

### 5.4.5 MySQL-Besonderheiten in der SQL-DML-Syntax

Die INSERT-Anweisung hat noch drei zusätzliche Optionen:

- **DELAYED:** In MyISAM- und ISAM-Tabellen bewirkt diese Option, dass die Anweisung in eine Warteschlange gesetzt und dadurch verzögert ausgeführt wird. In anderen Tabellen ist die Option nicht zulässig.
- **LOW\_PRIORITY:** Die INSERT-Anweisung wird erst ausgeführt, sobald kein anderer Benutzer mehr auf die Datenbank zugreift.
- **IGNORED:** Fehlermeldung beim Einfügen von doppelten Datensätzen, die mit PRIMARY KEY oder UNIQUE versehen sind, werden unterdrückt und die Datensätze werden trotzdem eingefügt.

Zu beachten ist, dass Spalten mit der Option AUTO\_INCREMENT bei INSERT-Anweisungen nicht berücksichtigt werden müssen, da diese Spalten systemintern automatisch (vgl. Abschnitt 5.3.10) gefüllt werden. Außerdem kann eine INSERT-Anweisung auch über eine SET-Option erfolgen:

```
<INSERT mit SET-Option Anweisung> ::=  
  INSERT INTO Tabellename SET <Wertzuweisung> [ , <Wertzuweisung>]... ;
```

Es gibt eine REPLACE-Anweisung, die ähnlich wie die INSERT-Anweisung arbeitet, mit dem Unterschied, dass vorhandene Daten durch neue ersetzt werden – also eine entsprechende Funktionalität zum MERGE von Oracle und SQL. Des Weiteren gibt es beim UPDATE eine Version, die Datenänderungen in mehreren Tabellen ermöglicht.

## 5.5 Die Datenabfragesprache (DQL, Data Query Language)

Inhalt dieses Sprachbestandteils von SQL ist nur die SELECT-Anweisung zum Lesen von Daten in ihrer komplexen Ausprägung. Zudem finden die hier definierten <Anfragesätze> und <Suchbedingungen>, wie wir bereits gesehen haben, Verwendung in vielen Anweisungen der SQL-DDL und -DML.

### 5.5.1 Die SELECT-Anweisung in der Grundform

Die SELECT-Anweisung ist wohl die umfangreichste SQL-Anweisung, obwohl sie keine Daten verändert, sondern nur liest. Sie besteht in der Grundform aus sechs Klauseln.

```
<SELECT Grundform> ::=  
  <SELECT Klausel>  
  <FROM Klausel>  
  [<WHERE Klausel>]  
  [<GROUP BY Klausel>]  
  [<HAVING Klausel>]  
  [<ORDER BY Klausel>];
```

Tabelle 5.23

### Ein Überblick über die sechs Klauseln in der Grundform

<b>SELECT</b>	Projiziert auf die gewünschten Spalten	obligatorisch
<b>FROM</b>	Definiert die benutzten Tabellen	obligatorisch
<b>WHERE</b>	Selektiert aufgrund der Suchbedingung bestimmte Datensätze der Tabellen	optional
<b>GROUP BY</b>	Gruppert Datensätze auf Basis gleicher Spaltenwerte	optional
<b>HAVING</b>	Selektiert nur Gruppen, die der HAVING-Bedingung genügen. Die Bedingung darf Gruppenfunktionen enthalten.	optional
<b>ORDER BY</b>	Bestimmt die Reihenfolge der Datensätze	optional

Minimal muss eine Anfrage also aus den beiden Klauseln SELECT und FROM bestehen. Bevor wir in den nächsten Kapiteln in die Einzelheiten einsteigen, sollen einige einfache Beispiele der SELECT-Anweisung vorgestellt werden. Der Spaltenausdruck ist im einfachsten Fall eine Liste von Spaltennamen, die durch Kommas getrennt sind, oder ein „\*“, der für alle Spalten der Tabelle steht.

### Beispiele

- Man gebe eine Liste aller Kunden mit Kun\_Nr, Vorname und Nachname aus Gummersbach aus!

```
SELECT Kun_Nr, Vorname, Nachname  
FROM Kunden  
WHERE Ort = 'Gummersbach';
```

### Ergebnis:

Kun_Nr	Vorname	Nachname
5	Hannelore	Sündbald
6	Birgit	Wal

- Welche Abteilungen gibt es in der Byce & Co.-Datenbank?

```
SELECT Abt_Nr, Name FROM Abteilungen;
```

### Ergebnis:

Abt_Nr	Name
1	Geschäftsleitung
2	Produktion
3	Verkauf
4	Einkauf
5	Arbeitsvorbereitung
6	Datenverarbeitung

- Geben Sie Nachnamen und Vornamen aller Angestellten aus!

```
SELECT Vorname || ' ' || Nachname AS Angestellte
  FROM Angestellte;
```

Ergebnis:

Angestellte	Angestellte	Angestellte
Josefine Müller	Jonas Falser	Lucas Glatt
Hans Fama	Erna Wanne	Barbara Gotte
Iris Heck	Ilse Brunn	Holger Kall
Otto Schmidt	Willi Brater	Franz Käse
Anna Weber	Susanne Bär	Anna Kussmann
Paul Frisch	Hermann Budar	Leo Bold
Paula Frisch	Max Bille	Max Butz
Ernst Schneider	Thorsten Wahn	Isabel Barthels
Anna Weber	Petra Wuton	Hugo Schmidt

27 Datensätze werden ausgewählt (diese Zeilen werden eigentlich untereinander ausgegeben und sind hier nur aus Platzmangel auf drei Spalten verteilt). Vor- und Nachname werden durch den Operator „Konkatenation“ ( „||“) verbunden und in der Ausgabe durch ein Leerzeichen getrennt.

- Welchen Inhalt hat die Relation „Werke“?

```
SELECT * FROM Werke;
```

Ergebnis:

WNR	BEzeichnung	ORT	STRASSE
1	Montagewerk	Gladbeck	Adlerstr.
2	Vorfertigung	Gladbeck	Am Pferdekamp

## 5.5.2 Spalten projizieren mit der SELECT-Klausel

Die SELECT-Klausel wählt die Attribute der zu erstellenden Anfrage aus einer oder mehreren Tabellen aus. Dabei können nicht nur Spaltennamen verwendet werden, sondern auch Ausdrücke auf Spaltennamen, wie wir sie im Abschnitt 5.3.1. zusammengestellt haben.

```
<SELECT Klausel> ::= 
  SELECT [ ALL | DISTINCT ] <Spaltenausdruck> [, <Spaltenausdruck> ]...
<Spaltenausdruck> ::= / [ Tabellenname.]*
  | [ Tabellenname.]Spaltenname [ AS Neuer_Spaltenname ]
  | <Ausdruck> [ AS Neuer_Spaltenname ] /
```

Anmerkungen:

- ALL gibt auch doppelte Datensätze aus und DISTINCT unterdrückt doppelte.
- \* gibt alle Spalten der beteiligten Tabellen aus.
- „Ausdruck“ entspricht dem Ausdruck in der CREATE TABLE-Anweisung, also Operationen und Funktionen auf Spaltennamen und Konstanten.
- Der Tabellenname kann auch eine in der FROM-Klausel vergebene Abkürzung sein.
- Die FROM-Klausel wird im nächsten Kapitel behandelt.

