

Selektion

Selektion → sucht aus einer Relation bestimmte Tupel heraus. Zur Selektion dient ein Selektionsprädikat B mit

$$B: R(A_1, A_2, \dots, A_n) \rightarrow \{true, false\}$$

Die Selektion S aus R unter der Selektionsbedingung B ist definiert durch

$$S_B(R) := \{t \in R \mid B(t) = true\}$$

also alle Tupel t der Relation R , für die die Selektionsbedingung $B(t)$ ein wahres Ergebnis liefert. Alle anderen Tupel werden ausgeblendet.

In der Praxis werden Selektionen durch Vergleich von Spalten mit Konstanten oder anderen Attributen mit den Vergleichsoperatoren $<, \leq, =, >, \geq$ durchgeführt. Mehrere Vergleiche können durch logische Verknüpfung (UND, ODER, NICHT) verbunden werden. Die Reihenfolge mehrerer hintereinander ausgeführter Selektionen darf vertauscht werden. Dabei sollte aus Effizienzgründen jedoch die Selektion mit der kleinsten Ergebnisrelation nach innen gezogen werden.

Projektion

Projektion → Ausblenden von Spalten einer Relation. Gegeben sei eine Relation R mit $R := R(A_1, A_2, \dots, A_n)$ und eine geordnete Teilmenge x der Attribute $A_1 \dots A_n$ mit $x = (A_{i_1}, A_{i_2}, \dots, A_{i_k})$ und $k < n$. Dann ist die Projektion auf x definiert durch

$$P_x(R) := \{t' = (d_{i_1}, \dots, d_{i_k}) \mid t \in R\}$$

wobei identische Tupel eliminiert werden.

Kartesisches Produkt

Das kartesische Produkt zweier Relationen R_1 und R_2 mit $R_1 = R(A_1, A_2, \dots, A_n)$ und $R_2 = R(B_1, B_2, \dots, B_m)$ wird durch Kombination jedes Tupels der Relation R_1 mit jedem Tupel der Relation R_2 gebildet:

$$R_1 \times R_2 := \{(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m) \mid (A_1, A_2, \dots, A_n) \in R_1 \wedge (B_1, B_2, \dots, B_m) \in R_2\}$$

Beispiel:

R_1	A_1	A_2	R_2	B_1	B_2	B_3	$R_1 \times R_2$	A_1	A_2	B_1	B_2	B_3
	1	A		1	x	y		1	A	1	x	y
	2	B		2	v	w		1	A	2	v	w
	3	C						2	B	1	x	y
								2	B	2	v	w
								3	C	1	x	y
								3	C	2	v	w

Vereinigung

Die Vereinigung zweier Relationen R_1 und R_2 entspricht der Mengentheoretischen Vereinigung unter den Voraussetzungen:

1. R_1 und R_2 müssen den gleichen Grad haben
2. Jedes Attribut von $A_i \in R_1$ muss Wertebereichskompatibel mit dem korrespondierenden Attribut $B_i \in R_2$ sein (Namensgleichheit ist jedoch nicht erforderlich)

Sind diese Bedingungen erfüllt, heißen R_1 und R_2 *Unionkompatibel*.

Durchschnitt

Der Durchschnitt zweier Relationen R_1 und R_2 entspricht dem Mengentheoretischen Durchschnitt. Voraussetzung ist, dass R_1 und R_2 Unionkompatibel sind.

Differenz

Die Differenz zweier Relationen R_1 und R_2 entspricht der Mengentheoretischen Differenz. Voraussetzung ist, dass R_1 und R_2 Unionkompatibel sind.

Theta-Verbund (Theta-Join)

Der Theta-Join ist die allgemeinste Form des Verbunds. Für zwei Relationen R_1 und R_2 mit $R_1 = R(A_1, A_2, \dots, A_n)$ und $R_2 = R(B_1, B_2, \dots, B_m)$ und ein Selektionsprädikat B mit

$$B := R_1 \times R_2 \rightarrow \{true, false\}$$

der Form

$$B(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m) := A_i \vartheta B_j \text{ mit } \vartheta \in \{<, \leq, =, >, \geq\}, i \in \{1, \dots, n\}, j \in \{1, \dots, m\}$$

ist definiert durch

$$join(R_1, A_i \vartheta B_j, R_2) := \{(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m) \in R_1 \times R_2 \mid A_i \vartheta B_j\}$$

Beispiel:

R_1	A_1	A_2	R_2	B_1	B_2	B_3
	1	A		1	x	y
	2	B		2	v	w
	3	C				

Theta-Join mit Selektionsprädikat $A_1 = B_1$:

$R_{1_{A_1=B_1}} R_2$	A_1	A_2	B_1	B_2	B_3
	1	A	1	x	y
	2	B	2	v	w

Theta-Join mit Selektionsprädikat $A_1 > B_1$:

$R_{1_{A_1>B_1}} R_2$	A_1	A_2	B_1	B_2	B_3
	2	B	1	x	y
	3	C	1	x	y
	3	C	2	v	w

Equi-/Natural-Join

Ein Equi-Join ist ein Theta-Join, der im Selektionsprädikat nur den Gleichheits-Operator zulässt. Werden in der Ergebnisrelation des Equi-Joins alle doppelt vorkommenden Attribute nur einmal aufgelistet, spricht man von einem *Natural-Join*.

Ein Equi- oder Natural-Join heißt *Verlustfrei*, wenn alle Tupel aus R_1 und R_2 im Ergebnis vertreten sind. Dies bedeutet, dass sich die ursprünglichen Relationen durch geeignete Projektionen wieder herstellen lassen.

Zerlegt man umgekehrt eine Relation in zwei neue Relationen und lassen sich diese durch einen Natural-Join wieder zur ursprünglichen Relation zusammenführen, nennt man die Zerlegung *verbundtreu*.

Beispiel: Natural-Join mit Selektionsprädikat $A_1 = B_1$ (R_1 und R_2 siehe oben):

$R_1^*_{A_1=B_1} R_2$	A_1	A_2	B_2	B_3
	1	A	x	y
	2	B	v	w

Die beiden Projektionen $P_{(A_1, A_2)}$ und $P_{(A_1, B_2, B_3)}$ stellen die Ausgangsrelation wieder her. Im zweiten Fall wird dabei das Tupel $(2, v, w)$ nur einmal im Ergebnis berücksichtigt. Der Join ist also Verlustfrei.

Outer-Joins

Die bisher vorgestellten Joins werden auch als *Inner-Joins* bezeichnet, da sie nur Tupel in der Ergebnisrelation erzeugen, falls der Attributwert der ersten Relation auch in der zweiten Relation vorkommt. Im Gegensatz dazu erzeugt ein *Outer-Join* zumindest alle Tupel einer Relation im Ergebnis. Je nachdem welche dies ist, spricht man von einem *Left*-, *Right*- oder *Full Outer Join*.

Der Left-Outer-Join zweier Relationen R_1 und R_2 ist ein Natural-Join $R_1 * R_2$, bei dem alle Tupel der linken Relation R_1 , die im Natural-Join unterdrückt würden, als Tupel im Ergebnis mit aufgeführt werden. Die fehlenden Werte der Spalten aus R_2 werden dabei mit dem *NULL*-Wert aufgefüllt. Der Right-Outer-Join wird entsprechend definiert.

Ein Full-Outer-Join vervollständigt in beide Richtungen. Er ist ein Natural-Join, bei dem alle Tupel der linken und rechten Relation erhalten bleiben und fehlende Werte mit *NULL* aufgefüllt werden.

Beispiel:

R_1	A_1	A_2	R_2	B_1	B_2	B_3
	1	A		1	x	y
	2	B		2	v	w
	3	C		4	k	l

Left-Outer-Join mit dem Selektionsprädikat $A_1 = B_1$:

$R_1^*_{A_1=B_1} R_2$	A_1	A_2	B_1	B_2	B_3
	1	A	1	x	y
	2	B	2	v	w
	3	C	NULL	NULL	NULL

Right-Outer-Join mit dem Selektionsprädikat $A_1 = B_1$:

$R_1^*_{A_1=B_1} R_2$	A_1	A_2	B_1	B_2	B_3
	1	A	1	x	y
	2	B	2	v	w
	NULL	NULL	4	k	l

Full-Outer-Join mit Selektionsprädikat $A_1 = B_1$:

$R_1^*_{A_1=B_1} R_2$	A_1	A_2	B_1	B_2	B_3
	1	A	1	x	y
	2	B	2	v	w
	3	C	NULL	NULL	NULL
	NULL	NULL	4	k	l

Generelle Anmerkungen zu Joins

Folgendes gilt für alle Typen von Joins:

- Die Join-Attribute, über die der Join ausgeführt wird, müssen keine Keys sein
- Die Join-Attribute müssen in beiden Relationen nicht notwendig gleich heißen
- Die Domänen der Join-Attribute müssen übereinstimmen
- Jede Relation kann mit jeder anderen durch einen Join verknüpft werden, falls obige Bedingung erfüllt ist

Wird eine Relation mit sich selbst verknüpft, nennt man das einen *Auto-Join*.

Division

Für zwei Relationen R_1 und R_2 mit $R_1 = R(A_1, A_2, \dots, A_n)$ und $R_2 = R(A_{i_0})$ mit $i_0 \in \{1, \dots, n\}$ ist das Ergebnis der Division erklärt durch:

$$R_1 \div R_2 := P_{A'}(R_1) - P_{A'}((P_{A'}(R_1) \times R_2 - R_1)) \text{ mit } A' = \{A_1, A_2, \dots, A_n\} - \{A_{i_0}\}$$

Beispiel:

R_1	A_1	A_2
	1	A
	1	B
	1	C
	2	A
	3	B

R_2	A_2
	A
	B

$R_1 \div R_2$	A_2
	1

Die Division im Relationalen Modell ist mit der ganzzahligen Division vergleichbar: $R = R_1 \div R_2$ ist die größte Relation für die $R * R_2$ Teilmenge von R_1 ist.

Die Structured Query Language SQL

SQL ist:

- Eine Datendefinitionssprache (Data Definition Language, DDL)
- Eine View-Definitionssprache (View Definition Language, VDL)
- Eine Datenmanipulationssprache (Data Manipulation Language, DML)

Allgemein Regeln zur SQL-Syntax:

- SQL ist Formatfrei, d.h. Anweisungen können über mehrere Zeilen verteilt und durch Leerzeichen und Tabulatoren strukturiert werden
- Anweisungen werden mit einem Semikolon beendet
- Groß- und Kleinschreibung spielt für Schlüsselworte, Tabellen- und Spaltennamen keine Rolle
- Groß- und Kleinschreibung ist signifikant für Feldinhalte

Zur exakten Definition der SQL-Syntax werden wir die sogenannte Erweiterte Backus-Naur-Form (EBNF) verwenden. Sie umfasst folgende Symbole:

- Reservierte Wörter werden groß geschrieben: `SELECT, FROM, UPDATE, ...`
- Alternativentrennung: `[+|-] Konstante`
- Optionalsymbol: `SELECT [ALL] ...`
- Wiederholungssymbol: `FROM Tabelle {, Tabelle}`

SQL als Daten- und View-Definitionssprache

SQL kennt Befehle zum Erstellen neuer Tabellen, Views, Indexe und Domänen sowie zum Ändern derselben. Sie lauten:

- CREATE TABLE | VIEW | INDEX | DOMAIN
- ALTER TABLE | INDEX | DOMAIN

Die CREATE-Anweisung

Die Syntax der CREATE-Anweisung zum Erstellen einer neuen Tabelle lautet wie folgt:

```
create-table          ::=  CREATE TABLE table-name (column-descr
                           , [column-descr | table-constraint])

column-descr          ::=  column-name data-type [column-constraints]

column-constraints    ::=  [CONSTRAINT constraint-name]
                           | NOT NULL
                           | PRIMARY KEY
                           | UNIQUE
                           | REFERENCES other_table

table-constraint      ::=  [CONSTRAINT constraint-name]
                           | PRIMARY KEY (column-list)
                           | UNIQUE {column-list}
                           | FOREIGN KEY {column-list}
                           REFERENCES reference-specification

column-list           ::=  column-name {, column-name}

reference-specification ::=  other_table [ (referenced-columns) ]

data-type             ::=  SMALLINT
                           | INTEGER
                           | FLOAT
                           | DOUBLE PRECISION
                           | DATE
                           | TIME
                           | TIMESTAMP
                           | DECIMAL [(precision [, scale])]
                           | NUMERIC [(precision [, scale])]
                           | CHAR [(length)]
                           | CHARACTER [(length)]
                           | VARCHAR [(length)]
```

Für DECIMAL und NUMERIC bezeichnet *precision* die Anzahl der signifikanten Stellen, *scale* die Anzahl der Nachkommastellen (Default: 0). Ist die Anzahl der Nachkommastellen beim Einfügen zu klein, wird gerundet. Für die String-Typen ist *length* optional, der Default ist 1.

