Teil 3:

SQL Fortsetzung: Joins, Subselects, Aggregatfunktionen, Derived Tables, Views

Joins

Unser bisheriges SQL erlaubt bereits die Spezifikation von INNER-Joins durch Kombination von Produkt und Restriktion. In vielen Fällen, im speziellen für OUTER-Joins ist es aber erforderlich, spezielle Syntax für Joins zu verwenden. Statt eines Produktes in *SelBase* spezifiziert man gleich den Join inklusive der Join-Bedingung (Restriktion). Das kann in manchen Fällen auch zu höherer Effizienz führen. In jedem Fall verbessert es die Lesbarkeit von Select-Anweisungen.

§ SelBase = Tabellenname {", " Tabellenname | JoinType Tabellenname "ON" Cond}.

§ JoinType = (("LEFT" | "RIGHT" | "FULL") ["OUTER"] | ["INNER"]) "JOIN".

Bei Outer-Joins wird ein erweitertes Produkt gebildet, dass alle Tupel der rechten, linken oder beider Relationen enthält. Fehlende Attribute sind als NULL definiert.

Die obige Grammatikregel für *SelBase* ist etwas vereinfacht. In Wirklichkeit können Joins auch geklammert werden um von der Linksassoziativität abzuweichen.

Manche Datenbanken (z.B. Oracle) unterstützen diese Form von Joins nicht. Sie erlauben OUTER-Joins über spezielle Vergleichsoperatoren (z.B. =*, *=, *=*), die zur Bildung eines erweiterten Produktes führen. Diese Form ist aber weniger mächtig, nicht standardisiert und führt nicht immer zum genau gleichen Ergebnis.

Beispiel: Suche freie Artikelnummern unter Berücksichtigung von Lücken mittels LEFT-Join

```
SELECT A.artnr + 1
FROM Artikel A LEFT JOIN Artikel B ON (A.artnr + 1 = B.artnr)
WHERE B.artnr IS NULL
```

Subselect

SQL erlaubt die Verwendung von SELECT auch innerhalb von Ausdrücken (Subselect, Subquery, nested select). In diesem Fall muss der SELCT-Ausdruck in runde Klammern eingeschlossen werden. Generell wird zwischen *correlated* und *uncorrelated* Subqueries unterschieden.

Eine uncorrelated Subquery verwendet keine Attribute der umschließenden Query. Es genügt daher, wenn das RDBMS sie ein einziges Mal ausführt und die Ergebnisse für die Verwendung in der umschließenden Query zwischenspeichert.

Eine correlated Subquery liegt vor, wenn die Subquery Attribute der umschließenden SELECT-Anweisung benutzt. Die Subquery muss dann für alle Tupel der umschließenden Anweisung ausgeführt werden. In vielen Fällen lassen sich solche Subselects auch als Joins schreiben. In manchen Fällen kann der Query-Compiler ein derartiges Subselect automatisch in einen Join umformen und damit effizienter ausführen.

§ Subselect = "(" SelectExpr ")".

Beispiel für correlated Subquery: Suche freie Artikelnummern unter Berücksichtigung von Lücken mittels Subquery.

```
SELECT artnr + 1
FROM Artikel A
WHERE (SELECT B.artnr FROM Artikel B WHERE B.artnr = A.artnr + 1)
      IS NULL
```

Exists

Der EXISTS-Operator innerhalb von Ausdrücken wertet ein SELECT aus und liefert true, wenn das Ergebnis nicht leer ist.

§ ExistsOp = "EXISTS" Subselect.

Beispiel: Suche freie Artikelnummern unter Berücksichtigung von Lücken mittels EXISTS.

```
SELECT artnr + 1
FROM Artikel A
WHERE NOT EXISTS
  (SELECT B.artnr FROM Artikel B WHERE B.artnr = A.artnr + 1)
```

Quantifizierte Operatoren

ALL oder ANY (alias SOME) nach Vergleichsoperator (=, <>, <, >, <=, >=) (z.B. =ANY) vergleicht einen skalaren Wert mit einer Menge von einstelligen Tupel. Semantisch steckt dahinter ein Allquantor der Art: (für alle i: i Element aus Subselect: skalar op i) bzw. ein Existenzquantor der Art (es existiert ein i: i Element aus Subselect: skalar op i).

§ quantifiedOp = ("=" | "<>" | ... ">=") ("ALL" | "ANY" | "SOME") Subselect.