### Beispiele

```
SELECT * FROM Kunden;
SELECT DISTINCT Kunden.Nachname FROM Kunden;
SELECT 'Netto', Gehalt-Abzuege, Gehalt * 0.1,
      Vorname||' '|Nachname AS Name
  FROM Angestellte;
SELECT Ang_Nr, Nachname, CURRENT_DATE, Eintrittsdatum
  FROM Angestellte;
```

## 5.5.3 Daten aus (mehreren) Tabellen auswählen mit der FROM-Klausel

In der FROM-Klausel wird angegeben, aus welchen Tabellen die zu selektierenden Daten stammen. Es sind eine oder mehrere Tabellen sowie Unterabfragen (nur SQL2003 und Oracle) und sogenannte JOIN-Tabellen zulässig.

```
<FROM Klausel> ::= FROM <Tabellenreferenz>
<Tabellenreferenz> ::= 
  [ Tabellenname [Abkürzung] | (<Unterabfrage>) Abkürzung | <JOIN Tabelle> ]
  [, / Tabellenname [Abkürzung] | (<Unterabfrage>) Abkürzung | <JOIN Tabelle> ] ...
```

Anmerkungen:

- Im einfachsten Fall besteht die FROM-Klausel aus dem Schlüsselwort FROM und einem Tabellennamen.
- Es können aber auch mehrere Tabellen, durch Kommas getrennt, einbezogen werden.
- Eine Tabelle kann auch mehrfach aufgelistet werden (Selbstreferenz, Rekursion). In diesem Fall müssen unterschiedliche Abkürzungen als sogenannte Tabellenalise vergeben werden.
- Falls mehrere Tabellen vorkommen, die gleiche Spaltennamen haben, z.B. die Kun\_Nr in den Relationen „Kunden“ und „Auftraege“, ist es notwendig, Abkürzungen oder den ausgeschriebenen Tabellennamen zu verwenden. Diese Abkürzungen gehen als Referenzierungen mit in den Spaltenausdruck der anderen Klausel ein.
- Die FROM-Klausel ist ein Synonym für die Funktionalität des kartesischen Produkts aus der relationalen Algebra. Die Datensätze der verschiedenen Tabellen oder

Unterabfragen werden wie beim kartesischen Produkt, also jeder mit jedem verknüpft (vgl. Abschnitt 4.1.3). Durch passende WHERE-Klauseln können daraus JOIN-Verknüpfungen werden.

- Wir beschränken uns erst einmal in den Beispielen auf die Verwendung von Tabellennamen. Die Erläuterungen zu den Unterabfragen finden Sie im Abschnitt 5.5.9 und die für die <JOIN Tabelle> im Abschnitt 5.5.4.

### Beispiele

```
SELECT * FROM Teile;
SELECT * FROM Teile, Artikel;
SELECT Kunden.Kun_Nr, Kunden.Vorname, Kunden.Nachname, Auftraege.AuftragsNr
FROM Kunden, Auftraege
WHERE Auftraege.Kun_Nr = Kunden.Kun_Nr;
SELECT K.Kun_Nr, K.Vorname, K.Nachname, A.AuftragsNr
FROM Kunden K, Auftraege A
WHERE A.Kun_Nr = K.Kun_Nr;
SELECT 1000 - (t.MindestBestand-t.Bestand) * l.Bestand
FROM Teile t, Lagerbestand l
WHERE t.TNr = l.TNr;
```

Beim zweiten Beispiel sind die beiden Tabellen Teile und Artikel mit einem kartesischen Produkt verknüpft, im dritten bzw. vierten Beispiel als Join, um genauer zu sein, als Equi Join bzw. sogar als Natural Join.

Die dritte und die vierte Anweisung haben das gleiche Ergebnis, nur dass in der vierten Anweisung Tabellenabkürzungen benutzt werden.

Man kann Tabellen auch mit sich selbst verknüpfen:

```
SELECT s1.OTeil, s2.OTeil FROM Struktur s1, Struktur s2;
SELECT s1.OTeil
FROM Struktur s1, Struktur s2
WHERE s1.OTeil = s2.UTeil;
```

Beim ersten SELECT wird jeder Datensatz der Tabelle „Struktur“ mit jedem Satz derselben Tabelle verbunden. Das Ergebnis ist ein kartesisches Produkt mit  $51^2=2601$  Datensätzen, da die Strukturtabelle der Byce & Co.-Datenbank 51 Sätze hat. Im zweiten SELECT erhält man alle Datensätze der Strukturtabelle, die als UTeil und OTeil irgendwo aufgeführt werden, also um alle Baugruppen. Es handelt sich um 24 Datensätze.

### 5.5.4 Mehrere Tabellen mit JOIN-Operatoren abfragen

Die Syntax der Basisform der FROM-Klausel haben wir gerade erläutert. Damit können Sie wie bei den Beispielen gezeigt auch einen NATURAL-JOIN über mehrere Tabellen darstellen, wenn Sie die Spalten in der WHERE-Bedingung mit dem Gleichheitszeichen verbinden. Die Syntax der FROM-Klausel lässt sich noch erweitern um alle JOIN-Operatoren der Relationalen Algebra. Bislang haben wir die FROM-Klausel nur mit der Funktionalität kennengelernt, dass alle aufgelisteten Tabellen bzw. Unterabfragen durch ein kartesisches Produkt miteinander verknüpft werden und die semantisch richtigen Verknüpfungen erst in der WHERE-Klausel spezifiziert werden. Mit den JOIN-Operatoren wird es nun möglich, gleich in der FROM-Klausel die JOIN-

Verknüpfung samt Bedingung zu formulieren. Die Verwendung von JOIN-Operatoren, von denen wir die meisten im Abschnitt 4.1.3 vorgestellt haben, zeigt zudem noch einmal den unmittelbaren Zusammenhang zwischen SQL und der relationalen Algebra auf (vgl. Abschnitt 5.7).

```
<JOIN Tabelle> ::= <Kreuzprodukt> | <THETA JOIN> | <NATURAL JOIN>
<Kreuzprodukt> ::= <Tabellenreferenz> CROSS JOIN <Tabellenfaktor>
<THETA JOIN> ::= <Tabellenreferenz> [ <JOIN Typ> ] JOIN <Tabellenreferenz>
                  | ON <Suchbedingung>
                  | USING ( Spaltenname [ , Spaltenname ]... ) ]
<NATURAL JOIN> ::= <Tabellenreferenz>
                  NATURAL [ <JOIN Typ> ] JOIN <Tabellenfaktor>
<JOIN Typ> ::= INNER | / LEFT | RIGHT | FULL | / OUTER ]
```

Eine <Tabellenreferenz> (vgl. Abschnitt 5.5.3) kann ein Tabellenname sein, ein Anfrageausdruck (vgl. Abschnitt 5.5.11) oder auch wieder eine <JOIN Tabelle>. Ein <Tabellenfaktor> kann nur ein Tabellenname oder ein Anfrageausdruck sein.

Es lassen sich drei JOIN-Operatoren differenzieren, das <Kreuzprodukt>, der <NATURAL JOIN> und der <THETA JOIN>, der auch INNER JOIN heißt. Das <Kreuzprodukt> bildet das kartesische Produkt der beiden Datensatzmengen, die als Operatoren angegeben sind, und stellt damit nur eine andere Schreibweise von der uns bereits bekannten Form der FROM-Klausel mit einer Komma getrennten Liste von Tabellennamen oder Anfrageausdrücken ohne eine WHERE-Klausel dar. Es ist der einzige JOIN-Operator, bei dem doppelte Spalten nicht automatisch eliminiert werden.

Für die beiden anderen JOIN-Operatoren (THETA JOIN und NATURAL JOIN) können vier JOIN-Typen spezifiziert werden:

Tabelle 5.24

#### Die vier JOIN-Typen des THETA- und NATURAL JOIN

	Bedeutung
INNER	Die beiden Datensatzmengen werden ohne „dangling tuples“ verknüpft. In der Ergebnismenge sind nur die Datensätze enthalten, die mit einem Satz aus der jeweils anderen Datensatzmenge verknüpft werden konnten (DEFAULT).
LEFT OUTER	Vom linken Operanden werden alle Sätze in die Ergebnismenge aufgenommen. Zu jedem Satz wird, falls es passt, ein Satz aus der rechten Menge verknüpft. Falls es nicht passt, bleiben für diesen Satz die Attribute der rechten Menge leer (vgl. linker OUTER JOIN im Abschnitt 4.1.3).
RIGHT OUTER	Das analoge Verhalten zum linken OUTER JOIN, nur dass jetzt alle Datensätze des rechten Operanden genommen werden (vgl. rechter OUTER JOIN im Abschnitt 4.1.3).
FULL OUTER	Hier werden sowohl alle Datensätze des rechten als auch des linken Operanden genommen und die jeweiligen „dangling tuples“ mit NULL-Werten aufgefüllt.