Viele RDBMS unterstützen weitere Datentypen wie BIT, BLOB oder INTERVAL. Hier hilft im Zweifelsfall nur Nachschlagen in der Dokumentation.

Beispiele:

```
CREATE TABLE PERSON
(
    PID            INTEGER NOT NULL PRIMARY KEY,
    NACHNAME       CHAR(80),
    VORNAME        CHAR(80)
);
```

Oder alternative mit einer Table-Constraint:

```
CREATE TABLE PERSON
(
    PID            INTEGER NOT NULL,
    NACHNAME       CHAR(80),
    VORNAME        CHAR(80),
    PRIMARY KEY (PID)
);
```

Eine Angestellten-Tabelle könnte dann wie folgt aussehen:

```
CREATE TABLE EMPLOYEE
(
    EMPID          INTEGER NOT NULL PRIMARY KEY,
    PID            INTEGER REFERENCES PERSON(PID),
    SALARY         DECIMAL(10, 2)
);
```

oder wieder alternativ mit Table-Constraints:

```
CREATE TABLE EMPLOYEE
(
    EMPID          INTEGER NOT NULL,
    PID            INTEGER,
    SALARY         DECIMAL(10, 2),
    PRIMARY KEY (EMPID),
    FOREIGN KEY (PID) REFERENCES PERSON(PID)
);
```

Erstellen von Domänen

Die Syntax der CREATE-Anweisung zum Erstellen einer Domäne lautet wie folgt:

```
create-domain ::= CREATE DOMAIN domain-name [AS] data-type
               [ DEFAULT default-expr ]
               [ domain-constraint ]
               [ check-rule ];
```

```
default-expr  ::= NULL
               | constant
               | built-in-function
```

```
domain-constraint ::= NOT NULL
```

Check-Regeln

Check-Regeln werden verwendet, um den Wertebereich eines Datentyps oder einer Domäne weiter einzuschränken.

Syntax:

```
Check-rule ::= [CONSTRAINT constraint-name] CHECK ( boolean-expression );
```

Beispiele:

- `CREATE DOMAIN Gueltig AS DATE check(value > '2000-01-01');`
- `CREATE TABLE Mitarbeiter
(
 Anrede CHAR(4) CHECK (Anrede IN ('Herr', 'Frau'))
);`

Erstellen von Views

Die Syntax der CREATE-Anweisung zum Erstellen eines neuen Views lautet wie folgt:

```
create-view      ::=  CREATE VIEW view-name [(column-list)]  
                  AS select-statement [WITH CHECK OPTION];
```

```
column-list      ::=  column-name {, column-name}
```

Die Daten eines Views werden nicht separat gespeichert, nur seine Definition. Er stellt lediglich eine neue Sicht auf bestehende Tabellen dar. Trotzdem kann er wie eine Tabelle verwendet werden.

Bei der Definition eines Views können die Spalten umbenannt werden. Dies geschieht durch Angabe einer Liste von Spalten nach dem Namen des Views. Falls dort Angaben gemacht werden, muss ihre Anzahl mit der Anzahl der Spalten des SELECT-Statements übereinstimmen.

Beispiel:

```
CREATE VIEW VNACHNAMEN AS  
    SELECT  
        NACHNAME  
    FROM PERSON;
```

Erstellen von Indexen

Die Syntax der CREATE-Anweisung zum Erstellen eines neuen Index lautet wie folgt:

```
create-index      ::=  CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]  
                      INDEX index-name ON table-name (column {, column});
```

Ein eindeutiger Index (UNIQUE) verhindert das Einfügen doppelter Werte und des NULL-Wertes. Indexe dienen generell der Beschleunigung lesender Zugriffe. Jeder Index erhöht allerdings den Aufwand bei ändernden Operationen, da er ggf. aktualisiert werden muss.

Indexe sollten deshalb mit Bedacht verwendet werden. Gute Gründe für einen Index können sein:

- Eine Spalte taucht sehr häufig in Suchbedingungen auf
- Eine Spalte kommt sehr häufig in Join-Bedingungen vor
- Eine Tabelle enthält sehr viele Datensätze

Wird ein Index zur Beschleunigung von Suchanfragen erstellt, sollte er in der passenden Sortierung (Default ist `ascending`) erstellt werden. Es können auf Spalten(-gruppen) auch gleichzeitig auf- und absteigende Indexe gelegt werden.

Beispiel:

```
CREATE ASCENDING INDEX INACHNAMEN ON PERSON(NACHNAME);
```

Die **ALTER**-Anweisung

Ändern von Tabellendefinitionen

Zu den möglichen Änderungen gehören das

- Hinzufügen
- Ändern
- Löschen

von Spalten oder Constraints. Die Syntax hierzu lautet:

```
alter-table      ::=  ALTER TABLE table-name operation {, operation};
```

```
operation        ::=  ADD col_def  
                    |  ADD table-constraint  
                    |  ALTER [COLUMN] column-name alt-col-clause  
                    |  DROP col  
                    |  DROP CONSTRAINT constraint
```

```
alt-col-clause   ::=  TO new-column-name  
                    |  TYPE new-column-datatype  
                    |  POSITION new-column-position
```

Beispiel:

```
CREATE TABLE PERSON  
(  
    PID          INTEGER NOT NULL,  
    NACHNAME     CHAR(80) ,  
    VORNAME      CHAR(80)  
);
```

```
ALTER TABLE PERSON ADD PRIMARY KEY(PID);
```

Ändern von Indexen

Indexe können lediglich aktiviert bzw. deaktiviert werden. Dies geschieht wie folgt:

```
alter-index      ::=  ALTER INDEX index-name new-state;
```

```
new-state        ::=  ACTIVE | INACTIVE
```

Das Ändern eines Index ist nur möglich, falls er nicht als `UNIQUE` oder zu einem primären Schlüssel oder einer Fremdschlüsselbeziehung gehört. In diesem Fall hilft nur löschen und erneut anlegen.

Das Deaktivieren eines Index kann sinnvoll sein, wenn eine große Datenmenge eingespielt werden soll. Dies kann durch die erforderlichen Aktualisierungen des Index sehr viel Zeit kosten, die durch Deaktivieren und anschließendes Reaktivieren des Index eingespart werden kann.

Die DROP-Anweisung

Das Löschen von Tabellen, Indexen und Views erfolgt schlicht und einfach mit dem Befehl

```
drop-something ::= DROP what some-name;
```

```
what ::= TABLE | VIEW | INDEX | DOMAIN
```

Der SQL92-Standard kennt noch weitere Befehle zu Definitionszwecken. Die bisher vorgestellten sind jedoch mit Abstand die gängigsten. Die nicht erwähnten sind zudem nicht in allen verbreiteten RDBMS vollständig unterstützt.

SQL als Daten-Abfragesprache

Die folgenden Aspekte einer Datenbankabfrage müssen mit Hilfe einer Datenabfragesprache abgebildet werden können:

- Welche Attribute sollen angezeigt werden?
- Unter welchem Namen erscheint ein Attribut im Ergebnis?
- Woher kommen die Daten?
- Welche Bedingung muss ein Datensatz erfüllen, um im Ergebnis zu erscheinen?
- In welcher Reihenfolge erscheinen die Datensätze im Ergebnis?
- Sollen die Datensätze gruppiert werden?

Die SELECT-Anweisung

Das SQL-Arbeitstier, das alle diese Anforderungen erfüllen kann, ist die `SELECT`-Anweisung. Ihre Syntax lautet:

```
Select-statement ::= SELECT [DISTINCT | ALL]
                    [[table-name|table-alias],]*
                    | select-column {, select-column}
                    FROM table-reference {, table-reference}
                    [WHERE search-condition]
                    [ORDER BY order-list];
```

```
Select-column ::= [[table-name|table-alias].]column-name
                 [AS new-column-name]
                 | constant [AS new-column-name]
                 | expression [AS new-column-name]
                 | function [AS new-column-name]
```

```
Table-reference ::= table-name [table-alias]
```

```
Order-list ::= column-reference {, column-reference}
              [ASC[ENDING] | DESC[ENDING]]
              {, order-list}
```

```
column-reference ::= column-name | column-number
```

Auf den Aufbau des Elements `search-condition` werden wir noch genauer eingehen. In dieser (vereinfachten) Form der `SELECT`-Anweisung finden sich die relationalen Operationen Selektion (`WHERE`), Projektion (`SELECT...`) und Umbenennung (`AS...`) wieder.

Erläuterung der einzelnen Bestandteile der SELECT-Anweisung

Durch Angabe eines „*“ im Rumpf der SELECT-Anweisung werden alle Spalten der zugehörigen Tabelle ausgegeben. Bei der Erstellung von Datenbank-Anwendungen sollte dies in der Regel vermieden werden. Nicht benötigte Spalten kosten Speicherplatz und Zeit.

Das optionale Selektionsprädikat direkt nach dem Schlüsselwort SELECT hat folgende Auswirkungen:

Wert	Verhalten
ALL	Ausgabe aller Treffer inklusive doppelt vorkommender Sätze (Default)
DISTINCT	Unterdrückung doppelt vorkommender Datensätze

Nur mit dem Prädikat DISTINCT verhält sich SQL also korrekt im Sinne des relationalen Modells!

Mit AS können Spalten umbenannt werden. Dies ist zwar selten notwendig, kann aber manchmal der Lesbarkeit förderlich sein.

Nach dem Schlüsselwort FROM können eine oder mehrere Tabellen spezifiziert werden. **Vorsicht:** Ohne weitere Vorkehrungen wird das kartesische Produkt der angegebenen Tabellen gebildet. Für jede Tabelle kann ein Alias vergeben werden.

Beispiele (beziehen sich auf `employee.gdb`):

- ```
SELECT * FROM employee;
```

Dieses SELECT liefert alle Sätze der Tabelle `employee`; alle Spalten werden zurückgegeben.

- ```
SELECT
    LAST_NAME,
    FIRST_NAME
FROM employee;
```

Dieses SELECT liefert alle Sätze der Tabelle `employee`, aber nur die Spalten `LAST_NAME` und `FIRST_NAME`

- ```
SELECT
 T1.LAST_NAME,
 T2.DEPARTMENT
FROM EMPLOYEE T1, DEPARTMENT T2;
```

Das Resultat dieses SELECT ist eine Liste, in der jeder Nachname aus `employee` mit jedem Abteilungsnamen aus `department` gepaart ist (kartesisches Produkt!).

Kleiner Vorgriff: Wie kann das repariert werden? Lösung:

```
SELECT
 T1.LAST_NAME,
 T2.DEPARTMENT
FROM EMPLOYEE T1, DEPARTMENT T2
WHERE T1.DEPT_NO = T2.DEPT_NO;
```

Hier handelt es sich um einen Equi-Join, gefolgt von einer Projektion. Ohne die Projektion (also mit einem „\*“) würde das Ergebnis die Spalte `DEPT_NO` zweimal enthalten, einmal die aus `employee` und zusätzlich die aus `department`.

- ```
SELECT
    JOB_CODE
FROM EMPLOYEE
ORDER BY 1;
```

Dieser `SELECT` liefert alle Funktionszebezeichnungen aller Beschäftigten mit Wiederholungen, insgesamt 42 Treffer, da in der Tabelle `employee` 42 Einträge existieren. Mit der Modifikation

```
SELECT DISTINCT
    JOB_CODE
FROM EMPLOYEE
ORDER BY 1;
```

sind es nur noch 13 tatsächlich verschiedene. Wichtig: Das Schlüsselwort `DISTINCT` bezieht sich auf ganze Tupel. Mit der zusätzlichen Spalte `JOB_COUNTRY`

```
SELECT DISTINCT
    JOB_CODE,
    JOB_COUNTRY
FROM EMPLOYEE
ORDER BY 1;
```

werden es mehr, nämlich 22 Treffer.

