, spezifiziert SQL 2003 eine neue Anweisung der DML, nämlich das MERGE-Statement, dass ähnlich einer IF-Anweisung in Programmiersprachen arbeitet:

```
MERGE INTO Ziel USING Quelle
ON Bedingung
WHEN MATCHED THEN Update
WHEN NOT MATCHED THEN Insert
```

Beispiel für das oben erwähnte Szenario:

```
MERGE INTO Lager
USING (SELECT ArtikelNummer, Bezeichnung, Anzahl FROM
     Lieferung)
ON (Lager.ArtikelNummer = Lieferung.ArtikelNummer)
WHEN MATCHED THEN UPDATE SET
     Beschreibung=Lieferung.Beschreibung,
     Anzahl=Lager.Anzahl + Lieferung.Anzahl
WHEN NOT MATCHED THEN
     INSERT (ArtikelNummer, Bezeichnung, Anzahl) VALUES
     (Lieferung.ArtikelNummer, Bezeichnung, Anzahl)
```

11 SQL/XML

XML hat sich in den letzten Jahren zu dem Datenaustauschformat schlechthin entwickelt. Verschiedene Faktoren haben dazu beigetragen, XML „hip“ zu machen. Ähnlich wie kurze Zeit zuvor mit Java geschehen, so bindet SQL nun auch XML als Bestandteil ein. Aufgrund seiner Bedeutung wurde diesem Thema der Part 14 des neuen Standards gewidmet.

11.1 Erinnerung/Erklärung XML

Obwohl sicherlich vielen bereits geläufig, so soll an dieser Stelle noch einmal eine kurze Erläuterung zu XML stehen, insbesondere zu den verwendeten Bezeichnungen.
Sehen wir uns ein XML-Dokument an:

```
<?xml version='1.0' encoding='UTF-8'?>
<Studenten>
     <Student MatrikelNummer="185934">
          <Vorname>Eliza</Vorname>
          <Name>Day</Name>
          <Anschriften>
               <Anschrift>Rosengasse 7, 53117 Bonn</Anschrift>
          </Anschriften>
     </Student>
     <Student MatrikelNummer="193429">
```

```
          <Vorname>Hans</Vorname>
          <Name>Gl&uuml;ck</Name>
          <Anschriften>
               <Anschrift>Goldgasse 9, 51345 Much</Anschrift>
          </Anschriften>
     </Student>
</Studenten>
<!—Hier k&oouml;nnte die Datei enden-->
```

Es handelt sich bei einem XML-Dokument um einen String, der aus folgendem besteht:

1. Einem Vorspann, hier in der 1. Zeile, in dem die XML-Version und die Zeichenkodierung notiert wird
2. Tags, d.h. Zeichenfolgen in spitzen Klammern, die ggf. eines oder mehrere durch Leerzeichen getrennte Attribute der Form `Attributname="Attributwert"` enthalten.
3. Elementinhalten, die zwischen einem öffnenden Tag (ohne Schrägstrich) und einem schließenden Tag (mit Schrägstrich) stehen
4. Entitätsreferenzen der Form `&referenz;` die auf vordefinierte Zeichenfolgen (auch Sonderzeichen) verweisen. Einige, wie die aus HTML bekannten Umlaute, sind vordefiniert
5. Kommentaren der Form `<!--String-->`, die nicht behandelt werden und ihrerseits beliebige Strings enthalten dürfen

Ferner gilt folgendes:

- Tags dürfen als Inhalte weitere Tags enthalten, jedoch müssen die Tags wohlgeformt sein, d.h. die Elemente müssen (wie Klammernpaare) vor ihrem übergeordneten Element wieder geschlossen sein.
- Leere Elemente dürfen auch in der Form `<Element/>` geschrieben sein
- Sog. „Whitespace“, das sind mehrfache Leerzeichen oder Zeilenumbrüche, wird üblicherweise nicht beachtet
- Welche Elemente welche Unterelemente enthalten dürfen, wird in einer speziellen XML-Datei, dem sogenannten „Schema“ festgelegt.
- Ein XML-Dokument muss genau ein oberstes Element haben, XML-Inhalte (content) dürfen auch mehrere parallele Elemente enthalten (aus dem Baum wird ein Wald).

11.2 XML als Datentyp

Beim Datentyp XML handelt es sich zunächst um einen ganz normalen Datentyp, der überall dort eingesetzt werden kann, wo die herkömmlichen Datentypen wie `INTEGER` eingesetzt werden können. Es sei aber dringend darauf hingewiesen, dass der XML-Datentyp keinen String repräsentiert, sondern einen (internen) Baum, auf dem operiert werden kann.
Der XML-Datentyp kann dabei sowohl XML-Dokumente als auch XML-Inhalte beinhalten. Ob der Inhalt ein Dokument ist, kann mit dem Prädikat IS DOCUMENT festgestellt werden. Um beispielsweise nur die Zeilen einer Tabelle zurückzugeben, in der die Spalte myXML ein Dokument enthält, wäre folgende Abfrage möglich:

```
SELECT * FROM Tabelle WHERE myXML IS DOCUMENT
```