Die Funktionalität des <THETA JOIN>-Operators entspricht dem THETA-JOIN aus der Relationalen Algebra, mit dem Unterschied, dass gleichlautende Spalten automatisch eliminiert werden. Entweder wird mit *ON <search condition>* eine beliebige verknüpfende Bedingung angegeben oder es werden mit der *USING*-Option die Spaltennamen aufgelistet, für die Gleichheitsverknüpfungen formuliert werden. Diese Spaltennamen müssen in beiden Operandenmengen gleich heißen. Werden mehrere Spaltennamen angegeben, dann werden die einzelnen Vergleiche mit AND verknüpft.

Die Funktionalität des <NATURAL JOIN>-Operators entspricht dem NATURAL-JOIN aus der Relationalen Algebra. Eine verknüpfende Bedingung muss nicht angegeben werden, da ja automatisch für alle Spalten, die gleich heißen, ein Gleichheitsvergleich generiert wird. Mehrere Vergleiche werden dann mit AND verbunden. Die doppelten Spalten werden eliminiert.

Bis auf die Funktionalität der OUTER JOIN-Operatoren sind diese Schreibweisen semantisch redundant zur bisherigen Syntax aus Abschnitt 5.5.3 mit einer entsprechenden WHERE-Klausel. JOINED TABLE-Ausdrücke lassen sich also nur als linker Operand schachteln, nicht als rechter<sup>44</sup>. JOIN-Verknüpfungen können nicht über Large Object-Spalten ausgeführt werden.

### Beispiele



- Das kartesische Produkt zwischen den Tabellen „Teile“ und „Struktur“ liefert 3111 Datensätze:

```
SELECT COUNT(*) FROM Teile CROSS JOIN Struktur;
SELECT COUNT(*) FROM Teile, Struktur;
```

- Der THETA-JOIN mit der Bedingung, dass das Teil ein Oberteil in der Struktur-tabelle sein muss, liefert 51 Sätze:

```
SELECT * FROM Teile INNER JOIN Struktur ON Teile.TNr = Struktur.OTeil;
SELECT * FROM Teile, Struktur WHERE Teile.TNr = Struktur.OTeil;
```

- Der rechte bzw. linke OUTER JOIN liefert 51 bzw. 103 Datensätze:

```
SELECT * FROM Teile RIGHT OUTER JOIN Struktur ON Teile.TNr = Struktur.OTeil;
SELECT * FROM Teile LEFT OUTER JOIN Struktur ON Teile.TNr = Struktur.OTeil;
```

- Ein NATURAL JOIN kann zwischen den Angestellten und der Abteilung formuliert werden.

```
SELECT * FROM Abteilungen NATURAL JOIN Angestellte;
SELECT * FROM Abteilungen, Angestellte
WHERE Abteilungen.Abt_Nr = Angestellte.Abt_Nr;
SELECT * FROM Abteilungen, Angestellte
WHERE Abteilungen.Abt_Nr = Angestellte.Abt_Nr
AND Abteilungen.Ort = Angestellte.Ort;
```

<sup>44</sup> Um die Details und Grenzen dieser Verschachtelung herauszufinden, verweisen wir auf die Originalliteratur [ANSI SQL 2003b], [ORACLE SQL 2005, Kapitel 19] und [MySQL 2006, Kap. 13.2.7.1].

Der NATURAL JOIN der ersten Anfrage liefert ein überraschendes Ergebnis: 5 Datensätze statt der 27 Mitarbeiter, die in verschiedenen Abteilungen arbeiten (Ergebnis der zweiten Anfrage). Zu beachten ist, dass neben der Abt\_Nr auch noch der Ort in beiden Tabellen auftaucht. Die Semantik des ersten und dritten NATURAL JOIN ist damit: „Zeigen Sie die Mitarbeiter deren Wohnort auch ihr Arbeitsort ist.“ Damit aus der dritten Anfrage eine echte NATURAL JOIN-Simulation wird, müssten die doppelten Spalten mit der SELECT-Liste herausprojiziert werden.

### 5.5.5 Datensätze selektieren mit der WHERE-Klausel

WHERE-Klauseln legen anhand von Suchbedingungen fest, welche Zeilen aus den Datensätzen, die die FROM-Klausel bereitstellen, ausgewählt werden. Ausgewählt heißt, dass die Ergebnismenge der Anfrage nur die Datensätze enthält, für die die Bedingung der WHERE-Klausel zu TRUE ausgewertet wird. Ist das Ergebnis FALSE oder UNKNOWN, so wird der Datensatz verworfen (dreiwertige Logik). Zur Erinnerung verweisen wir noch mal darauf, dass das Ergebnis von Integritätsbedingungen anders interpretiert wird. Dort wird UNKNOWN wie TRUE behandelt (vgl. Abschnitt 5.3.2).

Wir werden in diesem Kapitel nur einfache Suchbedingungen behandeln, die keine weiteren Unterabfragen zulassen. Unterabfragen in Suchbedingungen sind das Thema im Abschnitt 5.5.9. Diese einfachen Suchbedingungen sind meist eine Folge von Prädikaten, die durch AND oder OR verbunden sind. Die Klammerung und die Priorität der Operatoren bestimmen dabei die Reihenfolge der Abarbeitung.

### Beispiel

Selektieren Sie alle Angestellten, die der Abteilung 2 oder keiner Abteilung angehören und in beiden Fällen weniger als 5000 Euro verdienen.

```
SELECT * FROM Angestellte
WHERE (Abt_Nr = 2 OR Abt_Nr IS NULL) AND Gehalt < 5000;
```

Tabelle 5.25

### Beispiele für mögliche Prädikate

Prädikat	Beispiel
Vergleichsprädikat	a.LiefNr = l.LiefNr
Intervallprädikat	Gehalt BETWEEN 4000 AND 6000
Ähnlichkeitsprädikat	Nachname LIKE 'M%'
Test auf NULL	LiefNr IS [NOT] NULL
IN-Bedingung	Artikel_Typ IN ('Rahmen', 'Hinterrad')

```

<WHERE_Klausel> ::= WHERE <Suchbedingung>
<Suchbedingung> ::= / <Ausdruck> <Vergleichsoperator> <Ausdruck>
| <Ausdruck> [NOT] BETWEEN <Ausdruck> AND <Ausdruck>
| <alphanumerischer Ausdruck> [NOT] LIKE <Schablone>
| <Ausdruck> IS [NOT] NULL
| <Ausdruck> [NOT] IN (Konstante [, Konstante]...)
| NOT <Suchbedingung>
| <Suchbedingung> AND <Suchbedingung>
| <Suchbedingung> OR <Suchbedingung>
( <Suchbedingung> ) /

```

Vergleichsoperatoren, Schablonen und Ausdrücke wurden schon im Abschnitt 5.2 und bei der CREATE TABLE-Anweisung im Abschnitt 5.3.1 eingeführt. Suchbedingungen lassen sich rekursiv definieren, da sie auf sich selbst verweisen können.

### 5.5.5.1 Einfache Vergleiche

Prädikate der Gestalt

```
<Vergleichsprädikat> ::= <Ausdruck> <Vergleichsoperator> <Ausdruck>
```

nennen wir einfache Vergleiche. Da es aufgrund fehlender Spaltenwerte recht häufig vorkommt, dass einer der Ausdrücke NULL bzw. UNKNOWN ist, stellt sich die Frage nach dem Ergebnis des Vergleichs in einem solchen Fall. Ist einer der Ausdrücke unbekannt, wird der Vergleich insgesamt zu NULL/UNKNOWN ausgewertet.

#### Beispiele

- Welche Angestellten gehören zur Abteilung 2?

```
SELECT Nachname FROM Angestellte WHERE Abt_Nr = 2;
```

- Welche Angestellten haben mehr als 40% Abzüge?

```
SELECT Nachname, Gehalt, Abzuege
FROM Angestellte
WHERE Abzuege > Gehalt * 0.4;
```

### 5.5.5.2 Logische Operatoren

In der WHERE-Klausel können mehrere Bedingungen durch die Operatoren AND, OR und NOT verknüpft werden.