Die WHERE-Klausel

Im vorhergehenden Abschnitt haben wir bei der Beschreibung der Syntax der `SELECT`-Anweisung die `WHERE`-Klausel und die möglichen Selektionsprädikate nicht näher spezifiziert. Dies wird jetzt nachgeholt. Die einfachste Form der `WHERE`-Klausel besteht aus einer Reihe von logisch miteinander verknüpften Vergleichsausdrücken:

```
search-condition ::= value operator value
                  | NOT search-condition
                  | search-condition OR search-condition
                  | search-condition AND search-condition

operator         ::= = | < | > | <= | >= | !< | !> | <> | !=

value            ::= [[table-name|table-alias].]column-name
                  | constant
                  | arithmetic-expression
```

Der `NULL`-Wert muss in Vergleichsausdrücken besonders behandelt werden:

```
value IS [NOT] NULL
```

Beispiel (bezieht sich auf die Datenbank `employee.gdb`):

Die `SELECT`-Anweisung

```
SELECT
    DEPARTMENT
FROM DEPARTMENT WHERE MNGR_NO IS NULL;
```

liefert die vier Treffer

Marketing
Software Products Div.
Software Development
Field Office: Singapore

Nicht erlaubt ist hingegen:

```
SELECT
    DEPARTMENT
FROM DEPARTMENT WHERE MNGR_NO = NULL;
```

Weitere Möglichkeiten zur Formulierung der WHERE-Klausel liefern die drei Operatoren

- value [NOT] BETWEEN value AND value
- value [NOT] LIKE value
- value [NOT] in (value {, value})

Der BETWEEN-Operator

Beispiel (bezieht sich auf die Datenbank `employee.gdb`):

Die SELECT-Anweisung

```
SELECT
    LAST_NAME,
    FIRST_NAME,
    SALARY
FROM EMPLOYEE WHERE SALARY BETWEEN 40000 AND 50000;
```

liefert die drei Treffer

Nordstrom	Carol	42742.5
Page	Mary	48000
Yanowski	Michael	44000

Mit dem BETWEEN-Operator lassen sich auch völlig sinnfreie Bedingungen erzeugen:

```
SELECT * FROM EMPLOYEE WHERE 1 BETWEEN 2 AND 3;
```

zum Beispiel liefert die leere Menge zurück.

Der LIKE-Operator

Der LIKE-Operator ermöglicht die Verwendung von Wildcards beim Vergleich mit Zeichenketten. Er ist nur für die Typen CHAR, CHARACTER und VARCHAR definiert.

Der ANSI-Standard kennt nur zwei Platzhalter:

Platzhalter	Wirkung
%	Steht für kein, ein oder mehrere Zeichen
_	Steht für genau ein Zeichen

Demzufolge werden * und ? nicht von allen RDMBS unterstützt und sollten deshalb vermieden werden! Auch funktioniert % nicht auf allen Plattformen so, wie man es erwarten würde. Am sichersten ist es. Nur Ausdrücke zu verwenden, in denen % ausschließlich am Ende der Suchmaske steht.

Beispiel (bezieht sich auf die Datenbank employee.gdb):

```
SELECT LAST_NAME FROM EMPLOYEE WHERE LAST_NAME LIKE 'B_ld_in';
```

Liefert

Baldwin

Genauso wie

```
SELECT LAST_NAME FROM EMPLOYEE WHERE LAST_NAME LIKE 'Bal%';
```

Der IN-Operator

Der IN-Operator erlaubt eine abkürzende Schreibweise für mit OR verknüpfte Gleichheits-Vergleiche.

Beispiel (bezieht sich auf die Datenbank employee.gdb):

Die SELECT-Anweisung

```
SELECT
    LAST_NAME
FROM EMPLOYEE WHERE JOB_CODE IN ('Sales', 'Mktg');
```

liefert die drei Treffer

Johnson

Baldwin

Reeves

In der Aufzählung dürfen nur Konstanten gleichen Typs enthalten sein. Doppeltes Vorkommen ist ausgeschlossen, macht aber ohnehin keinen Sinn.

Sortieren von Abfrageergebnissen

Zum Sortieren von Abfrageergebnissen dient die `ORDER BY` Klausel. Wir haben die Syntax schon bei der Besprechung der `SELECT`-Anweisung kennen gelernt:

```
order-clause      ::=  ORDER BY order-list {, order-list}

order-list        ::=  column-reference {, column-reference}
                    [ASC[ENDING] | DESC[ENDING]]

column-reference  ::=  column-name | column-number
```

Es können mehrere Sortierfelder angegeben werden. Die Sortierung erfolgt dann mit dem ersten Feld beginnend bis zum letzten wie bei der lexikographischen Sortierung. Sortierfelder können über ihre Namen oder die Nummer ihrer Spalte referenziert werden. Achtung: Die Nummerierung beginnt mit 1. Auf- und absteigende Sortierung können beliebig gemischt werden. Ohne Angabe wird aufsteigend sortiert.

Das Ergebnis einer Sortierung muss auf verschiedenen Plattformen nicht notwendig übereinstimmen. Auf Systemen, die EBCDIC verwenden, kann das Ergebnis anders ausfallen als auf ASCII-basierten. In der ASCII-Kodierung liegen die Ziffern `'0'` bis `'9'` vor den Groß- und Kleinbuchstaben im Bereich `0x30` bis `0x39`. EBCDIC platziert sie im Bereich `0xF0` bis `0xF9` hinter den Klein- und Großbuchstaben. Zeichenketten, die mit Ziffern beginnen, landen in beiden Kodierungen beim Sortieren also an unterschiedlichen Stellen.

Berechnungen in Spalten

In der allgemeinen Form der `SELECT`-Anweisung haben wir die Alternativen

```
| constant [AS new-column-name]
| expression [AS new-column-name]
| function [AS new-column-name]
```

für die selektierten Spalten zugelassen. Welche Möglichkeiten bieten sich hier?

Konstanten in Spalten

Feste Spalten in die Ergebnisse eines `SELECTs` zu mischen, scheint auf den ersten Blick keinen Sinn zu machen. Aber es finden sich bekanntlich für jeden Topf ein Deckel.

Beispiel (UNION kommt später):

```
SELECT DISTINCT
    'C' AS SOURCE,
    CUSTOMER
FROM CUSTOMER
UNION
SELECT DISTINCT
    'D' AS SOURCE,
    DEPARTMENT
FROM DEPARTMENT;
```

Das – leicht verkürzte – Ergebnis dieses SELECTs sieht wie folgt aus:

SOURCE	CUSTOMER
C	Max
C	Mrc. Beauvais
C	Signature Design
D	Consumer Electronics Div.
D	Corporate Headquarters
D	Customer Services

Hier wird durch eine Konstante in einer zusätzlichen Spalte `SOURCE` der Ursprung eines Datensatzes gespeichert. In der durch Vereinigung entstandenen Tabelle ist so immer noch erkennbar, ob der Datensatz aus der Tabelle `CUSTOMER` oder `DEPARTMENT` kam.

Arithmetische Ausdrücke in Spalten

SQL kennt nicht nur Konstanten in Spalten, es sind sogar (fast) beliebige arithmetische Ausdrücke möglich. Zur Verfügung stehen nicht nur die arithmetischen Grundoperationen, auch höhere mathematische Funktionen und Zeichenkettenmanipulationen sind möglich.

Firebird kann sogar um sogenannte User Defined Function (kurz UDFs) erweitert werden. Hierzu muss allerdings eine DLL erstellt werden, mit SQL-Mitteln geht das nicht. Wie die Methoden der Datenbank bekannt gemacht werden, kann man beispielhaft dem Skript `ib_udf.sql` im `examples` Verzeichnis entnehmen. In den Metadaten einer Firebird-Datenbank findet sich in der Tabelle `RDB$FUNCTIONS` eine Liste der unterstützten Funktionen.

Die Standard-Funktionen müssen einer neuen Datenbank mit der Anweisung

```
Isql database-name -i ib_udf.sql -u SYSDBA -p masterkey
```

bekannt gemacht werden. Dies erfolgt nicht automatisch!

Die wichtigsten Methoden zur String-Manipulation sind

- `LTRIM`
- `RTRIM`
- `LOWER`
- `UPPER`
- `STRLEN`
- `SUBSTR`

Die verfügbaren mathematischen Funktionen brauchen sich hinter klassischen Programmiersprachen auch nicht zu verstecken.

Beispiel (SQL kennt π):

```
SELECT DISTINCT (4.0*ATAN(1)) AS PI FROM EMPLOYEE;
```

liefert

π
3.1415926535897932

Tabellenweite Berechnungen

SQL kennt folgende Funktionen für Auswertungen auf Tabellenebene:

- SUM
- AVG
- MIN
- MAX
- COUNT

Das Ergebnis eines SELECTs mit einer tabellenweiten Berechnung ist immer eine Tabelle mit einer Zeile.

Summen- und Durchschnittsbildung

Die Syntax lautet:

```
SELECT func([ALL | DISTINCT] column-name | expression)
        [AS new-column-name] FROM table-name
```

```
func ::= SUM | AVG
```

Bemerkungen:

- Der Spaltentyp muss numerisch sein
- NULL-Werte werden nicht berücksichtigt
- DISTINCT unterdrückt doppelte Werte, ALL ist der Standard

Beispiele:

- Gehaltskosten:

```
SELECT SUM(SALARY) AS PAYROLL FROM EMPLOYEE;
```

liefert

PAYROLL
115522468

- Durchschnittliches Gehalt:

```
SELECT AVG(SALARY) AS AVG_SALARY FROM EMPLOYEE;
```

liefert

AVG SALARY
2750534.952380952425301074981689453125

- Kosten einer 5% Gehaltserhöhung für Normalverdiener:

```
SELECT SUM(SALARY*1.05-SALARY) FROM EMPLOYEE WHERE SALARY < 100000;
```

liefert

SUM
84046.450000000004

Minimum- und Maximumbildung

Die Syntax lautet:

```
SELECT func(column-name | expression)
      [AS new-column-name] FROM table-name;
```

Func ::= MIN | MAX;

Bemerkung:

- Der Spaltentyp muss numerisch oder alphanumerisch sein.

Beispiel:

Gehaltsspanne:

```
SELECT
      MIN(SALARY) ,
      MAX(SALARY)
FROM EMPLOYEE;
```

liefert

MIN	MAX
22935	99000000

Zeilen zählen

Die Syntax lautet:

```
SELECT COUNT(*|column-name) [AS new-column-name] FROM table-name;
```

Bemerkungen:

- * zählt alle Zeilen
- Wird ein Spaltenname angegeben, zählen Zeilen, die dort den NULL-Wert enthalten, nicht

Beispiele:

- Anzahl der Mitarbeiter:

```
SELECT COUNT (*) FROM EMPLOYEE;
```

liefert

COUNT
42

- Anzahl der Mitarbeiter mit externer Telefonnummer:

```
SELECT COUNT (PHONE_EXT) FROM EMPLOYEE;
```

liefert

COUNT
39

es gibt also offensichtlich drei Mitarbeiter, die keine externe Telefonnummer hinterlegt haben.

Gruppenverarbeitung

Die Gruppierung von Selektionsergebnissen dient nicht nur der optischen Aufbereitung; dies wäre bereits durch die `ORDER BY`-Klausel möglich. Auf Gruppen können zusätzlich Funktionen angewendet werden, deren Ergebnisse dann gruppenbezogen ermittelt werden. Die allgemeine Syntax lautet:

```
GROUP BY column-name {, column-name} [HAVING search-condition]
```

Bemerkungen:

- Anders als bei `ORDER BY` ist die Angabe von Spaltennummern nicht zulässig
- In Kombination mit Funktionen werden diese auf der niedrigsten (innersten) Gruppenwechselebene ausgewertet

Beispiele:

- ```
SELECT JOB_COUNTRY, SUM(SALARY) FROM EMPLOYEE GROUP BY JOB_COUNTRY;
```

Liefert die Gehaltskosten für jedes Land.

| JOB_COUNTRY | SUM          |
|-------------|--------------|
| Canada      | 100914       |
| England     | 95779.6875   |
| France      | 390500       |
| Italy       | 99000000     |
| Japan       | 13480000     |
| Switzerland | 110000       |
| USA         | 2345274.3125 |

- ```
SELECT JOB_COUNTRY, JOB_CODE, SUM(SALARY) FROM EMPLOYEE
WHERE JOB_COUNTRY='England' GROUP BY JOB_COUNTRY, JOB_CODE;
```

liefert die Gehaltskosten für England, gruppiert nach Berufsgruppen.

JOB_COUNTRY	JOB_CODE	SUM
England	Admin	22935
England	Eng	39224.0625
England	Sales	33620.625

Die HAVING-Klausel zur Gruppenselektion

Die HAVING-Klausel ist nicht mit der WHERE-Klausel zu verwechseln. Diese wird auf den gesamten Datenbestand angewendet. Die HAVING-Klausel hingegen selektiert zur Bedingung passende Gruppen.

Beispiele:

- ```
SELECT
 JOB_COUNTRY,
 SUM(SALARY)
FROM EMPLOYEE
WHERE SALARY < 100000
GROUP BY JOB_COUNTRY;
```
- ```
SELECT
    JOB_COUNTRY,
    SUM(SALARY)
FROM EMPLOYEE
GROUP BY JOB_COUNTRY
HAVING SALARY < 100000;
```

Wo liegt der Unterschied zwischen beiden SQLs? Zunächst ein Blick auf die Ergebnisse:

JOB_COUNTRY	SUM
England	95779.6875
USA	1585149.3125

bzw.