11.3 Anfragen mit XML: XQuery/XMLGEN

Mit XQuery hat das W3C einen Vorschlag zu einer Anfragsprache herausgegeben, die auf XML-Daten operiert. Für eine genauere Beschreibung dieser Sprache sei auf [5] verwiesen. In ihrer derzeitigen Version stellt dieses Sprache einfache Ausdrücke zur Selektion und Projektion bereit. So lassen sich beispielsweise die Vornamen aller Studenten mit einer Matrikel-Nummer kleiner 190.000 aus dem obigen Dokument wie folgt zurückgeben:

```
for $b in document(„studenten.xml“)/Studenten/Student
where data($b/@MatrikelNummer) lt 190000
return <Student Vorname=”{data($b/Vorname)}”></Student>
```

Wie man sieht, iteriert XQuery dabei über die einzelnen Elemente, die als Variablen (hier `$b`) angesprochen werden, und liefert bei erfolgreichem `where`-Vergleich den unter `return` angegebenen Ausdruck. Unterelemente werden als Pfad angesprochen. Schließlich ersetzt `lt` (less than) das Zeichen <.

SQL unterstützt XQuery dahingehend, dass über die Funktion `XMLGEN` eine solche Abfrage direkt ausgeführt werden kann. Auf SQL-Spalten kann dabei direkt über `$Spaltenname` zugegriffen werden. Ist die Spalte vom Datentyp `XML`, kommt wiederrum die Pfadschreibweise zum Einsatz.

Die im Folgenden besprochen Funktionen (außer `XMLAGG`) können auf diese Funktion zurückgeführt werden.

11.4 Export von SQL nach XML

Neben `XMLGEN` werden von SQL mehrere Funktionen angeboten, mit denen sich Spezialfälle ausführen lassen.

11.4.1 Funktionen zum Erzeugen von XML

11.4.1.1 Erstellen von Elementen mit XMLELEMENT

`XMLELEMENT(NAME` *Elementname*`,XMLATTRIBUTES(`*Attribut*$_1$`,...,`*Attribut*$_n$`),`*Inhalt*`)`
liefert ein einzelnes XML-Element zurück. Um in einer Tabelle Studenten, die Nachname, Vorname und Matrikelnummer enthält, ein XML-Element für jeden Studenten zu erstellen, kann man wie folgt anfragen:

```
SELECT
XMLELEMENT(NAME „Student“,
XMLATTRIBUTES(Nachname,Vorname),MatrNr) AS XMLWerte
FROM Studenten
```

Liefert folgende Tabelle:

XMLWerte
<Student NACHNAME=”Day” VORNAME=”Eliza”>185934</Student>
<Student NACHNAME=”Glück” VORNAME=”Hans”>193429</Student>

11.4.1.2 Erstellen von Elementwäldern mit XMLFOREST

`XMLFOREST(`*Element*$_1$`,...,`*Element*$_n$`)`
liefert einen Wald aus einzelnen XML-Elementen.

Gehen wir wieder von der eben genannten Tabelle aus, so liefert uns die Abfrage

```
SELECT
XMLFOREST(MatrNr,
XMLELEMENT(NAME „Student“, XMLATTRIBUTES(Nachname,Vorname)))
AS XMLWald
FROM Studenten
```

folgende Tabelle:

XMLWald
<MatrNr>185934</MatrNr><Student NACHNAME=”Day” VORNAME=”Eliza” />
<MatrNr>193429</MatrNr><Student NACHNAME=”Glück” VORNAME=”Hans” />

Insbesondere ist hier zu sehen, dass Spalten implizit in Elemente umgewandelt werden, die den Spaltenname als Elementnamen haben, und dass Elemente auch leer sein dürfen.

11.4.1.3 Konkatenieren von Elementen mit XMLCONCAT

`XMLCONCAT(Element`$_1$`,...,Element`$_n$`)` konkateniert die gegebenen Elemente zu einem Wald.

11.4.1.4 XML-Inhalte aus Spalten gruppieren mit XMLAGG

XMLAGG ist eine Aggregatfunktion, die die eine Gruppe von Datensätzen zu einem Wald konkateniert. So könnte man z.B. die Vornamen aller Studenten mit gleichem Nachnamen zusammenfassen.

```
SELECT Nachname, XMLAGG(Vorname) AS XMLGruppe FROM Studenten
GROUP BY Nachname
```

liefert dann:

Nachname	**XMLGRUPPE**
Day	`<VORNAME>Eliza</VORNAME><VORNAME>Doris</VORNAME>`
Glück	`<VORNAME>Hans</VORNAME>`

11.4.1.5 Parsen von Strings mit XMLPARSE

Damit die Arbeit mit XML nicht zur „Einbahnstraße“ wird, gibt es noch die Funktion
`XMLPARSE({DOCUMENT|CONTENT} Eingabestring [{PRESERVE|STRIP} WHITESPACE}`
Diese wandelt den Eingabestring zu einem XML-Typen um, der wie oben gesehen weiter verarbeitet werrden kann. `DOCUMENT` oder `CONTENT` gibt an, ob der String ein XML-Dokument oder XML-Inhalt beschreiben soll. Mit `PRESERVE` oder `STRIP WHITESPACE` wird schließlich noch angegeben, ob im String vorhandener Leerraum beibehalten oder (wie bei der Arbeit mit XML üblich) entfernt werden soll.