#### Beispiele

- Welche Kunden wohnen in Gummersbach oder Köln?

```
SELECT Nachname
FROM Kunden
WHERE Ort = 'Köln' OR Ort = 'Gummersbach';
```

- Welche Lieferanten kommen weder aus Dortmund noch aus Köln?

```
SELECT * FROM Lieferanten
WHERE Ort <> 'Dortmund'
AND Ort <> 'Köln';
```

Es können aber auch sobrisante Anfragen formuliert werden, wie:

- Welche Angestellten verdienen mehr als ihr direkter Vorgesetzter?

```
SELECT Vorg.Nachname AS Vorgesetzter,
      Vorg.Gehalt AS Gehalt_Vorgesetzter,
      Ang.Nachname AS Angestellter,
      Ang.Gehalt AS Gehalt_Angestellter
FROM Angestellte Vorg, Angestellte Ang, Abteilungen Abt_Ang
WHERE Ang.Abt_Nr = Abt_Ang.Abt_Nr
AND Vorg.Ang_Nr = Abt_Ang.Leiter
AND Ang.Gehalt > Vorg.Gehalt;
```

Bei dieser Abfrage werden Sie feststellen, dass die Fahrrad-Welt Byce & Co. noch in Ordnung ist: Es gibt keinen Angestellten, der der letzten WHERE-Bedingung genügt.

Die Formatierung von Anfragen ist naturgemäß nicht standardisiert, bei Oracle gibt es unter SQL\*PLUS die COLUMN-Anweisung<sup>45</sup>, unter MySQL Formatstrings, die z.B. die Ausgabe für Datumsfelder formatieren.

Falls in einer Suchbedingung mehrere Abfragen vorkommen, muss man die Prioritätenreihenfolge beachten oder die Abarbeitungsreihenfolge explizit mit Klammern „()“ spezifizieren:

Tabelle 5.26

#### Prioritätenreihenfolge der Operatoren

Priorität	Operator
1	Alle Vergleichsoperatoren
2	NOT
3	AND
4	OR

#### Beispiel

Bestimmen Sie zum einen alle Angestellten, die Informatiker sind, und zum anderen alle Betriebswirte, die 6000 € oder mehr verdienen!

```
SELECT Nachname, Beruf, Gehalt
FROM Angestellte
WHERE Beruf = 'Informatiker'
OR Beruf = 'Betriebswirt'
AND Gehalt >= 6000;
```

45 Für die obige SELECT-Anweisung könnte z.B. unter Oracle-SQL\*PLUS mit

COLUMN Chef FORMAT a15  
COLUMN Gehalt\_chef FORMAT 99999  
COLUMN Angestellter FORMAT a15  
COLUMN Gehalt\_Angestellter FORMAT 99999  
eine Formatierung vorgenommen werden, um den Ausdruck besser lesbar zu gestalten.

Ergebnis:

Nachname	Beruf	Gehalt
Müller	Betriebswirt	10000
Heck	Betriebswirt	7500
Weber	Informatiker	6600
Frisch	Informatiker	5000
Frisch	Informatiker	5000
Käse	Informatiker	4500
Kussmann	Informatiker	4500
Bold	Informatiker	3500

Da die Priorität von AND höher ist als die des Operators OR, wird AND zuerst abgearbeitet.

Durch Klammerung kann man die Reihenfolge der Auswertung der Prädikate verändern und damit auch die Semantik der Anfrage, die nun heißt: Bestimmen Sie alle Informatiker und Betriebswirte, die 6000 € oder mehr verdienen:

```
SELECT Nachname, Beruf, Gehalt
FROM Angestellte
WHERE (Beruf = 'Informatiker' OR Beruf = 'Betriebswirt')
AND Gehalt >= 6000;
```

Ergebnis:

Nachname	Beruf	Gehalt
Müller	Betriebswirt	10000
Heck	Betriebswirt	7500
Weber	Informatiker	6600

### 5.5.3 Dreiwertige Logik

Bei den relationalen Datenbanken tritt noch eine Besonderheit durch die Zulassung von NULL-Werten auf. Es gibt bei SQL wie auch bei Oracle und MySQL eine dreiwertige Logik mit den Werten TRUE, FALSE und UNKNOWN (wird auch als NULL dargestellt, z.B. bei MySQL). Der Wahrheitswert UNKNOWN tritt auf, wenn ein Wert, der NULL ist, mit einem anderen Wert verglichen wird. Bei jedem der nachfolgenden Operatoren weisen wir explizit darauf hin, was sein Ergebnis ist, wenn einer der Operanden NULL/UNKNOWN ist. In der Praxis ist es manchmal bedauerlich, dass man nur einen unbekannten Zustand hat, da würde man auch gerne zwischen NULL im Sinne von „gibt es nicht“ und im Sinne von „nicht bekannt“ unterscheiden.

Tabelle 5.27

#### Wahrheitstabelle des AND-Operators

	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Tabelle 5.28

#### Wahrheitstabelle des OR-Operators

	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Tabelle 5.29

#### Wahrheitstabelle des NOT-Operators

	TRUE	FALSE	UNKNOWN
TRUE	FALSE	TRUE	UNKNOWN
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

#### Beispiel

```
SELECT Name, Telefonnr FROM Lieferanten;
```

```
SELECT Name, Telefonnr FROM Lieferanten
WHERE Telefonnr = '0221/985688'
OR Telefonnr <> '0221/985688';
```

Bei der TelefonNr sind NULL-Werte zugelassen, es kann also Lieferanten geben, deren Telefonnummer nicht in der Datenbank abgelegt ist. Die erste SELECT-Anweisung zeigt, dass es einen Lieferanten gibt, der keine Telefonnummer hat. Für diesen Datensatz liefert der Vergleich mit einem Wert den Wahrheitswert UNKNOWN. Bei der zweiten SELECT-Anweisung wird dieser Lieferant nicht ausgegeben, da nur diejenigen Lieferanten selektiert werden, für die die WHERE-Bedingung insgesamt zu TRUE ausgewertet wird (vgl. Abschnitt 5.5.7).

### 5.5.5.4 Der BETWEEN-Operator

Mit diesem Prädikat lassen sich Wertebereiche abfragen, die für Attribute vorgesehen sind.

```
<Suchbedingung>46 ::= <Ausdruck> [NOT] BETWEEN <Ausdruck> AND <Ausdruck>
```

Beim BETWEEN-Operator sind die Grenzwerte beim Vergleich mit eingeschlossen, es handelt sich damit um größer/gleich- und kleiner/gleich-Abfragen. Als Ausdrücke sind unter anderem Konstanten, Spaltennamen, Funktionsaufrufe und Anfrageausdrücke, die nur einen Ergebniswert liefern, zugelassen. Ist einer der drei Ausdrücke NULL, so ist das Ergebnis UNKNOWN.

#### Beispiele

```
SELECT Nachname, Vorname, Gehalt FROM Angestellte
WHERE Gehalt BETWEEN 3000 AND 3600;
```

#### Ergebnis:

Nachname	Vorname	Gehalt
Falser	Jonas	3600
Brunn	Ilse	3600
Bold	Leo	3500
Butz	Max	3500
Barthels	Isabel	3500
Schmidt	Hugo	3500

Bei Datumsfeldern kann der BETWEEN-Operator im jeweiligen Datumsformat auch auf DATE-Spalten angewendet werden:

```
SELECT Vorname, Nachname, Eintrittsdatum
FROM Angestellte
WHERE Eintrittsdatum BETWEEN '01.01.1995' AND '01.01.2000';
/* Oracle-Datumsformat, wenn die Spracheinstellung deutsch ist*/47
SELECT Vorname, Nachname, Eintrittsdatum
FROM Angestellte
WHERE Eintrittsdatum BETWEEN '1995-01-01' AND '2000-01-01';
/*MySQL-Datums-Format*/
SELECT Vorname, Nachname, Eintrittsdatum
FROM Angestellte
WHERE TO_CHAR(Eintrittsdatum, 'DD.MM.JJJJ')
      BETWEEN '01.01.1995' AND '01.01.2007';
/* auf der sicheren Seite und unabhängig von irgendwelchen Einstellungen: Konvertierung des Datums in eine Zeichenketten, Oracle-spezifisch*/
```

<sup>46</sup> Die hier und bei den anderen Operatoren definierte <Suchbedingung> ist ein Auszug der <Suchbedingung> vom Anfang dieses Kapitels.