JOB_COUNTRY	SUM
ENGLAND	95778.6875

Das Ergebnis der zweiten Query enthält keine Zeile für die USA. Wieso nicht? Es gibt dort Angestellte, die mehr als 100.000\$ verdienen, wie folgende Query zeigt:

```
SELECT DISTINCT
    JOB_COUNTRY
FROM EMPLOYEE WHERE SALARY >= 100000;
```

Ergebnis:

JOB_COUNTRY
Canada
France
Italy
Japan
Switzerland
USA

Wie wir bereits früher gesehen haben, beträgt die Gehaltssumme in den USA 2345274.3125\$. Dort liegt die Summe der Gehälter unter 100000 bei 1585149.3125\$, die Differenz liegt bei 760125\$, wie folgende Abfrage bestätigt:

```
SELECT
    SUM(SALARY)
FROM EMPLOYEE
WHERE SALARY >= 100000 AND JOB_COUNTRY='USA'
GROUP BY JOB_COUNTRY;
```

Die HAVING-Klausel sorgt also dafür, dass die USA im zweiten Ergebnis nicht mehr erscheint, da nicht alle dort Beschäftigten unter 100000\$ verdienen. Für England ist der Wert in beiden Abfragen identisch, da dort niemand mehr als 100000£ verdient.

Joins in SQL-Abfragen

Inner Joins

Wir haben Beispielhaft bereits eine Form des Inner-Joins kennengelernt. Genauer handelte es sich dabei um einen Equi-Join. Die Syntax lautete:

```
SELECT
    ...
FROM TABLE1, TABLE2
WHERE TABLE1.COLUMN_NAME1 = TABLE2.COLUMN_NAME2;
```

Dabei wird zunächst das kartesische Produkt von TABLE1 und TABLE2 gebildet und anschließend die Zeilen selektiert, für die die Join-Bedingung TABLE1.COLUMN_NAME1 = TABLE2.COLUMN_NAME2 erfüllt ist. Diese Form eines Joins ist nicht auf zwei Tabellen beschränkt. Bei mehr als zwei Tabellen ist allerdings Vorsicht geboten, da wegen der großen intern zu erzeugenden Hilfstabellen die Laufzeit schnell (exponentiell) ansteigen kann. Es existiert eine weitere syntaktische Möglichkeit einen Inner-Join zu formulieren:

```
SELECT
    ...
FROM TABLE1 [INNER] JOIN TABLE2
ON TABLE1.COLUMN_NAME1 = TABLE2.COLUMN_NAME2;
```

Es ist zulässig, in einem Join die Gleiche Tabelle mehrmals zu verwenden (Auto-Join). Dann muss, um Mehrdeutigkeiten zu vermeiden, allerdings mit Tabellen-Aliasen gearbeitet werden.

Beispiel: Alle Departments mit Head Departments

```
SELECT
    T1.DEPARTMENT, T2.DEPARTMENT AS HEAD_DEPARTMENT
FROM DEPARTMENT T1 JOIN DEPARTMENT T2
ON T1.HEAD_DEPT=T2.DEPT_NO;
```

In der obigen Liste fehlt jetzt natürlich 'Corporate Headquarters', da sie kein Head Department besitzen.

Ein Join kann sich nicht nur auf Tabellen beziehen. Es sind auch weitere verbundene Tabellen sowie Views erlaubt.

Beispiel: Wie oben, bloß ein Level mehr

```
SELECT
    T1.DEPARTMENT,
    T2.DEPARTMENT,
    T3.DEPARTMENT
FROM DEPARTMENT T1
JOIN DEPARTMENT T2 ON T2.HEAD_DEPT=T2.DEPT_NO
JOIN DEPARTMENT T3 ON T2.HEAD_DEPT = T3.DEPT_NO;
```

Dies kann auch wie folgt ausgedrückt werden:

```
SELECT
    T1.DEPARTMENT,
    T2.DEPARTMENT,
    T3.DEPARTMENT
FROM DEPARTMENT T1, DEPARTMENT T2, DEPARTMENT T3
WHERE T1.HEAD_DEPT=T2.DEPT_NO AND T2.HEAD_DEPT=T3.DEPT_NO;
```

Outer-Joins

Im vorherigen Abschnitt haben wir festgestellt, dass die Liste der Departments und ihrer Head Departments 'Corporate Headquarters' nicht enthält. Dies liegt an dem verwendeten Inner Join. Bei der theoretischen Betrachtung von Verbünden im relationalen Modell haben wir gelernt, dass hier Outer Joins Abhilfe schaffen können. Die Syntax lautet:

```
SELECT
    ...
FROM TABLE1 outer-join-type OUTER JOIN TABLE2
ON TABLE1.COLUMN_NAME1 = TABLE2.COLUMN_NAME2;
```

outer-join-type ::= LEFT | RIGHT | FULL

Beispiel: Alle Departments mit ihren Head Departments

```
SELECT
    T1.DEPARTMENT,
    T2.DEPARTMENT
FROM DEPARTMENT T1 LEFT OUTER JOIN DEPARTMENT T2
ON T1.HEAD_DEPT = T2.DEPT_NO;
```

Wir haben Left Outer Join verwendet, da wir alle Einträge der ersten (linken) Tabelle sehen wollen, auch wenn es keinen korrespondierenden Eintrag – also kein Head Department – in der zweiten Tabelle gibt.

Vereinigung in SQL-Abfragen

Um die mengentheoretische Vereinigung von Tabellen und Abfrageergebnissen in SQL formulieren zu können, gibt es die UNION-Anweisung. Die allgemeine Syntax lautet:

```
select-statement  
UNION [ALL] select-statement {, UNION [ALL] select-statement}
```

Bemerkungen:

- Die Ergebnisse der beteiligten Abfragen müssen Union-kompatibel sein
- Das optionale ALL sorgt dafür, dass doppelte Zeilen nicht unterdrückt werden
- ORDER BY ist nur hinter der letzten Abfrage zulässig, Sortierspalten müssen über ihren Index spezifiziert werden
- Für die Benennung der Ergebnisspalte ist die erste Abfrage ausschlaggebend

Beispiele:

- Gemischte Telefonliste für Abteilungen und Kunden in Tokyo:

```
SELECT  
    CUSTOMER,  
    PHONE_NO,  
    'C'  
FROM CUSTOMER WHERE CITY='Tokyo'  
UNION ALL  
SELECT  
    DEPARTMENT,  
    PHONE_NO,  
    'D'  
FROM DEPARTMENT WHERE LOCATION='Tokyo'  
ORDER BY 1;
```

Ergebnis:

CUSTOMER	PHONE_NO	
Field Office:Japan	3 5350 0901	D
MPM Corporation	3 880 77 19	C

- Gehaltsliste mit Summe als letzter Zeile:

```
SELECT
    LAST_NAME,
    FIRST_NAME,
    SALARY
FROM EMPLOYEE
UNION ALL
SELECT
    CAST (NULL AS VARCHAR(20)),
    CAST('Payroll:' AS VARCHAR(15)),
    SUM(SALARY)
FROM EMPLOYEE;
```

Ergebnis (nur die beiden letzten Zeilen und die Spaltenüberschriften):

LAST_NAME	FIRST_NAME	SALARY
...	...	...
Guckenheimer	Mark	32000.0
	Payroll:	1.15522468E8

UNIONS können auch eingesetzt werden, um Outer Joins zu simulieren. Als Beispiel noch einmal die Liste aller Abteilungen mit ihren Managern:

```
SELECT
    DEPARTMENT,
    LAST_NAME,
    FIRST_NAME
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.EMP_NO = D.MNGR_NO
UNION
SELECT
    DEPARTMENT,
    CAST(NULL AS VARCHAR(20)),
    CAST(NULL AS VARCHAR(15))
FROM DEPARTMENT WHERE MNGR_NO IS NULL;
```

Unterabfragen

Einige Abfragen lassen sich in SQL nicht so einfach realisieren, wie das auf den ersten Blick erscheint.

Beispiel: (welcher Mitarbeiter hat das höchste Gehalt?)

```
SELECT
    LAST_NAME,
    FIRST_NAME,
    MAX(SALARY)
FROM EMPLOYEE;
```

sieht gut aus, ist aber kein gültiges SQL. Die tabellenweite Funktion `MAX` hat ein einzeliges Ergebnis, welches nicht mit mehrzeiligen kombiniert werden darf. Wie geht es dennoch?

Lösung:

- Suche das maximale Gehalt
- Suche den Mitarbeiter mit diesem Gehalt

Mit einer Unterabfrage kann dies wie folgt realisiert werden:

```
SELECT
    LAST_NAME,
    FIRST_NAME,
    SALARY
FROM EMPLOYEE
WHERE SALARY = (SELECT MAX(SALARY) FROM EMPLOYEE);
```

(Unter-)Abfragen lassen sich nach ihrer Struktur und Anzahl der Treffer in drei Typen unterteilen:

- **Typ I (singleton select):**
Abfragen, die genau einen Wert ergeben (SELECT auf eine Spalte mit einem Treffer)
- **Typ II (list select):**
Abfragen, die beliebig viele Werte ergeben (SELECT auf eine Spalte mit beliebiger Trefferzahl)
- **Typ III (arbitrary select):**
Sonstige Abfragen (SELECT auf mehrere Spalten mit beliebiger Trefferzahl)

Unterabfragen sind in der WHERE- oder HAVING-Klausen in folgenden Varianten möglich:

```
comparison-operator (singleton_select)
| comparison-operator comparison-qualifier (list-select)
| [NOT] EXISTS (arbitrary-select)
```

```
comparison-qualifier ::= ANY | ALL | SOME
```

Ein Singleton Select ist auch in der Feldliste einer SELECT-Anweisung zulässig. Auf diese Weise kann zum Beispiel das Ergebnis einer tabellenweiten Funktion mehrmals in einem Resultat vorkommen. **Beispiel:**

```
SELECT
    LAST_NAME,
    FIRST_NAME,
    (SELECT MIN(SALARY) FROM EMPLOYEE) AS MIN_SALARY
    SALARY
    (SELECT MAX(SALARY) FROM EMPLOYEE) AS MAX_SALARY
FROM EMPLOYEE WHERE JOB_COUNTRY = 'England';
```

Ergebnis:

LAST_NAME	FIRST_NAME	MIN_SALARY	SALARY	MAX_SALARY
Bennet	Ann	22935	22935	99000000
Reeves	Roger	22935	33620.625	99000000
Stansbury	Willie	22935	39224.0625	99000000

Beispiele für die verschiedenen Varianten:

- Wie lauten die Nach- und Vornamen der Mitarbeiter, die in der gleichen Abteilung wie Roger Reeves arbeiten?

```
SELECT
    LAST_NAME,
    FIRST_NAME,
    DEPARTMENT
FROM EMPLOYEE JOIN DEPARTMENT
ON EMPLOYEE.DEPT_NO = DEPARTMENT.DEPT_NO
WHERE EMPLOYEE.DEPT_NO = (
    SELECT
        DEPT_NO
    FROM EMPLOYEE
    WHERE LAST_NAME='Reeves' AND FIRST_NAME='Roger');
```

Ergebnis:

LAST_NAME	FIRST_NAME	DEPARTMENT
Bennet	Ann	European Headquarters
Reeves	Roger	European Headquarters
Stansbury	Willie	European Headquarters

- Wie lauten die Nach- und Vornamen der Vertreter, die Verkäufe mit ausstehender Bezahlung getätigt haben?

```
SELECT
    LAST_NAME,
    FIRST_NAME
FROM EMPLOYEE E WHERE JOB_CODE = 'SREP'
AND EXISTS (
    SELECT
        SALES_REP
    FROM SALES
    WHERE SALES_REP=E.EMP_NO AND PAID != 'y');
```

Ergebnis:

LAST_NAME	FIRST_NAME
Sutherland	Claudia
Yamamoto	Takashi
Osborne	Pierrre

- Welche Abteilungen finanzieren überhaupt Projekte?

```
SELECT
    DEPARTMENT
FROM DEPARTMENT
WHERE DEPT_NO IN (SELECT DEPT_NO FROM PROJ_DEPT_BUDGET);
```

- Länder, in denen kein Job ausgeübt wird:

```
SELECT
    C.COUNTRY
FROM COUNTRY C
WHERE NOT EXISTS (
    SELECT
        J.JOB_CODE,
        J.JOB_COUNTRY
    FROM JOB J
    WHERE J.JOB_COUNTRY = C.COUNTRY);
```

- Alle Mitarbeiter, die keine Manager sind:

```
SELECT
    LAST_NAME,
    FIRST_NAME
FROM EMPLOYEE
WHERE EMP_NO NOT IN (SELECT MNGR_NO FROM DEPARTMENT);
oder alternativ:
```

```
SELECT
    LAST_NAME,
    FIRST_NAME
FROM EMPLOYEE
WHERE EMP_NO != ALL (
    SELECT
        MNGR_NO
    FROM DEPARTMENT
    WHERE NOT MNGR_NO IS NULL);
```

Analog kann „=ANY“ als Alternative zu „IN“ verwendet werden.