Beispiel: Selektiere alle Lieferanten, von denen in der Tabelle Artikel ausschließlich teure Artikel (preis > 1000) geführt werden.

```
SELECT * FROM Lieferant L WHERE 1000 < ALL (SELECT preis FROM Artikel A
  WHERE A.lifnr = L.lifnr)
```

Mengenwertiges Insert

Durch Kombination von INSERT und SELECT kann in vielen RDBMS nicht nur eine Zeile pro INSERT eingefügt werden, sondern eine ganze Relation. Die Regel für die Insert-Anweisung wird erweitert zu:

§ InsertStat = "INSERT" ["INTO"] Tabellenname ["(" AttrList ")"]

("VALUES" "(" Const {"," Const} ")"

| SelectExpr).

Beispiel: Kopieren einer Tabelle

```
INSERT INTO ArtikelCopy SELECT * FROM Artikel
```

Erzeugen einer großen Testtabelle: Annahme: sch(Tab10): {[x: integer]} mit den Werten 0..9.

```
INSERT INTO BigTable
 SELECT T1.x*10000 + T2.x*1000 + T3.x*100 + T4.x*10 + T5.x
 FROM Tab10 T1, Tab10 T2, Tab10 T3, Tab10 T4, Tab10 T5
```

Aggregatfunktionen

Erlauben das Verdichten von Mengen auf skalare Werte. Beispiele sind das Zählen der Elemente, das Bilden einer Summe, das Finden von Maximum oder Minimum oder das Bilden des arithmetischen Mittels. Aggregatfunktionen sind innerhalb von *SelList* erlaubt. Ein Mischen von Aggregatfunktionen und normalen Attributen oder Ausdrücken, in den normale Attribute vorkommen, ist nicht sinnvoll und daher verboten. Man kann eine Relation nicht gleichzeitig verdichten und nicht verdichten.

§ AggregateFunction = "COUNT" "(" "*" | ["DISTINCT"] Name ")"
| "SUM" "(" ["DISTINCT"] Expr ")"
| "MIN" "(" Expr ")"
| "MAX" "(" Expr ")"
| "AVG" "(" ["DISTINCT"] Expr ")".

Man beachte den Unterschied zwischen SELECT DISTINCT ... und DISTINCT innerhalb einer Aggregatfunktion. Das äußere DISTINCT wirkt auf die Ergebnismenge, das DISTINCT innerhalb der Aggregatfunktionen wirkt auf die Berechnung dieser Funktion. Wenn SELECTs mit Aggregaten nur eine Ergebniszeile liefern, ist das äußere DISTINCT auf jeden Fall überflüssig.

Beispiele: Die Anzahl der verschiedenen Lieferanten der Artikel

```
SELECT COUNT(DISTINCT lifnr) FROM Artikel
```

Der gesamte Lagerwert:

```
SELECT SUM(preis * menge) AS Lagerwert FROM Artikel
```

Gruppenwechsel

Die Aggregatfunktionen wirken normalerweise auf die gesamte Ergebnismenge. Manchmal möchte man aber eine Gruppenbildung innerhalb einer Ergebnisrelation und auf die Gruppen möchte man Aggregatfunktionen anwenden. Ein Beispiel wäre eine Tabelle der Lagerwerte pro Lieferant. Diese könnte mit einem Subselect erstellt werden wie folgt:

```
SELECT DISTINCT lifnr,
  (SELECT SUM(menge * preis) FROM Artikel AS A
     WHERE A.lifnr = Artikel.lifnr) AS Lagerwert
FROM Artikel
```

Der Nachteil dieser Lösung ist die komplizierte Formulierung und die ev. langsame Ausführung, wenn für jeden Lieferant (ohne ausreichende Optimierung) ein Subselect ausgeführt wird.

Abhilfe schafft hier der Gruppenwechselmechanismus von SQL, der es gestattet, die Gruppierung durch das syntaktische Konstrukte GROUP BY angefügt an SelectTerm auszudrücken.

§ SelectTerm = "SELECT" .... ["GROUP" "BY" AttrList].

Bei jeder Änderung einer der Ausdrücke in der Gruppierungsliste wird ein Gruppenwechsel ausgelöst und die Aggregatfunktionen wirken auf alle Elemente der Gruppe. Attribute, die in der Gruppenwechselliste vorkommen, können in SelList stehen. Obiges Beispiel kann damit vereinfacht werden zu:

```
SELECT lifnr, SUM(menge * preis) AS Lagerwert
FROM Artikel
GROUP BY lifnr
```

Es könnte auch eine Sortierung der Ausgabe erfolgen unter Verwendung der Ergebnisse der Aggregatfunktionen.

```
SELECT lifnr, SUM(menge * preis) AS Lagerwert
FROM Artikel
GROUP BY lifnr
ORDER BY Lagerwert
```

Nicht in jedem Fall möchte man alle Gruppen ausgeben. Daher benötigt man noch einen Mechanismus zur Einschränkung der betrachteten Gruppen. Dieser Mechanismus ist in Form der HAVING-Klausel verfügbar. Die nach HAVING angegebene Bedingung muss für die Gruppen erfüllt sein. In der Having-Bedingung sind nur Attribute erlaubt, die auch für die Gruppierung verwendet werden einschließlich Aggregatfunktionen, die im WHERE nicht erlaubt sind.