<sup>47</sup> In SQL werden einzeilige Kommentare mit -- am Zeilenanfang markiert und mehrzeilige mit /\* \*/ begrenzt (vgl. auch PL/SQL im Abschnitt 7.1).

Bei alphanumerischen Spalten wird intern in ASCII-Code umgewandelt und dieser Wert mit den Grenzen des BETWEEN-Operators verglichen. Notwendig ist dabei die Verwendung des Wildcard „%“.

```
SELECT Vorname, Nachname, Eintrittsdatum
FROM Angestellte
WHERE Nachname BETWEEN 'F%' AND 'H%';
```

### 5.5.5.5 Der LIKE-Operator

Der LIKE-Operator dient zum Abfragen von Textmustern über Schablonen und Wildcards. Als Wildcards sind die im Abschnitt 5.2.4 definierten Symbole „%“ und „\_“ zulässig. „%“ steht für kein, ein oder mehrere Zeichen, „\_“ für ein einzelnes Zeichen. Ist der alphanumerische Ausdruck NULL, dann ist das Ergebnis des Vergleichs UNKNOWN.

```
<Suchbedingung> ::= <alphanumerischer Ausdruck> [NOT] LIKE <Schablone>
<Schablone> ::= Zeichenkette mit Wildcards (% oder _) als alphanumerische Konstante oder Variable
```

#### Beispiele

```
SELECT Nachname FROM Angestellte
WHERE Nachname LIKE 'W%';
```

```
SELECT Nachname FROM Angestellte
WHERE Nachname LIKE 'W%r';
```

```
SELECT Nachname FROM Angestellte
WHERE Nachname NOT LIKE '%r';
```

```
SELECT Nachname FROM Angestellte
WHERE Nachname LIKE '_e%';
```

### 5.5.5.6 Der IN-Operator

Mit dieser Bedingung wird getestet, ob ein Vergleichswert in einer Menge von Werten enthalten ist. Ist der Ausdruck NULL, so ist das Ergebnis UNKNOWN.

```
<Suchbedingung> ::= <Ausdruck> [NOT] IN (Konstante [, Konstante]... )
```

#### Beispiel

Welche Kunden wohnen in Köln oder Gummersbach?

```
SELECT * FROM Kunden WHERE Ort IN ('Köln', 'Gummersbach');
```

### Regeln des IN-Operators:

- In der hier vorgestellten Form darf die Menge nach „IN“ keine Ausdrücke in Form von Variablen oder Operatoren enthalten (4 \* 7 z.B. ist nicht erlaubt), im Abschnitt 5.5.9 wird zusätzlich eine IN-Syntax vorgestellt, die Unterabfragen erlaubt.
- Alle Konstanten haben den gleichen Datentyp.
- Die Menge darf nicht zwei gleiche Konstanten enthalten.

### 5.5.7 Das NULL-Prädikat

Mit diesem Prädikat ist eine Abfrage möglich, ob ein Ausdruck, z.B. eine Spalte leer, also NULL ist. Dieser Operator ist explizit dafür gedacht, das Auftreten von NULL in einer Bedingung in TRUE oder FALSE umzuwandeln.

```
<Suchbedingung> ::= <Ausdruck> IS [NOT] NULL
```

#### Beispiele

- Welche Angestellte gehören zu keiner Abteilung?

```
SELECT * FROM Angestellte WHERE Abt_Nr IS NULL;
```

- Welche Lieferanten haben das Feld Telefonnr mit einem Wert versorgt?

```
SELECT * FROM Lieferanten WHERE Telefonnr IS NOT NULL;
```

Mit der **Oracle-NVL-Funktion** kann ein NULL-Wert durch einen beliebigen anderen Wert ersetzt werden. Ist jedoch eine Telefonnummer gegeben, so wird diese angezeigt. Nur wenn keine gegeben ist, wird der Text „Ohne Tel\_Nr“ ausgegeben.

```
SELECT NVL(Telefonnr, 'Ohne Tel_Nr') FROM Lieferanten;
```

Dieser Funktion entspricht unter MySQL die **IFNULL-Funktion**:

```
SELECT IFNULL(Telefonnr, 'Ohne Tel_Nr') FROM Lieferanten;
```

### 5.5.6 SQL-Gruppenfunktionen

Bei Anfragen ist im Spaltenausdruck der SELECT-Klausel wie auch in der HAVING-Klausel (vgl. Abschnitt 5.5.7) unter bestimmten Voraussetzungen die Verwendung der SQL-Gruppenfunktionen MAX, MIN, SUM, AVG und COUNT aus Abschnitt 5.2.3 zulässig.

```
<Gruppenfunktion> ::=  
  ( COUNT(*)  
  | <Gruppenfunktionstyp> ( [ALL | DISTINCT] <numerischer Ausdruck> ) )  
  
<Gruppenfunktionstyp> ::= / COUNT | MIN | MAX | SUM | AVG /
```

#### Anmerkungen:

- COUNT(\*) ermittelt die Anzahl der Datensätze einer Tabelle.
- Als numerischer Ausdruck sind z.B. numerische Spalten, Konstanten, Berechnungen zugelassen (vgl. Abschnitt 5.2).
- Die Funktionen COUNT(Spaltenname), SUM (Spaltenname) und AVG(Spaltenname) beziehen bei DISTINCT nur die unterschiedlichen Werte einer Spalte mit ein.
- Bei ALL werden alle Spaltenwerte, auch Duplikate, berücksichtigt; dies ist die DEFAULT-Option.
- NULL-Werte werden bei den Gruppenfunktionen nicht berücksichtigt. Bei COUNT (Spaltenname) heißt das, die Datensätze, die in dieser Spalte den Wert NULL haben, werden nicht gezählt, ebenso bei der AVG- und auch bei der SUM-Funktion.
- Zum Vergleich: Bei einer Addition mit dem „+“-Operator wird das Ergebnis NULL, wenn einer der Operanden NULL ist, bei SUM und AVG bleiben solche Datensätze unberücksichtigt, für alle Nicht-NULL-Datensätze wird die Berechnung weitergeführt.
- Gruppenfunktionen können nicht in WHERE-Bedingungen verwendet werden, da sie sich nicht auf einen einzelnen Datensatz, sondern auf mehrere beziehen.

#### Beispiele

- Wie viele Angestellte hat die Abteilung 2?

```
SELECT COUNT(*) FROM Angestellte WHERE Abt_Nr = 2;
```

#### Ergebnis:

```
COUNT(*)
```

```
8
```

- Wie hoch ist das Jahresgehalt aller Angestellten der Abteilung 2?

```
SELECT SUM(Gehalt) * 12 FROM Angestellte WHERE Abt_Nr = 2;
```

#### Ergebnis:

```
SUM(Gehalt)*12
```

```
376800
```

- Wie hoch ist das durchschnittliche Gehalt der Informatiker?

```
SELECT AVG(Gehalt)  
FROM Angestellte  
WHERE Beruf = 'Informatiker';
```

#### Ergebnis:

```
AVG(Gehalt)
```

```
4850
```

- Ermitteln Sie das durchschnittliche Gehalt, das maximale Gehalt, das minimale Gehalt und die Summe aller Gehälter aller Angestellten!

```
SELECT AVG(Gehalt), MAX(Gehalt), MIN(Gehalt), SUM(Gehalt)
FROM Angestellte;
```

#### Ergebnis:

AVG(Gehalt)	MAX(Gehalt)	MIN(Gehalt)	SUM(Gehalt)
4601,11	10000	500	124230

- Ermitteln Sie die Summe über alle Gehälter, wobei auch Gehälter, die mehrfach vorkommen, berücksichtigt werden!

```
SELECT SUM(ALL Gehalt) FROM Angestellte;
```

oder

```
SELECT SUM(Gehalt) FROM Angestellte;
```

Ergebnis: 124230 €

- Ermitteln Sie die Summe über alle Gehälter, wobei Gehälter, die mehrfach vorkommen, nur einmal berücksichtigt werden!

```
SELECT SUM(DISTINCT Gehalt) FROM Angestellte;
```

Ergebnis: 96130 €

Bei den Gruppenfunktionen kann man zwar die Anzahl der Datensätze, die in die Berechnung eingehen, durch die WHERE-Bedingung einschränken. Wenn man jedoch gruppieren will, also Berechnungen für Gruppen von Zeilen aufstellen möchte, braucht man zusätzlich die GROUP BY-Klausel.