- Alternative für die Ermittlung des Mitarbeiters mit dem höchsten Gehalt:

```
SELECT
    LAST_NAME,
    FIRST_NAME
FROM EMPLOYEE
WHERE SALARY >= ALL (SELECT SALARY FROM EMPLOYEE);
```

Wir haben gesehen, dass eine Unterabfrage auswärtige Spalten referenzieren kann. Dies wird als „outer reference“, die Unterabfrage dann auch als „unabhängige Abfrage“ oder „correlated subquery“ bezeichnet. Abhängige Abfragen müssen für jede Zeile des Ergebnisses der Hauptabfrage ausgeführt werden. Dies kann zu Performanceproblemen führen! Unabhängige Abfragen hingegen müssen nur einmal ausgeführt werden.

Zusammengefasst noch einmal die wichtigsten Regeln zur Bildung von Subqueries:

- Eine Subquery steht in runden Klammern
- Eine Subquery ist immer rechtsseitiger Term einer Vergleichs- oder EXISTS-Bedingung
- Außer bei EXISTS darf im SELECT einer Subquery nur ein Ausdruck/eine Spalte stehen
- Dieser hat den gleichen Typ wie der linksseitige Term
- Beim Vergleich darf das Ergebnis der Unterabfrage nur eine Zeile liefern
- Bei IN, ANY, ALL, SOME und EXISTS ist ein mehrzeiliges Ergebnis zulässig
- ORDER BY ist für Subqueries unzulässig
- UNIONs sind als Subqueries nicht zulässig

Der SQL92-Standard erlaubt Subqueries auch an allen Stellen, an denen Tabellen zulässig sind. Hier hat – nicht nur – Firebird (MySQL 4) leider ein Defizit, denn dies wird nicht unterstützt. Kommerzielle Konkurrenten wie DB2 von IBM und Oracle bieten dieses Feature jedoch. Es erlaubt die effizientere Formulierung vieler Joins, da WHERE-Bedingungen nach „innen“ gezogen werden können.

SQL zur Datenmanipulation

Alle SQL-Anweisungen zur Datenmanipulation geben keine Daten zurück, sie dienen ausschließlich zur Änderung bestehender Daten. Die möglichen Änderungen umfassen

- Das Einfügen einer oder mehrerer neuer Zeilen in eine Tabelle
- Das Ändern einer oder mehrerer bestehender Zeilen einer Tabelle
- Das Löschen von einer oder mehrerer Zeilen aus einer Tabelle

Einfügen von Daten in Tabellen

Zum Anfügen von Datensätzen an bestehende Tabellen dient das INSERT-Statement. Es existiert in verschiedenen syntaktischen Varianten.

```
INSERT INTO objekt [(column-name {, column-name})] data-values;
```

```
object      ::= table-name | view-name
```

```
data-values ::= VALUES (value {, value}) | select_expr
```

```
value       ::= constant | expression | function
```

```
constant    ::= num | 'string'
```

```
function    ::= CAST(value AS data-type)
```

Bemerkungen:

Es ist unter bestimmten Bedingungen möglich, nicht nur in Tabellen, sondern sogar in Views Daten einzufügen. Zur Erinnerung: Views sind nur Sichten auf eine oder mehrere Tabellen. Sie enthalten selbst keine Daten. Diese Option wird in der Praxis allerdings selten genutzt, wir werden sie deshalb nicht näher besprechen.

Die Angabe der Spalten, in die die Werte eingefügt werden sollen, ist optional. Wird diese Liste weggelassen, muss für jede Spalte der Tabelle ein Wert angegeben werden, ggf. NULL. Die Reihenfolge ist durch die Reihenfolge der Spalten beim Anlegen der Tabelle vorgegeben. Wird auf der anderen Seite eine Liste von Spalten angegeben, dürfen nur für diese Werte angegeben werden! Spalten, die nicht leer bleiben dürfen, müssen natürlich gefüllt werden! Nicht explizit gefüllte Spalten werden mit dem NULL-Wert belegt.

Wird für die einzufügenden Daten die `VALUES`-Notation verwendet, kann nur ein Datensatz zur Zeit eingefügt werden.

Wird eine `SELECT`-Anweisung zum Füllen der einzufügenden Zeilen verwendet, können mehrere Zeilen auf einmal eingefügt werden. Für das Resultat der `SELECT`-Anweisung gelten die gleichen Einschränkungen wie bei der `VALUES`-Notation.

Die meisten RDBMS unterstützen eine Reihe von symbolischen Ausdrücken, die (nicht nur) beim Einfügen von Daten nützlich sind. Unter anderem gibt es

- `CURRENT TIME`
- `CURRENT DATE`
- `CURRENT TIMESTAMP`

um die aktuelle Zeit, das aktuelle Datum und einen aktuellen Zeitstempel zu erzeugen. Hierbei werden jeweils die beim Datenbank-Server zur Ausführungszeit geltenden Werte eingesetzt.

Achtung: Zeitzonen können bei verteilten Anwendungen zu einem nicht versiegenden Quell der Freude werden!

Besonderheiten bei Firebird:

Firebird unterstützt diese Konstanten leider nicht, es gibt aber zum Teil Umgehungs-lösungen:

- `CAST(, now' as DATE)` kann statt `CURRENT DATE` und
- `CAST('now' as TIMESTAMP)` statt `CURRENT TIMESTAMP`

Verwendet werden. Für den SQL-Datentyp `TIME`, also die Konstante `CURRENT TIME` geht das leider nicht.

Auch den `COUNTER`-Datentyp aus Access werden Sie in Firebird vermissen. Die Aufgabe, für eine Spalte beim Einfügen fortlaufende Nummern zu vergeben, muss in Firebird mit einem sogenannten Generator gelöst werden. Die Anweisungen:

```
CREATE GENERATOR EMP_NO_GEN;  
SET GENERATOR EMP_NO_GEN TO 0;  
  
... GEN_ID(EMP_NO_GEN, 1) ...
```

Definieren einen Generator namens `EMP_NO_GEN`, der sich den letzten ausgegebenen Wert merkt und diesen bei jedem Aufruf der Form `GEN_ID(EMP_NO_GEN, 1)` um Eins erhöht und den neuen Wert zurückgibt. Obiger Generator würde also Eins als ersten Wert ausgeben. Der Zweite Parameter der `GEN_ID`-Funktion legt die Schrittweite fest.

Die bereits definierten Generatoren finden sich in der Systemtabelle `RDB$GENERATORS` und müssen dort explizit mit `DELETE` gelöscht werden, ein `DROP` für Generatoren gibt es nicht.

Beispiele:

- Backup einer Tabelle

```
CREATE TABLE COUNTRY_BAK
(
    COUNTRY          VARCHAR(15) NOT NULL PRIMARY KEY,
    CURRENCY          VARCHAR(10) NOT NULL
);
```

```
INSERT INTO COUNTRY_BAK SELECT * FROM COUNTRY;
```

erstellt ein Backup der Tabelle COUNTRY namens COUNTRY_BAK.

- Einfügen eines weiteren Landes

```
INSERT INTO COUNTRY(COUNTRY, CURRENCY) VALUES('Spain', 'Pesetas');
```

oder kürzer

```
INSERT INTO COUNTRY VALUES('Portugal', 'Escudos');
```

- Aktualisieren der Backup-Tabelle

```
INSERT INTO COUNTRY_BAK
SELECT * FROM COUNTRY C
WHERE NOT EXISTS (
    SELECT * FROM COUNTRY_BAK CB WHERE C.COUNTRY=CB.COUNTRY);
```

Ändern von Datensätzen

Das Ändern von Datensätzen in Tabellen erfolgt über das UPDATE-Statement. Die Syntax lautet:

```
UPDATE object SET [column-name]=value, {, [column-name]=value}
[WHERE search-condition];
```

object ::= table-name | view-name

value ::= constant | expression | function

constant ::= num | 'string'

function ::= CAST(value AS data-type)

Achtung: Alle Zeilen, auf die die WHERE-Bedingung zutrifft, werden aktualisiert! Es empfiehlt sich dringend, die Korrektheit der Suchbedingung vorher zu überprüfen. Wird keine WHERE-Bedingung angegeben, werden alle Zeilen aktualisiert.

Beispiel: Euro-Einführung

```
UPDATE COUNTRY_BAK
SET CURRENCY='Euro'
WHERE COUNTRY IN ('Italy', 'France', 'Germany', 'Netherlands', 'Belgium',
'Austria', 'Portugal', 'Spain');
```

In der Realität müssten natürlich ebenfalls alle betroffenen Geldbeträge umgerechnet werden.

Löschen von Datensätzen

Das Löschen von Datensätzen erfolgt mit Hilfe des `DELETE`-Statements. Syntax:

```
DELETE FROM table-name [WHERE search-condition];
```

Bemerkungen:

- Ohne eine `WHERE`-Klausel werden alle Datensätze gelöscht. Die Tabellendefinition bleibt jedoch erhalten
- `DELETE` löscht ganze Sätze, zum Löschen einzelner Felder muss `UPDATE` verwendet werden
- `DELETE` löscht unwiderruflich, also die `WHERE`-Bedingung vorher testen!

Beispiel: Euro-Umstellung in `COUNTRY_BAK` rückgängig machen

```
DELETE FROM COUNTRY_BAK WHERE CURRENCY='Euro';
INSERT INTO COUNTRY_BAK
SELECT * FROM COUNTRY C
WHERE NOT EXISTS (SELECT * FROM COUNTRY_BAK CB WHERE C.COUNTRY=CB.COUNTRY);
```

SQL zur Administration von Datenbank-Zugriffsrechten

SQL kennt zur Administration von Datenbank-Zugriffsrechten standardmäßig Benutzer (Users) und Berechtigungen (Privileges).

Die verfügbaren Berechtigungen sind:

- `SELECT`
- `INSERT`
- `UPDATE`
- `DELETE`
- `REFERENCES`

Bei der Installation des Datenbanksystems wird in der Regel ein initialer Benutzer angelegt, dessen Passwort entweder direkt abgefragt wird oder schleunigst zu ändern ist! („`SYSDBA`“ und „`masterkey`“ bei Firebird).

Wie die Benutzer angelegt werden ist vom verwendeten RDBMS abhängig, einige Systeme können die im Wirtsbetriebssystem bekannten Benutzer verwenden (häufig unter UNIX-artigen Systemen anzutreffen), andere verwalten die Benutzer in einer eigenen Datenbank („`isc4.gdb`“ bei Firebird).

Dementsprechend unterscheiden sich auch die Tools zur Anlage neuer Benutzer, bei Firebird heißt es „`gsec.exe`“ und findet sich im `bin`-Verzeichnis der Firebird-Installation. Kommen Sie nicht auf die Idee, neue Benutzer selbst in der Datenbank „`isc4.gdb`“ anzulegen, das Passwort wird nicht im Klartext, sondern als Hashwert gespeichert! Den können Sie nicht selbst berechnen.

Die GRANT-Anweisung

Sind die Datenbank-Benutzer erst einmal eingerichtet, können sie mit Hilfe der GRANT-Anweisung ihre spezifischen Berechtigungen erhalten. Die Syntax lautet:

```
GRANT privileges ON what TO whom [WITH GRANT OPTION]
```

```
privileges ::= SELECT
            | DELETE
            | INSERT
            | UPDATE [(col {,col})]
            | REFERENCES [(col {,col})]
```

```
what ::= [TABLE] tablename | viewname
```

```
whom ::= PUBLIC | user-list
```

```
user-list ::= user {, user}
```

```
user ::= [USER] username | rolename
```

Bemerkungen:

- Der User PUBLIC ist ein Platzhalter für alle bekannten DB-Benutzer; was er darf, dürfen alle
- Die UPDATE-Berechtigung kann spalten-spezifisch vergeben werden, wichtig z.B. bei Gehaltserhöhungen ;-)
- Die Klausel WITH GRANT OPTION erlaubt dem Benutzer, seine Berechtigungen an andere weiterzugeben
- Rollen als Mittel zur Administration von Zugriffsrechten kommen später

Beispiele:

- Einen Benutzer BOFH auf alles berechtigen

```
GRANT ALL PRIVILEGES ON COUNTRY TO BOFH;
GRANT ALL PRIVILEGES ON CUSTOMER TO BOFH;
...
GRANT ALL PRIVILEGES ON SALES TO BOFH;
```

- Einen Benutzer DAU ausschließlich zum Lesen berechtigen

```
GRANT SELECT ON COUNTRY TO DAU;
GRANT SELECT ON CUSTOMER TO DAU;
...
GRANT SELECT ON SALES TO DAU;
```

Wie man hier bereits sieht, kann die Verwaltung vieler unterschiedlich berechtigter Benutzer erheblichen Aufwand verursachen. Aus diesem Grund unterstützen viele RDBMS Rollen als abstraktes Konzept der Verwaltung von Zugriffsberechtigungen (auch Firebird).