§ SelectTerm = "SELECT" .... ["GROUP" "BY" Expr {"," Expr} ["HAVING" Cond]].

Beispiel: Selektieren der Lagerwerte der wichtigen Lieferanten, das seien jene, für die der Lagerwert eine bestimmte Grenze überschreitet.

```
SELECT lifnr, SUM(menge * preis) AS Lagerwert
FROM Artikel
GROUP BY lifnr HAVING SUM(menge * preis) > 10000
ORDER BY Lagerwert
```

Roll-up, drill-down

Das Hinzufügen von Attributen in die Gruppierungsliste wird als *drill-down* bezeichnet. Mit jedem zusätzlichen Attribut erfolgt eine feinere Gruppierung und damit eine weniger starke Verdichtung der Daten. Das Gegenteil ist das Weglassen von Attributen, das als *roll-up* bezeichnet wird. Es führt zu einer gröberen Gruppierung und damit zu einer stärkeren Verdichtung der Ergebnisse. Das extremste roll-up ist das Weglassen aller Attribute, das zu einem einzigen Ergebnistupel führt weil die gesamte Tabelle als eine einzige Gruppe betrachtet wird. Das extremste drill-down wäre das Hinzufügen aller Attribute, was die ursprüngliche Tabelle als Ergebnis liefern würde.

Data Warehouse

Das Verdichten von Daten ist eine zeitaufwändige Operation wenn alle Tupel einer Relation berücksichtigt werden müssen. Solche Operationen können die gleichzeitige Benutzung eines RDBMS für andere Benutzer stark einschränken. Daher hat man sich eine Abhilfe überlegt, die zum Begriff *Data Warehous* führt. Darunter versteht man eine Ansammlung von Daten, die (hauptsächlich) read-only zugegriffen wird. Der Hauptzweck ist die statistische Analyse von historischen Daten, zum Beispiel die Analyse von Verkaufszahlen pro Jahr, Region, Artikelgruppe etc. Man spricht in diesem Zusammenhang auch von OLAP (online analytical processing) im Gegensatz zu OLTP (online transactional processing). Die Aggregierung von Daten in einem Data Warehouse ist durch die Verwendung einer speziellen Speicherungsstruktur wesentlich effizienter als mit normalen Relationen.

Hat man kein Data Warehouse zur Verfügung, so kann man in vielen RDBMS durch *Materialisierung von Aggregaten* eine gute Lösung erzielen. Man möchte aber nicht für alle N möglichen drill-down Stufen einer Relation R eine eigene Aggregierung durchführen, weil der Aufwand (N * |R|) ist, wobei N mit der Anzahl der Attribute exponentiell ansteigt (bei 3 Attributen ist $N = 2^3 = 8$) und N einzelne SQL-Befehle formuliert werden müssen. Besser ist es, wenn man mit einem Befehl (und einem Durchlauf) die feinste Gruppierung durchführt und basierend auf diesen Daten alle möglichen roll-up Stufen berechnet. Dadurch können

weniger stark verdichtete Daten aus stärker verdichteten Daten ermittelt werden, ohne jedes mal die gesamte Relation zu lesen.Genau diese Eigenschaft hat der CUBE-Operator, der in vielen RDBMS verfügbar ist. Der Operator ROLLUP, der meist zusätzlich zu CUBE verfügbar ist, liefert nicht alle möglichen 2^x Kombinationen von Verdichtungen sondern geht hierarchisch (gemäß der Attributreihenfolge) vor und liefert damit *x* Verdichtungsstufen.

CUBE Operator

Der Cube-Operator wird als Option zu GROUP BY angegeben und bewirkt dass zusätzlich zu den durch GROUP BY gelieferten Tupel alle möglichen Kombinationen von verdichteten Daten geliefert werden. Dabei gibt es verschiedene Schreibweisen. Die erste ist die in SQL-99 vorgeschlagene. Die zweite ist eine ältere Form.

§ SelectTerm = ... "GROUP" "BY" "CUBE" "(" AttrList ")".

§ SelectTerm = ... "GROUP" "BY" AttrList "WITH" "CUBE".

Mit Hilfe der INSERT-SELECT Anweisung kann das Ergebnis von CUBE in eine Hilfstabelle geschrieben werden, die als Data-Warehouse verwendet werden kann.