### 5.5.7 Gruppierung mit den GROUP BY- und HAVING-Klauseln

Mit der GROUP BY-Klausel werden die Ergebnisdatensätze nach der Auswertung der WHERE-Klausel in Gruppen eingeteilt, so dass innerhalb der Gruppen die Werte der GROUP BY-Spalten gleich sind.

```
<GROUP BY Klausel> ::= GROUP BY Spaltenname [, Spaltenname ]...
<Having Klausel> ::= HAVING <Suchbedingung>
```

#### Beispiel

- Berechnen Sie die Summe über alle Gehälter der Angestellten der gleichen Abteilung!

```
SELECT Abt_Nr, SUM(Gehalt) FROM Angestellte GROUP BY Abt_Nr;
```

#### Ergebnis:

Abt_Nr	SUM(Gehalt)
1	10000
2	31400
3	20310
4	14720
5	21800
6	26000

Mit der HAVING-Klausel wird dann noch für die einzelnen Gruppen eine Suchbedingung formuliert.

#### Beispiel

- Berechnen Sie die Summe über alle Gehälter derjenigen Abteilungen, die mindestens sechs Angestellte haben!

```
SELECT Abt_Nr, SUM(Gehalt) AS SUMME
FROM Angestellte
GROUP BY Abt_Nr
HAVING COUNT(*) > 5;
```

#### Ergebnis:

Abt_Nr	SUMME
2	31400
6	26000

#### Anmerkungen:

- In der SELECT-Klausel dürfen nur zwei Arten von Spalten vorkommen:
  - die, die mit einer Gruppenfunktion versehen sind, und
  - die anderen müssen in der GROUP-BY-Klausel enthalten sein.
- In der Ausgabe wird über die GROUP BY-Spalten aufsteigend sortiert.
- Die Suchbedingung entspricht der Suchbedingung der WHERE-Klausel mit der Ausnahme, dass in der HAVING-Bedingung auch die Gruppenfunktionen zulässig sind, was bei der WHERE-Klausel nicht der Fall ist.

#### Weitere Beispiele

- Wie viele Angestellte haben die einzelnen Abteilungen der Firma Byce & Co.?

```
SELECT Abt_Nr, COUNT(Ang_Nr)
FROM Angestellte
GROUP BY Abt_Nr;
```

Ergebnis:

Abt_Nr	COUNT(Ang_Nr)
1	1
2	8
3	5
4	3
5	4
6	6

■ Welche Abteilungen haben mehr als vier Mitarbeiter?

```
SELECT Abt_Nr, COUNT(Ang_Nr)
FROM Angestellte
GROUP BY Abt_Nr;
HAVING COUNT(*) >= 5;
```

Ergebnis: Abteilungen 2, 3 und 6

■ Berechnen Sie das durchschnittliche Gehalt der Abteilungen!

```
SELECT Abt_Nr, AVG(Gehalt) FROM Angestellte GROUP BY Abt_Nr;
```

Ergebnis:

Abt_Nr	AVG(Gehalt)
1	10000.00
2	3925.00
3	4062.00
4	4906.67
5	5450.00
6	4333.33

### Beispiel

mit Fehler

```
SELECT Abt_Nr, Nachname, AVG(Gehalt)
FROM Angestellte
GROUP BY Abt_Nr
```

Diese Anweisung ergibt unter Oracle einen Syntaxfehler, da der Nachname weder eine GROUP-BY-Spalte ist noch mit einer Gruppenfunktion versehen ist. Unter MySQL wird die Anweisung ausgeführt, indem einfach der erste Angestellte der Abteilung mit Nachname selektiert wird, unabhängig davon, ob es mehrere Angestellte in einer Abteilung gibt.

Es soll noch einmal schrittweise anhand eines Beispiels verdeutlicht werden, wie eine GROUP-BY-Anweisung in einer SELECT-Anfrage ausgeführt wird. Wir nutzen das Beispiel gleichzeitig, um die Ausführungsreihenfolge der verschiedenen Klauseln einer Anfrage aufzuzeigen:

Tabelle 5.30

### Die Reihenfolge der Klauseln der SELECT-Anweisung

Ausführungsreihenfolge Priorität	Klausel	Funktion
1	FROM	Kartesisches Produkt
2	WHERE	Selektions- und Verknüpfungsbedingung für die Datensätze
3	GROUP BY	Gruppierung
4	HAVING	Bedingung für die Gruppe
5	ORDER BY <sup>48</sup>	Sortierung
6	SELECT	Projektion der Spalten

Welche Kunden aus Köln haben mehr als einen Auftrag erteilt?

1 Schritt: FROM-Klausel Kunden x Auftraege (Priorität 1)

```
SELECT K.*, A.* FROM Kunden K, Auftraege A;
```

Ergebnis: Diese Anfrage liefert als kartesisches Produkt 42 Datensätze mit 17 Spalten.

2 Schritt: WHERE-Klausel dazunehmen (Priorität 2)

```
SELECT K.*, A.*
FROM Kunden K, Auftraege A
WHERE K.Kun_Nr = A.Kun_Nr
AND K.Ort = 'Köln';
```

Ergebnis:

Kun_Nr	Nachname	Ort	Restliche Spalten der Kundentabelle	Auftrags Nr	Restliche Spalten der Auftragstabelle
1	Tholler	Köln	...	1	...
2	Falk	Köln	...	2	...
2	Falk	Köln	...	4	...
3	Müller	Köln	...	5	...
2	Falk	Köln	...	6	...

3 Schritt: GROUP BY hinzufügen (Priorität 3)

```
SELECT K.Kun_Nr, K.Nachname, K.Ort, COUNT(A.AuftragsNr)
FROM Kunden K, Auftraege A
WHERE K.Kun_Nr = A.Kun_Nr AND K.Ort = 'Köln'
GROUP BY K.Kun_Nr, K.Nachname, K.Ort;
```

48 vgl. Abschnitt 5.5.8

Ergebnis:

Kun_Nr	Nachname	Ort	Restliche Spalten der Kundentabelle	COUNT(A.AuftragsNr)
1	Tholler	Köln	.....	1
2	Falk	Köln	.....	3
3	Müller	Köln	.....	1

#### 4. Schritt: HAVING-Klausel (Priorität 4)

```
SELECT K.Kun_Nr, K.Nachname, K.Ort, COUNT(A.AuftragsNr)
FROM Kunden K, Auftraege A
WHERE K.Kun_Nr = A.Kun_Nr AND K.Ort = 'Köln'
GROUP BY K.Kun_Nr, K.Nachname, K.Ort
HAVING COUNT(*) > 1;
```

Ergebnis:

Kun_Nr	Nachname	Ort	Restliche Spalten der Kundentabelle	COUNT(A.AuftragsNr)
2	Falk	Köln	.....	3

#### 5. Schritt: ORDER BY-Klausel (Priorität 5)

Fällt hier weg, da nur ein Datensatz ausgewählt wurde.

#### 6. Schritt: SELECT-Klausel (Priorität 6)

```
SELECT K.Kun_Nr, K.Nachname, K.Ort, COUNT(A.AuftragsNr)
FROM Kunden K, Auftraege A
WHERE K.Kun_Nr = A.Kun_Nr AND K.Ort = 'Köln'
GROUP BY K.Kun_Nr, K.Nachname, K.Ort
HAVING COUNT(*) > 1;
```

Ergebnis:

Kun_Nr	Nachname	Ort	COUNT(A.AuftragsNr)
2	Falk	Köln	3

### 5.5.8 Sortieren mit der ORDER BY-Klausel

Die ORDER BY-Klausel bietet die Möglichkeit, die Ausgabe beliebig zu sortieren, wobei ASC für aufsteigend, DESC für absteigend steht. Bei alphanumerischen Zeichen wird nach dem ASCII-Code sortiert. Diese Funktionalität ist in der relationalen Algebra nicht zu finden (vgl. Abschnitt 4.1.3). Die DEFAULT-Option ist ASC.