Die REVOKE-Anweisung

Berechtigungen können mit dem REVOKE-Befehl natürlich auch wieder entzogen werden. Die Syntax lautet:

```
REVOKE [GRANT OPTIONS FOR] privileges ON what FROM whom
```

```
privileges      ::= ALL [PRIVILEGES] | privilege-list
```

```
privilege-list  ::= privilege {, privilege}
```

```
privileges      ::= SELECT  
                  | DELETE  
                  | INSERT  
                  | UPDATE [(col {, col})]  
                  | REFERENCES [(col {, col})]
```

```
what            ::= [TABLE] tablename | viewname
```

```
whom            ::= PUBLIC | user-list
```

```
user-list       ::= user {, user}
```

```
user            ::= user-name | rolename
```

Transaktionen und Parallelverarbeitung

RDBMS laufen in der Regel weder im Single-User-Modus, noch sind alle abgesetzten Abfragen stets fehlerfrei ausführbar. Im letzten Fall ist Vorsorge zu treffen, dass die Datenbank nicht in einem inkonsistenten Zustand verbleibt. Greifen mehrere Benutzer parallel schreibend und lesend auf die Datenbank zu, entstehen die üblichen Nebenläufigkeitsprobleme („Wer zuletzt schreibt, gewinnt“).

Transaktionen

Zur Sicherung von Datenbanken gegen inkonsistente Zustände nach abgebrochenen Operationen dient das sogenannte Transaktionskonzept. Die Gründe für den Abbruch von Datenbank-Operationen können vielfältig sein, von Programmier- bis zu Hardwarefehlern ist alles drin.

Definition: Eine *Transaktion* ist eine Folge von Datenbank-Operationen, die atomar ist. Das bedeutet, dass die Transaktion entweder ganz oder gar nicht ausgeführt wird.

Beispiel: Gehaltserhöhung in der Firebird-Datenbank `employee.gdb`

Zu diesem Zweck sind zwei SQL-Anweisungen erforderlich:

- `INSERT` eines Satzes in die Tabelle `SALARY_HISTORY`
- `UPDATE` eines Satzes in der Tabelle `EMPLOYEE`

Die zweite Anweisung kann zum Beispiel scheitern, falls der Benutzer keine `UPDATE`-Berechtigung auf die Spalte `SALARY` besitzt. Die beiden Anweisungen müssen aber gemeinsam ausgeführt werden, da sonst die Datenbank in einen fachlich inkonsistenten Zustand geraten könnte. Falls ein Angestellter bereits eine Gehaltserhöhung bekommen hat, muss der Wert in der Spalte `NEW_SALARY` im jüngsten Eintrag für diesen Angestellten mit seinem aktuellen Gehalt in der Spalte `SALARY` der Tabelle `EMPLOYEE` übereinstimmen.

Damit das RDBMS eine Transaktion auch als solche erkennt, müssen Anwendungsprogramme Datenbank-Operationen nach folgendem Schema durchführen:

- Einleiten einer Transaktion
- Absetzen der zur Transaktion gehörenden Statements
- `COMMIT` im Erfolgsfall
- `ROLLBACK` im Falle eines Fehlers

Was passiert intern beim Abarbeiten einer Transaktion? Die meisten RDBMS führen ein sogenanntes Journal, in dem der Zustand eines Datensatzes vor und nach einer Änderung – zusammen mit Zeitstempel, Benutzerkennung und Transaktions-ID – festgehalten wird. Beim Einfügen oder Löschen gibt es selbstverständlich nur einen neuen bzw. alten Zustand. Nach der erfolgreichen Ausführung wird der `COMMIT` ebenfalls im Journal vermerkt und alles ist in Ordnung. Im Fehlerfall wird ein `ROLLBACK` ausgeführt, der in umgekehrter Reihenfolge alle für diese Transaktion im Journal hinterlegten Sätze bis zum letzten `COMMIT` abarbeitet und den ursprünglichen Zustand wiederherstellt. Auch ein `ROLLBACK` wird natürlich im Journal vermerkt.

In der Tat sind die Dinge noch komplizierter, da bei der Durchführung eines `ROLLBACKs` ebenfalls Fehler auftreten können, die zum `ROLLBACK` des `ROLLBACKs` führen müssen.

Die Eigenschaften einer Transaktion werden häufig mit dem Akronym ACID zusammengefasst. Es steht für die Anfangsbuchstaben der Begriffe **A**tomicity, **C**onsistency, **I**solation und **D**urability.

Atomocity (Atomizität) bedeutet, dass eine Transaktion atomar ist, also ganz oder gar nicht ausgeführt wird.

Consistency (Konsistenz) bedeutet, dass eine Transaktion eine Datenbank von einem konsistenten in einen konsistenten Zustand überführt. Dies kann im Falle eines `ROLLBACKs` auch der Ausgangszustand sein.

Isolation bedeutet, dass Transaktionen in einem simulierten Single-User-Modus ablaufen und sich nicht gegenseitig beeinflussen.

Durability (Dauerhaftigkeit) bedeutet, dass nach einem `COMMIT` die Wirkung einer Transaktion dauerhaft ist.

Parallelverarbeitung

Im vorherigen Abschnitt ist bereits erwähnt worden, dass eine Transaktion isoliert ausgeführt werden muss. Dies wird in der Regel durch sogenannte *Locks* gewährleistet. Locks können nach

- *Granularität* und
- *Exklusivität*

klassifiziert werden. Die unterschiedlichen Granularitäten entsprechen den Lock-Ebenen

- Datenbank
- Tabelle
- Page
- Tupel
- Feld

In der Praxis sind die Level Page und Tupel (Row) am häufigsten anzutreffen, da alle anderen entweder zu grob oder unperformant sind. Für die geordnete Ausführung von Transaktionen sind eigentlich nur Row-Level-Locks erforderlich, die aber bei einem hohen Transaktionsaufkommen die Ausführungsgeschwindigkeit stark beeinträchtigen können.

Bei der Exklusivität werden

- exklusive und
- verteilte (shared)

Locks unterschieden. Exklusive und shared Locks dienen dazu, konkurrierende Transaktionen zu serialisieren. Exklusive Locks verhindern, dass weitere Locks – egal ob exklusiv oder shared – auf dieselbe Einheit erfolgen können. Shared Locks verhindern nur weitere exklusive Locks, aber keine weiteren shared Locks.

Beispiel: Zwei konkurrierende Versuche, demselben Mitarbeiter eine Gehaltserhöhung zu gönnen.

Ohne Vorsichtsmaßnahmen ist folgender Ablauf denkbar:

- T1 legt ihren Satz in `SALARY_HISTORY` an
- T2 legt ihren Satz in `SALARY_HISTORY` an
- T2 ändert das aktuelle Gehalt in `EMPLOYEE`
- T1 ändert das aktuelle Gehalt in `EMPLOYEE`

Das Resultat ist eine inkonsistente Datenbank, da das aktuelle Gehalt nicht mehr mit dem zugehörigen Eintrag der letzten Gehaltserhöhung übereinstimmt. Wie kann das verhindert werden?

- T1 muss vor der Ausführung einen exklusiven Lock auf den `EMPLOYEE`-Satz (oder noch feiner aber unperformanter auf das Feld `SALARY`) anfordern und darf erst starten, wenn dies erfolgreich war
- Versucht jetzt T2, ebenfalls einen exklusiven Lock auf denselben Satz zu erhalten, scheitert dies und T2 bricht entweder ab oder wartet, bis T1 den Lock wieder aufgehoben hat

- Nach erfolgreichem UPDATE muss T1 den Lock natürlich wieder aufheben

Die schwächeren Shared Locks sind immer dann erforderlich, wenn eine Transaktion nur selend auf Daten zugreift, aber sichergestellt werden muss,

- dass diese Daten während ihres Ablaufs nicht geändert werden dürfen oder
- Änderungen noch nicht committed wurden und deshalb durch einen Rollback eventuell wieder zurückgesetzt werden können

Beispiel: Liste aller Gehälter mit Summe der letzten Zeile gefolgt von der Summe

Hier ist folgendes Vorgehen sicher:

- Die Transaktion fordert einen Shared Lock auf die EMPLOYEE Tabelle an
- Der Display-Select wird ausgeführt
- Die Summation wird ausgeführt
- Der Shared Lock wird aufgehoben

Jetzt kann das Ergebnis keine Widersprüche enthalten, da eine ändernde Transaktion ja einen exklusiven Lock erhalten müsste. Dies wird durch den Shared Lock jedoch bis zum Ende der Transaktion verhindert. Durch den Shared Lock wird gleichzeitig verhindert, dass ein neuer Satz eingefügt werden kann sofern die einfügende Transaktion dies unter Verwendung eines exklusiven Locks tut.

Bei paralleler Verarbeitung können insgesamt drei Erscheinungen auftreten:

- *Dirty-Read (schmutziges Lesen):* T1 ändert eine Zeile, T2 liest danach diese Zeile vor dem COMMIT. T1 führt einen ROLLBACK durch. Die Zeile von T2 ist unsauber, da so nie dauerhaft in der Datenbank vorhanden
- *Non-Repeatable-Read (Nicht wiederholbares Lesen):* T1 liest eine Zeile, T2 ändert oder löscht diese Zeile mit einem COMMIT. Erneutes lesen von T1 führt zu einer geänderten oder gar keiner Zeile.
- *Phantom:* T1 liest n Zeilen, die einer bestimmten Suchbedingung genügen. T2 führt parallel INSERT-, UPDATE- und DELETE-Anweisungen aus, die die Ergebnismenge von T1 betreffen. Liest T1 jetzt mit der gleichen Suchbedingung, ändert sich das Ergebnis.

SQL unterstützt insgesamt vier verschiedene sogenannte Isolationsgrade (Isolation level 0...3), die diese Phänomene verhindern:

	Dirty-Read	Non-Repeatable-Read	Phantom
Isolation Level 0	+	+	+
Isolation Level 1	-	+	+
Isolation Level 2	-	-	+
Isolation Level 3	-	-	-

Deadlocks

Wann immer Locks verwendet werden, besteht die Gefahr, Deadlocks zu erzeugen. Diese entstehen durch Zyklen, in denen Transaktionen jeweils darauf warten, dass andere Locks wieder freigeben. Das Erkennen solcher Zustände liegt in der Verantwortlichkeit des RDBMS. Aufgehoben werden können sie nur durch das Abbrechen einer der beteiligten Transaktionen, häufig der jüngsten.

Trigger

Bei der Bearbeitung von Übungsblatt 10 ist einigen bereits aufgefallen, dass Datenbanken manchmal ein gewisses Eigenleben entwickeln. Beim Update auf die SALARY-Spalte der EMPLOYEE-Tabelle zum Beispiel wird automatisch ein Satz in der Tabelle SALARY_HISTORY eingefügt. Wie funktioniert das?

Das Stichwort zu diesem Problem heißt „Trigger“. Ein Trigger ist eine Regel, die beim Auftreten einer bestimmten Operation auf der Datenbank feuert und definierte Aktionen auslöst.

Mögliche Anwendungen von Triggern:

- Einfügen von Log-Sätzen (siehe SALARY_HISTORY, Stichwort: „Revisionsfähigkeit“)
- Verhindern von Inkonsistenzen in redundanten Informationen (Aktualisierung abhängiger Attribute)
- Überwachung von Integritätsbedingungen (CHECK- und referentielle Constraints)

Der CHECK hat auf Blatt 10 ebenfalls schon einige kalt erwischt.

Trigger erlauben es damit insbesondere, einen Teil der Anwendungslogik in die Datenbank zu verschieben. Dies hat nicht nur Vorteile, sondern kann organisatorische Probleme aufwerfen. Hier werden Aufgaben des DBAs und des DB-Anwendungsentwicklers miteinander verwoben. Außerdem leidet die Portabilität einer Anwendung, da das Erstellen von Triggern nicht normiert ist.

Die Syntax einer Trigger-Definition sieht wie folgt aus:

```
CREATE TRIGGER trigger-name FOR TABLE
[ACTIVE | INACTIVE]
when operation
[POSITION number]
AS trigger-body;

when                ::= BEFORE | AFTER

operation           ::= DELETE | INSERT | UPDATE

trigger-body        ::= [variable-declaration-list] block

variable-declaration-list ::= DECLARE VARIABLE variable data-type;
{, DECLARE VARIABLE variable data-type;}

block ::=
BEGIN
    compound-statement
    {, compound-statement}
END

compound-statement ::= block | statement
```

Die möglichen Anweisungen innerhalb des Trigger-Blocks sind nicht im SQL-Standard definiert und hängen vom verwendeten RDBMS ab.