Der Cube-Operator liefert NULL für verdichtete Attribute. Man beachte, dass Primärschlüssel nicht NULL sein dürfen. Daher wird in folgendem Beispiel kein Primärschlüssel auf der Ergebnistabelle definiert.

Beispiel: Verkaufstabelle enthält Tupel pro verkauftem Artikel.

sch(Verkauf): {[belegnr, artnr, jahr, region, artikelgruppe, menge]}

sch(Verkaufsstatistik): {[jahr, region, artikelgruppe, menge]}

```
INSERT INTO Verkaufsstatistik
  SELECT jahr, region, artikelgruppe, sum(menge)
  FROM Verkauf GROUP BY jahr, region, artikelgruppe WITH CUBE
```

Zur Selektion der Verkaufszahlen der Region 9:

```
SELECT * FROM Verkaufsstatistik
WHERE jahr IS NULL AND region = 9 AND artikelgruppe IS NULL
```

Abgeleitete Tabellen (derived tables)

Der Namensbestandteil S in SQL (Structured) kommt nicht zuletzt daher, dass man in SQL neben Subselects innerhalb von Ausdrücken auch Subselects zum Definieren der SelBase verwenden kann. Man kann also nicht nur eine in der Datenbank verfügbare (physische) Tabelle verwenden sondern auch Tabellen, die durch Selects definiert sind. Damit können komplexe Bedingungen, die man nur schwer oder gar nicht in einem SELECT unterbringt, in mehrere einfache Selects zerlegt werden, die von innen nach aussen zusammengesetzt werden. Solche innerhalb eines Selects erzeugten und benannten Hilfstabellen heissen abgeleitete (engl. derived) Tabellen.

§ SelBase = (Tabellenname | Subselect ["AS"] Name) ... .

Beispiel: Einschränken des Ergebnisses einer Aggregatfunktion ohne Verwendung von HAVING, zB Selektion der Lagerwerte aller Lieferanten mit Wert > 1000

```
SELECT lifnr, Lagerwert
FROM (SELECT lifnr, SUM(menge * preis) AS Lagerwert
     FROM Artikel
     GROUP BY lifnr) AS LWert
WHERE Lagerwert > 1000
ORDER BY Lagerwert
```

Views

Zur weiteren Vereinfachung von Selects kann man eine derived table auch im Schema der Datenbank unter einem Namen ablegen und hat sie über diesen Namen zur Verfügung. Man spricht in diesem Fall von einer *View*. Überall (oder fast überall) wo bisher ein Tabellenname verwendet werden konnte, kann auch ein Viewname stehen. Die Regel für SelBase erweitert sich damit auf:

§ SelBase = (Tabellenname | Viewname | Subselect ["AS"] Name) ... .

Views werden durch das DDL-Kommando CREATE VIEW definiert und gespeichert. Optional kann eine Umbenennung der Attributnamen erfolgen. Views sind nicht parametrisierbar.

§ CreateViewStat = "CREATE" "VIEW" Viewnanme ["(" Name {"," Name ")"}] "AS" SelectExpr.

Beispiel:

```
CREATE VIEW LWert AS
    SELECT lifnr, SUM(menge * preis) AS Lagerwert
       FROM Artikel
       GROUP BY lifnr

SELECT lifnr, Lagerwert FROM LWert
WHERE Lagerwert > 1000
ORDER BY Lagerwert
```

Views können mit dem Befehl DROP VIEW wieder gelöscht werden.

§ DropViewStat = "DROP" "VIEW" Viewname.

Beispiel

```
DROP VIEW LWert
```

Update einer View

Unter bestimmten Bedingungen können auf Views auch UPDATE, INSERT, DELETE-Operationen angewendet werden. Im einfachsten Fall einer 1:1 View kann das einfach nachvollzogen werden.

```
CREATE VIEW ArtikelView AS SELECT * FROM Artikel
```

Die View ist identisch zur darunterliegenden Tabelle, daher wirken sämtliche Modifikation so, also ob sie auf Artikel angewendet würden. Bei Verwendung von Projektion, Joins, etc. wird es schon schwieriger, bei Verwendung von Aggregaten oder UNION wird es unmöglich.

RDBMS stellen häufig spezielle Mechanismen bereit, um View-Updates explizit definieren zu können. Die ursprünglich angeforderte Änderungsoperation wird auf eine spezielle Anweisung abgebildet, die als Teil des Schemas definiert ist (z.B. INSTEAD OF Trigger in MsSQL Server).

RDBMS müssen Information bereitstellen, ob eine View *Updateable* ist oder nicht. Zumindest einfache Views müssen updateable sein.

Anwendung von View Updates speziell nach Schemaänderungen, um Anwendungen unverändert zu lassen. Die alte Tabelle wird als View basierend auf einer neuen, modifizierten Tabelle bereitgestellt.