<ORDER BY Klausel> ::= ORDER BY <Sortierung>

<Sortierung> ::= Spaltenname [DESC | ASC] [, Spaltenname [DESC | ASC]... ]

#### Beispiel

Geben Sie eine Liste aller Kunden aus, die nach Ort absteigend geordnet sind und dann noch mal aufsteigend nach Nachname:

```
SELECT Ort, Nachname, Vorname
FROM Kunden
ORDER BY Ort DESC, Nachname ASC;
```

Ergebnis:

Ort	Nachname	Vorname
Köln	Falk	Bernhardt
Köln	Franz	Helga
Köln	Müller	Tobias
Köln	Tholler	Andreas
Gummersbach	Sündbald	Hannelore
Gummersbach	Wal	Birgit
Gladbeck	Tisch	Hartmut

#### 5.5.9 Unterabfragen in einer SELECT-Anweisung

Eine Unterabfrage ist eine SELECT-Anweisung in einer SELECT-Anweisung. Sie ist in der WHERE-, HAVING- und FROM-Klausel vorgesehen. Wird sie in einer Suchbedingung verwendet, so wird das Ergebnis der Unterabfrage mit einem Ausdruck verglichen. Als Vergleichsoperatoren sind alle Vergleichsoperatoren aus Abschnitt 5.2 zulässig. Zu beachten ist aber auch hier wieder das Verhalten, wenn NULL-wertige Spalten auftreten oder aber die Ergebnismenge der Unteranfrage leer ist. Syntaktisch ist eine Unteranfrage ein Anfrageausdruck wie er im Abschnitt 5.5.11 eingeführt wird, so dass sich folgende formale Definition ergibt. Wir führen hier den Begriff der Unterabfrage explizit ein, weil er in vielen anderen Quellen erwendet wird (vgl. [Oracle SQL 2005] und [ANSI SQL 2003b]).

<Unterabfrage> ::= <Anfrageausdruck>

Für die Verwendung von Unterabfragen in Suchbedingungen der WHERE- und HAVING-Klauseln stellen wir nun die Operatoren ANY bzw. ALL, IN und EXIST vor.

##### 5.5.9.1 Die ANY | ALL-Bedingung

<Suchbedingung> ::=  
 <Ausdruck> <Vergleichsoperator> [ALL | ANY] (<Unterabfrage>)

Es gibt in diesem Fall drei Möglichkeiten:

- Bei Verwendung von „ALL“ muss die Bedingung für **alle** Ergebnisdatensätze der Unterabfrage wahr sein.
- Bei Verwendung von „ANY“ muss die Bedingung für mindestens **einen** Ergebnisdatensatz der Unterabfrage wahr sein.
- Fehlen sowohl „ALL“ als auch „ANY“, darf die Unterabfrage nur einen **einzelnen** Datensatz, nicht aber eine Menge von Datensätzen zurückliefern.

### Beispiele



- Wer verdient mehr als alle Angestellten von Herrn Schmidt, der die Angstelltennummer 4 hat?

```
SELECT Nachname, Gehalt
FROM Angestellte
WHERE Gehalt > ALL (SELECT Gehalt
                     FROM Angestellte a, Abteilungen b
                     WHERE a.Abt_Nr = b.Abt_Nr
                     AND Leiter = 4);
```

- Wer verdient mehr als irgendein Angestellter von Herrn Schmidt?

```
SELECT Nachname, Gehalt
FROM Angestellte
WHERE Gehalt > ANY (SELECT Gehalt
                     FROM Angestellte a, Abteilungen b
                     WHERE a.Abt_Nr = b.Abt_Nr
                     AND Leiter = 4);
```

- Gibt es einen Angestellten, der genau das durchschnittliche Gehalt über alle Gehälter verdient?

```
SELECT Nachname, Gehalt FROM Angestellte
WHERE Gehalt = (SELECT AVG(Gehalt) FROM Angestellte);
```

- Welche Angestellten haben nach Herrn Wanne bei der Firma Byce & Co. angefangen?

```
SELECT Nachname FROM Angestellte
WHERE Eintrittsdatum > (SELECT Eintrittsdatum FROM Angestellte
                           WHERE Nachname = 'Wanne');
```

Bei Unterabfragen mit einem Vergleichsoperator und ohne ANY und ALL muss das Ergebnis der Unterabfrage einen **einzelnen** Wert liefern, um den Vergleich durchführen zu können. Die Abfrage

```
SELECT Nachname FROM Angestellte
WHERE Eintrittsdatum > (SELECT Eintrittsdatum FROM Angestellte
                           WHERE Nachname = 'Weber');
```

führt zu einem Ausführungsfehler und wird vom Datenbankmanagementsystem zurückgewiesen:

FEHLER in Zeile 2:  
ORA-01427: Unterabfrage für eine Zeile liefert mehr als eine Zeile.

Die Überprüfung hängt also von den in der Tabelle vorhandenen Daten ab. Der Nachname „Weber“ kommt eben zweimal in der Byce & Co. Datenbank vor, während der Nachname „Wanne“ einmalig ist. Korrekt würde die Abfrage lauten:

```
SELECT Nachname FROM Angestellte
WHERE Eintrittsdatum > ALL (SELECT Eintrittsdatum
                             FROM Angestellte
                             WHERE Nachname = 'Weber');
```

### Korrelierte Unterabfragen

Eine Unterabfrage heißt **korreliert**, wenn es Spalten der äußeren SELECT-Anweisung gibt, die mit den Spalten der inneren SELECT-Anweisung in Beziehung gesetzt sind.

### Beispiel

```
SELECT Nachname, Gehalt
FROM Angestellte
WHERE Gehalt > ANY (SELECT Gehalt
                     FROM Angestellte a, Abteilungen b
                     WHERE a.Abt_Nr = b.Abt_Nr
                     AND Leiter = 4);
```

Diese Abfrage ist korreliert, weil die Gehaltsspalten der Ober- und Unterabfrage in Beziehung gesetzt sind. Abfragen mit dem ANY/ALL-Operator und dem nachfolgenden IN-Operator erzwingen aufgrund ihrer Syntax bereits die Korrelation. Schwieriger ist es, die Korrelation beim EXISTS-Operator einzuhalten, da man sie in der WHERE-Klausel selbst programmieren muss. Jedoch machen EXISTS-Unterabfragen meist nur Sinn, wenn sie korreliert sind.

### 5.5.9.2 Die IN-Bedingung

<Suchbedingung> ::= <Ausdruck> [ NOT ] IN (<Unterabfrage>)

Das Ergebnis der Anfrage ist TRUE, wenn der Ausdruck in der Ergebnismenge der Unteranfrage gefunden wird. Ist die Ergebnismenge leer und wird mit IN verglichen, dann wird auf FALSE erkannt. Wird jedoch bei leerer Ergebnismenge mit NOT IN verglichen, dann wird auf TRUE erkannt. Ist hingegen der Ausdruck NULL, so wird [NOT] IN zu UNKNOWN ausgewertet, mit der Konsequenz, dass in beiden Fällen der Datensatz nicht in die Ergebnismenge kommt. Der Ausdruck muss kompatibel zur Ergebnismenge der Unterabfrage sein, was heißt, dass die Datentypen und die Anzahl der selektierten Spalten mit den Datentypen und der Anzahl beim Ausdruck übereinstimmen müssen. Die Spaltennamen müssen nicht identisch sein.