Beispiele:

- Automatisches Anlegen eines Satzes in der SALARY_HISTORY bei Änderung des Gehalts

```
CREATE TRIGGER SAVE_SALARY_HISTORY FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
    IF (OLD.SALARY <> NEW.SALARY) THEN
        INSERT INTO SALARY_HISTORY
            (EMP_NO, CHANGE_DATE, UPDATER_ID, OLD_SALARY,
             PERCENTAGE_CHANGE)
        VALUES (OLD.EMP_NO, 'now', USER, OLD.SALARY,
                (NEW.SALARY-OLD.SALARY)*100/OLD.SALARY);
    END;
```

Das Beispiel zeigt, dass mit "OLD." bzw. "NEW." auf die Werte vor und nach der Änderung zugegriffen werden kann.

- Automatische Vergabe einer neuen Employee Nummer

```
CREATE TRIGGER SET_EMP_NO FOR EMPLOYEE
BEFORE INSERT AS
BEGIN
    NEW.EMP_NO = GEN_ID(EMP_NO_GEN, 1);
END;
```

Die Existenz dieses Triggers ist übrigens auch dafür verantwortlich, dass bei Verwendung des gleichen Generators in einem eigenen INSERT-Statement auf die EMPLOYEE-Tabelle die EMP_NO immer um zwei erhöht wird.

Die restlichen Bestandteile der Trigger-Definition haben folgende Bedeutung:

- ACTIVE bzw. INACTIVE legt fest, ob der Trigger aktiv (default) oder inaktiv ist
- BEFORE bzw. AFTER legen fest, ob der Trigger-Block vor oder nach der eigentlichen Operation ausgeführt wird
- POSITION number legt bei mehreren gleichartigen Triggern die Ausführungsreihenfolge fest. Kleinere Nummern haben höhere Priorität, identische Nummern führen zu keiner eindeutigen Reihenfolge

Wie fast zu vermuten, gibt es die Möglichkeit, Trigger zu aktivieren bzw. zu deaktivieren. Hierzu existiert ein ALTER TRIGGER-Statement mit folgender Syntax:

```
ALTER TRIGGER trigger-name
[ACTIVE | INACTIVE]
[when operation]
[POSITION number]
[AS trigger-boddy];
```

Hier sind jetzt natürlich die meisten Bestandteile optional. Es kann jedoch fast jeder Bestandteil eines Triggers nachträglich modifiziert werden, nur der Name ist fest.

Beispiel: Abschalten der automatischen Employee-Nummer Vergabe

```
ALTER TRIGGER SET_EMP_NO INACTIVE;
```

Funktionen und Prozeduren

Funktionen

Unter einer Funktion versteht man eine Ausführung einer Operationenfolge, die nur einen einzigen Wert zurückgibt. Sie kann Eingabeparameter besitzen und wird innerhalb einer anderen SQL-Anweisung wie eine skalare Funktion verwendet, d.h. der Rückgabewert wird an der entsprechenden Stelle in die weitere Verarbeitung einbezogen. Die Syntax der Definition einer Funktion lautet:

```
CREATE FUNCTION function-name ([f-parameter])
RETURNS data-type
BEGIN
    statement
    {, statement}
END;
```

f-parameter ::= parameter-name data-type {, parameter-name data-type}

Prozeduren

In einer Prozedur wird ebenfalls eine Operationsfolge bearbeitet. Der Schwerpunkt liegt hier auf der Verarbeitung von Daten im Datenbanksystem, z.B. in Form einer Änderung bzw. Aktualisierung. Daher werden im Anweisungsteil einer Prozedur oftmals weitere SQL-Anweisungen ausgeführt. Strukturänderungen sind dort allerdings nichts erlaubt.

Eine Prozedur kann Parameter enthalten, die

- Daten an sie übergeben (in-Parameter)
- ausschließlich für die Datenrückgabe verwendet werden (out-Parameter)
- sowohl für die Datenein- als auch –ausgabe verwendet werden (in/out-Parameter)

Die Syntax zur Definition einer Prozedur lautet:

```
CREATE PROCEDURE procedure-name ([p-parameter])
BEGIN
    statement
    {, statement}
END;
```

p-parameter ::= parameter {, parameter}

parameter ::= parameter-type parameter-name data-type

parameter-type ::= IN | OUT | INOUT

Löschen von Funktionen und Prozeduren

Eine Funktion oder Prozedur kann auch wieder aus dem Datenbanksystem gelöscht werden. Dazu verwendet man die Anweisung DROP FUCTION bzw. DROP PROCEDURE. Die Syntax lautet:

```
DROP
FUNCTION | PROCEDURE
name;
```

4. Einbettung von SQL in Programmiersprachen

SQL ist zwar eine ziemlich mächtige Sprache, die Erstellung von kompletten Datenbank Anwendungen ist mit ihr allerdings nicht möglich. SQL wird deshalb in andere Programmiersprachen eingebettet. Hierzu existieren im Wesentlichen zwei verschiedene Ansätze:

- **Call-Level-Interfaces (CLI)**
- **Embedded SQL (ESQL)**

Call-Level-Interfaces behandeln die SQL-Anweisung im Programm als Zeichenkette und übergeben sie zur Ausführung an parametrisierbare Methoden. Rückgabewerte und Ergebnismengen werden durch Methodenaufrufe ermittelt und nach einer geeigneten Typkonvertierung im Programm weiterverarbeitet. Beispiele für diese Technik sind ODBC (Open Database Connectivity) und JDBC (Java Database Connectivity).

Bei Embedded SQL werden die SQL-Anweisungen mit besonderen Schlüsselwörtern direkt in den Quelltext eingebettet. Die Kommunikation mit dem Programm erfolgt durch speziell deklarierte Host-Variablen. Damit der Compiler durch die eingebetteten SQL-Anweisungen nicht aus dem Tritt gebracht wird, müssen sie zunächst von einem Präprozessor in geeigneten Code übersetzt werden. Ein Java-Vertreter von ESQL ist SQLJ (DB2, Oracle), welches allerdings intern auf einen JDBC-Treiber zurückgreift.

Welcher der beiden Lösungen vorzuziehen ist, richtet sich nicht zuletzt nach den Projektvorgaben potentieller Kunden. Embedded SQL Lösungen sind allerdings im Java-Umfeld kaum anzutreffen, sondern eher in klassischen Großrechner-Bereichen und unter C.

Call-Level-Interfaces

In diesem Abschnitt wollen wir jeweils einen Vertreter dieser Gattung aus dem Java- und C-Umfeld näher betrachten.

JDBC – Java Database Connectivity

Um über das JDBC-API auf eine Datenbank zugreifen zu können, wird ein JDBC-Treiber für die entsprechende Datenbank benötigt. Dieser befindet sich bei kommerziellen RDBMS in der Regel im Lieferumfang. Für freie Datenbanken – wie etwa Firebird, MySQL und PostgreSQL – sind JDBC-Treiber in der Regel ebenfalls kostenfrei erhältlich.

Was leistet ein JDBC-Treiber?

- Herstellung einer Verbindung (*Connection*) zu einer Datenbank
- Absetzen von *Statements*, d.h. Abfragen und ändernden Anweisungen (*INSERT*, *UPDATE*, *DELETE*) an die Datenbank
- Verarbeiten der Ergebnisse (*ResultSets*)

Die im JDBC-API zur Verfügung stehenden Interfaces und Klassen finden sich in den Paketen `java.sql` und `javax.sql`. Dort finden sich im wesentlichen Interfaces, die vom Hersteller eines JDBC-Treibers mit Leben gefüllt werden müssen.

Typen von JDBC-Treibern

JDBC-Treiber werden je nach Arbeitsweise und Implementierung in 4 Klassen unterteilt:

- Typ-1: JDBC-ODBC Bridge (für Windows und Solaris) mit ODBC-Treiber
- Typ-2: Teilweise Java-JDBC-Treiber, die über das **Java Native Interface (JNI)** auf System-spezifischen Code zurückgreifen, der das spezifische Client-API der Datenbank verwendet.
- Typ-3: Pure Java-JDBC-Treiber, die eine Datenbank-unabhängiges Netzwerk-Protokoll verwenden und eine Middleware zur Umsetzung auf das Datenbank-spezifische Netzwerk-Protokoll benötigen
- Typ-4: Pure Java-JDBC-Treiber, die das Datenbank-spezifische Netzwerk-Protokoll verwenden

Diese Klassifikation legt nahe, dass nach Möglichkeit Typ-4-Treibern der Vorzug gegeben werden sollte. Sie verursachen den geringsten Installationsaufwand und bieten in der Regel die beste Performance.

Der Firebird JDBC-Treiber Jaybird ist ein Typ-4-Treiber. Er ist komplett in Java realisiert (nur `jar`-Dateien erforderlich) und kommuniziert mit dem Firebird-Server über dessen spezifisches Protokoll.

Prinzipielles Vorgehen bei JDBC-Zugriffen

Das Ausführen von SQL-Statements über das JDBC-API erfolgt in folgenden Schritten:

- Laden des JDBC-Treibers
- Öffnen einer Verbindung
- Erzeugen von Statement-Objekten
- Ausführen der Anweisung
- Auswertung der Ergebnismenge bei Auswahlabfragen

Laden des JDBC-Treibers

Zum Laden des JDBC-Treibers ist folgender Aufruf erforderlich:

```
Class.forName("org.firebirdsql.jdbc.FBDriver");
```

Hier wird Jaybird geladen, sein `jar`-File muss dazu natürlich im Classpath liegen.

Alle JDBC-Treiber verfügen über einen statischen Initialisierer, der den Treiber beim *DriverManager* registriert. Dies muss nicht explizit im Programm erfolgen, findet sich aber überflüssigerweise noch in vielen Beispielen. Es schadet zwar nicht, erzeugt aber ein nutzloses Duplikat.

Öffnen einer Verbindun

Nach dem Laden des Treibers kann über den *DriverManager* eine Connection angefordert werden. Folgender Aufruf ist dazu erforderlich:

```
Connection con = DriverManager.getConnection(url, user, password);
```

Für unsere Firebird-Datenbank `employee.gdb` und Jaybird sieht der Aufruf konkret wie folgt aus:

```
Connection con = DriverManager.getConnection(
    "jdbc:firebirdsql://localhost:3050/C:/programme/firebird/examples/employee.gdb",
    "SYSDBA",
    "masterkey");
```

Eine JDBC-URL setzt sich dabei aus drei durch einen Doppelpunkt getrennten Teilen zusammen:

- dem Protokoll (für JDBC immer „jdbc“)
- dem Subprotokoll, Treibername („firebirdsql“) oder Verbindungsart („odbc“)
- einer Information, wo die Datenbank zu finden ist

Der letzte Punkt kann unter anderem vom Subprotokoll und der Willkür des Treiber-Herstellers abhängen. Bei Jaybird lautet die Syntax:

```
//server:port/fully-qualified-database-path
```

Für unsere DB mit lokal laufendem Server auf dem Standard-Port 3050 also:

```
//localhost:3050/c:/programme/firebird/examples/employee.gdb
```

Erzeugen von Statement-Objekten

JDBC kennt drei verschiedene Typen von Statement-Klassen:

- Statement
- PreparedStatement
- CallableStatement

Der einfachste Typ `Statement` ist zunächst nicht fest mit einem SQL-Statement verknüpft. Dies wird in seiner vollständigen Form erst beim Ausführen der Abfrage als String übergeben. Dies muss allerdings vor jeder Abfrage erneut geschehen, ein einfaches `Statement`-Objekt hat kein Gedächtnis.

Prepared Statements werden beim Erzeugen mit einem SQL-Statement verknüpft. Dieses muss allerdings nicht vollständig sein und darf dort, wo Konstanten erlaubt sind, noch Platzhalter („?“) enthalten. Diese müssen vor der Ausführung der Abfrage mit Hilfe spezieller Setter mit gültigen Werten belegt werden. Nach der Ausführung der Abfrage kann ein Prepared Statement wiederverwendet und mit neuen Werten bestückt werden.

Die meisten RDBMS cachen Prepared Statements und können diese bei der zweiten Verwendung bereits deutlich schneller ausführen. Kommt ein bestimmtes SQL-Statement also häufiger und sogar mit wechselnden Parametern vor, empfiehlt sich ihre Verwendung.

Callable Statements schließlich bieten einen Zugriff auf Stored Procedures, fest in der Datenbank abgelegte Abfragen. Da dies aber stark von der verwendeten DB abhängt, sollte man sie meiden. Java ist schließlich Plattform-unabhängig!

Ausführung der Anweisung

Die Ausführung einer Abfrage mit Hilfe eines einfachen `Statement`-Objekts sieht wie folgt aus:

```
ResultSet rs = stmt.executeQuery("SELECT ...");
```

Ein `ResultSet` kann man sich als temporäre Tabelle vorstellen, die mit Hilfe der Methode `next` zeilenweise durchlaufen werden kann. Ein `ResultSet` verliert seine Gültigkeit, sobald das `Statement` geschlossen wird oder eine neue Query abgesetzt wird. Auch das Schließen der zugehörigen Verbindung invalidiert ein `ResultSet`.