11.4.2 Export ganzer Datenbankobjekte

Neben diesen „kleinen“ Exportfunktionen definiert SQL/XML einen ganzen XML-Namensraum für SQL.
Wenn wir ganze Tabellen exportieren, so werden dazu zwei Dokumente benötigt:

1. Ein Schema-Dokument, in dem die Datentypen der Tabelle mitsamt Einschränkugen (Maximalwert, Minimalwert etc.) gespeichert sind
2. Ein Datendokument, in dem die eigentlichen Inhalte der Tabellen abgelegt werden

Auch für die anderen Datenbankobjekte sind Regeln zum Export definiert. Eine genaue Besprechung würde allerdings viel zu weit führen, da dazu weitgehende Kenntnisse von XML-Schema nötig wären.

12 Zusammenfassung

Auch wenn SQL 2003 auf den ersten Blick wie eine Ergänzung zu SQL 99 wirkt, so stellt die neue Version einige interessante neue Konzepte dar und zeigt in gewisser Weise auch eine Philosophie, die weniger auf die Verwirklichung völlig neuer Ideen wie Objektorientierung, als vielmehr auf die Standardisierung von schon vorhandenen proprietären Umsetzungen setzt.
Das Kopieren von Tabellenschemata wird in erster Linie den Datenbankenentwickler als „syntaktischen Zucker“ unterstützen.
Das Konzept der „manifestierten Views“ wird mit dem „AS“ Operator der DDL zumindest statisch umgesetzt, auch wenn SQL sich weiterhin den „self-maintainable views“ noch nicht öffnet und die damit verbundenen Probleme durch Verbote umgeht. So wäre es sicherlich wünschenswert, die generierten Spalten auch tabellenübergreifend einsetzen zu können, was wiederum zu einem höheren Überwachungsaufwand für diese Spalten führen würde.
Bei den Identitätsspalten traten offensichtlich zwei Fraktionen gegeneinander an, die eine möchte dem DBMS die Wahl der Referenzwerte überlassen, die andere möchte die automatisch erzeugten Werte möglich auch außerhalb ihrer eigentlichen Funktion, z.B. als Kundennummer im „Real-Life“ anwenden, in gewisser Weise mit dem Streit über Zeiger in Programmiersprachen vergleichbar.
Nachdem mit dem Referenztyp in SQL 99 die eine Möglichkeit realisiert wurde, wurde nun also auch die andere Variante eingeführt.
Mit Multimengen verabschiedet sich SQL mehr und mehr von Codds streng relationalem Modell, insbesondere von der 1. Normalform. Mit ihnen wird das vor vier Jahren eingeführte Objektmodell erst sinnvoll einsetzbar.
Mit den externen Funktionen wird ein weiter Schritt in Richtung Außenwelt getan, da nunmehr das Einbinden von anderen Programmiersprachen offiziell ermöglicht wird.
Sowohl die Fensterfunktionen als auch die Stichprobenentnahme aus Daten dürfte besonders für Statistiker interessant sein, zumal auch Funktionen wie lineare Regression nun im Standard enthalten sind, wiederum syntaktischer Zucker.
Ganz neue Wege und damit auch eine Entwicklung in Richtung Web und W3C beschreitet SQL mit „SQL/XML“. Zwar hatten verschiedene DBMS Möglichkeiten zum Export nach XML geboten, da aber hier jedes Produkt seine eigene Lösung hatte, war die Nutzbarkeit doch sehr stark eingeschränkt.
Diese Erweiterung dürfte meines Erachtens daher auch die wichtigste sein, insbesondere wenn man bedenkt, das Datenbanksysteme auch in großem Maß als Server-Systeme im WWW eingesetzt werden.
Hier dürfte auch die Standardisierung am dringendsten gewesen sein, da gerade hier verschiedenste Implementierungen interagieren sollen.

Bibliographie

1. Eisenberg, Andrew et al., „SQL:2003 Has Been Published“, SIGMOD Record, Vol.33, No. 1, pp. 119-126, March 2004
2. iAnywhere, Dokumentation zu „SQL Anywhere 9.0.2“, http://www.ianywhere.com/developer/product_manuals/sqlanywhere/0902/de/html/index.html
3. Kulkami, Krishna, „Overview of SQL:2003“, Silicion Valley Laboratory of the IBM Corporation, San Jose, 2003-11-06
4. Türker, Can, „SQL:2003 – Was dürfen wir erwarten?“, Datenbank Spektrum 4/2002
5. Türker, Can „SQL:1999 & SQL:2003“, dpunkt.verlag, Heidelberg 2003
6. W3C, „XQuery 1.0: An XML Query Language“, W3C Working Draft 04 April 2005 http://www.w3.org/TR/xquery