Ein typisches Code-Fragment zum Auslesen eines ResultSets hat folgendes Aussehen:

```
while(rs.next())
{
    for(int i=1; i<=rs.getMetaData().getColumnCount(); i++)
    {
        String s = rs.getObject(i).toString();

        // Do something meaningful
    }
}

rs.close();
```

Wichtig: Der Zeilenzeiger (Cursor) steht nach der Ausführung vor der ersten Zeile und muss mit `next` zunächst auf die erste Zeile der Treffermenge vorgerückt werden. Außerdem ist JDBC im Gegensatz zu Java nicht 0-sondern 1-basiert.

Um alle Datenbank-Ressourcen wieder freizugeben, müssen anschließend noch das `Statement` und die `Connection` mit `close` geschlossen werden.

Verwendung von Prepared Statements

Die Ausführung einer Abfrage mit Hilfe eines Prepared Statement Objekts sieht wie folgt aus:

```
PreparedStatement pstmt = con.prepareStatement(
    „SELECT AVG(SALARY) FROM EMPLOYEE E JOIN DEPARTMENT D ON E.DEPT_NO=D.DEPT_NO“);

String locations[] = {„Boston“, „Burlington, VT“, „Cannes“, „Kuaui“, „London“,
    „Milan“, „Monterey“, „San Francisco“, „Singapore“,
    „Tokyo“, „Toronto“, „Zurich“};

for(int i=0; i<locations.length(); i++)
{
    pstmt.setString(1, locations[i]);
    ResultSet rs = pstmt.executeQuery();

    if(!rs.next())
    {
        throw new SQLException(„SELECT AVG(SALARY): no result“);
    }

    System.out.println(„average salary for “ + locations[i] + “: “ +
        rs.getDouble(1));
}

pstmt.close();
```

Ein SQL-Statement wird bis auf die variablen Anteile als Prepared Statement erzeugt und dann vor jeder Erneuten Verwendung mit Werten gefüllt. Eventuell benötigte einfache Hochkommata fügt JDBC selbst sein.

SQL-Datentypen und zugehörige Getter

Die SQL-Datentypen und die in Java verfügbaren primitiven Objekttypen werden wie folgt durch die `ResultSet.getXXX()` –Methoden aufeinander abgebildet:

Java-Typ	getXXX()	SQL-Typ
int	<code>getInt()</code>	INTEGER
float	<code>getFloat()</code>	FLOAT
double	<code>getDouble()</code>	DOUBLE
BigDecimal	<code>getBigDecimal()</code>	DECIMAL NUMERIC
String	<code>getString()</code>	VARCHAR CHAR
<code>java.sql.Date</code>	<code>getDate()</code>	DATE
<code>java.sql.Time</code>	<code>getTime()</code>	TIME
<code>java.sql.Timestamp</code>	<code>getTimestamp()</code>	TIMESTAMP
Object	<code>getObject()</code>	*

Transaktionsverarbeitung mit JDBC

Standardmäßig werden Connection-Objekte erzeugt, die im sogenannten Auto-Commit Modus arbeiten. Dies bedeutet, dass nach dem vollständigen Auslesen eines ResultSets oder vor dem Absetzen eines weiteren Statements über dieselbe Connection ein Commit ausgeführt wird.

Für das Abarbeiten mehrerer Statements innerhalb einer Transaktion ist diese Modus naturgemäß ungeeignet. Er kann deshalb mit der Methode

```
public void setAutoCommit(boolean autoCommit) throws SQLException;
```

geändert werden.

Wird nicht länger automatisch ein Commit oder Rollback ausgeführt, muss dies vom Programm selbst vorgenommen werden. Hierzu dienen die Methoden

```
public void commit() throws SQLException;
```

```
public void rollback() throws SQLException;
```

Im Zusammenhang mit der Transaktionsverarbeitung haben wir zusätzlich die sogenannten Isolation-Level kennengelernt. Diese werden mit Hilfe der Connection-Methode

```
public void setTransactionIsolation(int level) throws SQLException;
```

gesetzt. Es werden die Werte

- `Connection.TRANSACTION_READ_UNCOMMITTED`
- `Connection.TRANSACTION_READ_COMMITTED`
- `Connection.TRANSACTION_REPEATABLE_READ`
- `Connection.TRANSACTION_SERIALIZABLE`

unterstützt, die in der angegebenen Reihenfolge unseren Leveln 0 bis 3 entsprechen.

Eine Transaktion mit höchsten Isolation-Level läuft unter JDBC also nach folgendem Schema ab:

```
try
{
    try
    {
        con.setAutoCommit(false);
        con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

        // Process statements
    }
    catch(SQLException ex1)
    {
        con.rollback();
        throw new SomeException("error processing transaction", ex1);
    }

    con.commit();
}
catch(SQLException ex2)
{
    // Commit must have failed
}
```

ODBC – Open Database Connectivity

Das Pendant zum JDBC-basierten Datenbank-Zugriff in der C- und C++-Welt ist ODBC (Open Database Connectivity). Die Konzepte sind vergleichbar, da in C aber keine Klassen und Objekte zur Verfügung stehen, ist das API Handle-basiert.

Für Firebird existieren von XTG Systems ein „InterBase 6 ODBC Driver for Win32“ (Lizenz LGPL), der Download findet sich auf der IBPhoenix-Seite. Der ODBC-Treiber enthält ein Installationsprogramm. Bevor allerdings über ODBC auf eine Datenbank zugegriffen werden kann, muss im System ein DSN (Data Source Name) definiert werden. Der ODBC-Treiber wird dabei mit der Datenquelle verknüpft und unter einem symbolischen Namen registriert. In ODBC-basierten Programmen muss deshalb weder ein Treiber noch ein Datenbank-URL explizit angegeben werden, es wird nur ein DSN benötigt.

Embedded SQL

SQLJ ist eine Variante von SQL, die in der Java-Welt verbreitet ist. Der Programmcode wird von einem Präprozessor umgewandelt, das Ergebnis verwendet dann meist die JDBC-API. SQLJ-Präprozessoren sind u.a. von IBM und Oracle erhältlich, können aber meist auch für andere RDBMS verwendet werden, solange nur ein JDBC-Treiber verfügbar ist.

SQLJ bettet die SQL-Anweisungen mittels eines speziellen Schlüsselwortes in Java-Quelltexte ein. Syntax:

```
#sql { SQL-Statement };
```

Der Transfer von Daten zwischen SQL und Java erfolgt über sogenannte Host-Variablen. Die hier vorgestellten Beispiele stammen im wesentlichen aus der DB2-Dokumentation zu SQLJ. Das Laden des JDBC-Treibers und Anfordern von DB-Connections erfolgt wie bei der Vorstellung des JDBC-APIs bereits gesehen. Nur die eigentlichen Statements werden anders behandelt.

Beispiel:

```
try
{
    String jobUpdate = null;

    jobUpdate="Clerk";
    #sql {UPDATE staff SET job = :jobUpdate WHERE HOB = 'Mgr'};
    System.out.println("\nAll 'Mgr' have been demoted to 'Clerk'!");

    jobUpdate="Sales";
    #sql {DELETE FROM staff WHERE job = :jobUpdate};
    System.out.println("All 'Sales' people have been deleted");

    #sql {INSERT INTO staff VALUES(999, 'Testing', 99, :jobUpdate, 0, 0, 0)};
    System.out.println("New data has been inserted");
}
catch(Exception e)
{
    throw e;
}
finally
{
    // Rollback the transaction
    System.out.println("\nRollback the transaction...");
    #sql{ROLLBACK};
    System.out.println("Rollback done");
}
```

Beispiel: ResultSet-Verarbeitung mit SQLJ

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator CursorByName(String name, short dept);
#sql iterator CursorByPos(String, short);

// ...

try {
    CursorByName    cursorByName;
    CursorByPos     cursorByPos;

    String  name = null;
    short   dept = 0;

    // Using the JDBC ResultSet cursor method with a 'bind by name' cursor
    #sql cursorByName = {SELECT NAME, dept FROM staff WHERE job='Mgr'};

    while(cursorByName.next()) {
        name = cursorByName.name();
        dept = cursorByName.dept();

        System.out.print(" name = " + name);
        System.out.print(" dept = " + dept);
        System.out.print("\n");
    }

    cursorByName.close();

    // Using the SQLJ iterator cursor method with a "bind by position" cursor
    #sql cursorByPos = {SELECT name, dept FROM staff WHERE job='Mgr'};
    while(true) {
        #sql {FETCH :cursorByPos INTO :name, :dept}
        if(cursorByPos.endFetch())
            break;

        System.out.print(" name = " + name);
        System.out.print(" dept = " + dept);
        System.out.print("\n");
    }
    cursorByPos.close();
}
catch(Exception e) {
    throw e;
}
finally {
    // Rollback the transaction
    #sql {ROLLBACK};
    System.out.println("Rollback done");
}
```

JDBC Performance Tuning

Der erste Schritt bei der Optimierung von Datenbankzugriffen besteht aus dem Profiling. Ein denkbarer Ansatz ist, sich auf der Treiberebene in jeden Aufruf einzuklinken und die Ausführungszeiten zu protokollieren. Der frei verfügbare JDBC-Treiber „com.p6spy.engine.spy.P6SpyDriver“ verfolgt genau diesen Ansatz. Er wird statt des normalen, Datenbank-spezifischen JDBC-Treibers verwendet. In seiner Konfigurations-Datei „spy.properties“ wird allerdings der richtige Treiber angegeben:

```
realdriver=com.mysql.jdbc.Driver
```

Mit Hilfe weiterer Einträge

```
log4j.appender.SQLPROFILER_CLIENT=org.apache.log4j.net.SocketAppender
log4j.appender.SQLPROFILER_CLIENT.RemoteHost=localhost
log4j.appender.SQLPROFILER_CLIENT.Port=4445
log4j.appender.SQLPROFILER_CLIENT.LocationInfo=true

log4j.logger.p6spy=DEBUG, SQLPROFILER_CLIENT
```

kann p6spy so konfiguriert werden, dass er sein Log nicht nur auf die Konsole schreibt, sondern Verbindungen zu Profiling-Clients aufbaut. Die Beispielkonfiguration trifft etwa für den SQLProfiler (Link bei <http://www.p6spy.com>) zu. Dort finden sich auch Links zu weiteren SQL-Profiling Tools und zu Artikeln zum Thema.

Sind mit Hilfe des Profilers die Engpässe lokalisiert, kann über Tuningmaßnahmen nachgedacht werden. Hier bieten sich zunächst die Verwendung der bereits vorgestellten Prepared Statements an. Diese machen aber nur dann Sinn, falls sie tatsächlich mit Hilfe desselben Connection-Objekts wiederholt ausgeführt werden. Dies kann in der Regel nicht sicher gestellt werden.

In diesem Fall bietet sich die Verwendung von Prepared Statement Caches an. Hier wird im Prinzip eine kleine Map mit offenen Prepared Statements verwaltet, die wieder verwendet werden können. Als Schlüssel können die Statement-Strings selbst dienen.

Es gibt JDBC-Treiber, die das von sich aus unterstützen (Jaybird). Für den Rest der Welt gibt es zum Beispiel Commons DBCP (<http://jakarta.apache.org/commons/dbcp/>) aus dem Jakarta-Projekt. DBCP benötigt zusätzlich noch Commons Pool und die Commons Collections.

Commons DBCP bietet zusätzlich noch Connection Pooling, eine Sache, die prinzipiell angeraten ist. Das Aufbauen von Verbindungen zur Datenbank ist eine teure Operation. Verbindungen sollten deshalb wiederverwendet werden. Natürlich muss mit Timeout-Problemen etc. geeignet umgegangen werden.

Dazu wird in der Regel eine Dummy-Abfrage verwendet, die möglichst kurze Ausführungszeit hat. Für MySQL funktioniert „select null“. Der Pool geht hierbei in regelmäßigen Abständen alle Connections durch und überprüft sie durch Absetzen der Dummy-Abfrage auf Gültigkeit. Die Dummy-Abfrage vor jeder echten Abfrage abzusetzen wäre kontraproduktiv!

Beispiel: Pooling + Statement Caching mit DBCP

```
Properties properties = new Properties();
properties.put("username", „root");
properties.put("password", "");
properties.put("url", "jdbc:mysql://localhost/cdcol");
// properties.put("driverClassName", "com.p6spy.engine.spy.P6SpyDriver");
properties.put("driverClassName", "com.mysql.jdbc.Driver");
properties.put("poolPreparedStatements", "true");
properties.put("maxOpenPreparedStatements", "5");
// properties.put("validationQuery", "select null");

DataSource ds = null;
try
{
    ds = BasicDataSourceFactory.createDataSource(properties);
}
catch(Exception e)
{
    e.printStackTrace();
    return;
}

Connection con = ds.getConnection();
PreparedStatement pst = con.prepareStatement("select * from cds");
ResultSet rs = pst.executeQuery();

...
```