**Beispiel**

Welche Kunden haben keine Aufträge erteilt?

```
SELECT * FROM Kunden
WHERE Kun_Nr NOT IN (SELECT Kun_Nr FROM Auftraege);
```

**5.5.9.3 Die EXISTS-Bedingung**

Die EXISTS-Bedingung wird als wahr erkannt, wenn die Unterabfrage mindestens einen Datensatz selektiert. Mit EXISTS wird also der Existenzquantor umgesetzt. Ist die Untermenge leer und wird mit EXISTS verglichen, dann ist das Ergebnis FALSE, wird mit NOT EXISTS verglichen, ist das Ergebnis für alle Datensätze der oberen Anfrage TRUE. Das gleiche Verhalten tritt auf, wenn eine der korrelierenden Spalten NULL ist, denn dann ist die Untermenge leer. Bei den meisten Anfragen führt nur eine Korrelation zu semantisch richtigen Ergebnissen.

```
<Suchbedingung> ::= EXISTS ( <Unterabfrage> )
```

**Beispiele**

- Selektieren Sie alle Artikel, für die mindestens ein Auftrag vorliegt!

```
SELECT DISTINCT TNr, Bezeichnung
FROM Artikel a
WHERE EXISTS (SELECT * FROM Auftragspositionen p
               WHERE a.TNr = p.TNr);
```

- Selektieren Sie alle Artikel, für die kein Auftrag vorliegt!

```
SELECT DISTINCT TNr, Bezeichnung
FROM Artikel a
WHERE NOT EXISTS (SELECT * FROM Auftragspositionen p
                   WHERE a.TNr = p.TNr);
```

Leider gibt es in SQL keine einfache Abbildung des ALL-Quantors. Um Abfragen, die diesen Quantor benutzen, umzusetzen, muss man mit einer doppelten Verneinung der EXISTS-Bedingung, also mit einem ziemlichen Umweg, arbeiten.

- Welche Lieferanten können alle Materialien liefern?

Diese Aussage heißt in der doppelten Verneinung: „Welche Lieferanten können kein Material nicht liefern?“ Denn wenn es kein Material gibt, dass sie nicht liefern können, können sie alle Materialien liefern.

```
SELECT l.Lief_Nr, l.Name
FROM Lieferanten l
WHERE NOT EXISTS (SELECT * FROM Teile t
                  WHERE t.Typ = 'Material'
                  AND NOT EXISTS (SELECT *
                                  FROM Lieferprogramme p
                                  WHERE l.Lief_Nr = p.Lief_Nr
                                  AND t.TNr = p.TNr));
```

**Ergebnis:**

Lief_Nr	Name
2	IMPORT-Wendel

Eine andere Möglichkeit, die Abfrage nach allen Materialien zu befriedigen, besteht darin, zu zählen, wie viele Materialien es insgesamt gibt und wie viele Materialien die einzelnen Lieferanten liefern können. Wenn ein Lieferant die gleiche Anzahl Materialien liefert, die es gibt, kann man logisch schließen, dass er alle Materialien liefert.

```
SELECT l.Lief_Nr, l.Name, COUNT(*)
FROM Lieferanten l, Lieferprogramme p, Teile t
WHERE t.TNr = p.TNr
AND l.Lief_Nr = p.Lief_Nr
AND t.Typ = 'Material'
GROUP BY l.Lief_Nr, l.Name
HAVING COUNT(*) = (SELECT COUNT(*) FROM Teile
                     WHERE Typ = 'Material');
```

Unterabfragen in SELECT-Anweisungen lassen sich beliebig schachteln. Allerdings ist gerade bei Verwendung der EXISTS-Bedingung in großen Tabellen oder einer hohen Verschachtelungstiefe mit einer schlechten Performance zu rechnen.

**5.5.9.4 Unterabfragen in der FROM-Klausel**

Wie das Syntaxdiagramm der FROM-Klausel im Abschnitt 5.5.3 gezeigt hat, kann dort anstelle von Tabellennamen ein Anfrageausdruck stehen. Für die Ergebnismenge einer solchen Unteranfrage muss dann allerdings eine Abkürzung als Name vergeben werden. Dies gibt es für SQL2003 und Oracle, nicht aber bei MySQL.

**Beispiel**

Für welche Artikel liegt mindestens ein Auftrag vor?

```
SELECT DISTINCT a.TNr, a.Bezeichnung
FROM Artikel a, (SELECT * FROM Auftragspositionen) p
WHERE a.TNr = p.TNr;
```

Die obige SELECT-Anweisung liefert das gleiche Ergebnis wie die folgende Anweisung mit dem EXISTS-Prädikat:

```
SELECT DISTINCT TNr, Bezeichnung
FROM Artikel a
WHERE EXISTS (SELECT * FROM Auftragspositionen p
               WHERE a.TNr = p.TNr);
```

In SQL gibt es gerade bei der SELECT-Anweisung oft verschiedene Syntaxmöglichkeiten, die das gleiche Resultat an Datensätzen liefern. Einen Einblick erhalten Sie im Abschnitt 5.5.9.6.

**5.5.9.5 Unterabfragen in der UPDATE-Anweisung**

Auch in einem UPDATE-Statement kann eine Unterabfrage stehen, mit der man Werte aus einer Tabelle in eine andere übertragen kann:

```
UPDATE Angestellte
SET Gehalt = (SELECT AVG(Gehalt) FROM Angestellte)
WHERE Abt_Nr = 1;
```

Diese Möglichkeit ist nicht auf eine einzelne Spalte beschränkt. Die Unterabfrage kann auch mehrere Spalten liefern. Allerdings muss dann natürlich hinter der SET-Klausel des UPDATE-Statement die entsprechende Spaltenzahl stehen:

```
UPDATE Angestellte
SET (Gehalt, Abzuege) = (SELECT AVG(Gehalt), MAX(Abzuege)
                           FROM Angestellte)
WHERE Abt_Nr = 1;
```

UPDATE-Anweisungen mit SELECT-Zuweisungen sind im SQL-Standard und unter Oracle vorgesehen, aber noch nicht unter MySQL.

### 5.5.9.6 Unterschiedliche SELECT-Abfragen mit dem gleichen Ergebnis

Oft gibt es zu einer Anfrage alternative SELECT-Anweisungen, die das gleiche Ergebnis liefern, obwohl sie unterschiedliche WHERE-Klauseln haben. Dies ist gerade bei der Verwendung von ANY/ALL, IN und EXISTS der Fall, aber nicht nur dort. Eine Tabelle mit alternativen Abfragen ist z.B. in Fritze<sup>49</sup> enthalten und wurde hier ergänzt. Diese Übersicht erhebt keinen Anspruch auf Vollständigkeit.

Tabelle 5.31

Alternative SELECT-Anweisungen	
Alternative 1	Alternative 2
WHERE X = ANY (SELECT Spalte FROM Tabelle)	WHERE X IN (SELECT Spalte FROM Tabelle)
WHERE X [NOT] IN (SELECT Spalte FROM Tabelle)	WHERE [NOT] EXISTS (SELECT * FROM Tabelle WHERE X = Tabelle.Spalte)
WHERE X <[=] ANY (SELECT Spalte FROM Tabelle)	WHERE X <[=] (SELECT MAX (Spalte) FROM Tabelle)
WHERE X >[=] ANY (SELECT Spalte FROM Tabelle)	WHERE X >[=] (SELECT MIN (Spalte) FROM Tabelle)
WHERE X <> ANY (SELECT Spalte FROM Tabelle)	Die Bedingung kann wegfallen, falls die Unteranfrage mehr als eine Zeile hat, denn dann ist die Bedingung immer wahr.
WHERE X = ALL (SELECT Spalte FROM Tabelle)	Die Bedingung liefert immer FALSE, falls die Unteranfrage mehr als eine Zeile hat.
WHERE X <[=] ALL (SELECT Spalte FROM Tabelle)	WHERE X <[=] (SELECT MIN(Spalte) FROM Tabelle)
WHERE X >[=] ALL ( SELECT Spalte FROM Tabelle)	WHERE X >[=] (SELECT MAX(Spalte) FROM Tabelle)

49 vgl. [Fritze et al. 2002, S. 117 ff.]

### Alternative SELECT-Anweisungen (Fortsetzung)

#### Alternative 1

```
WHERE X <> ALL  
(SELECT Spalte FROM Tabelle)  
WHERE EXISTS  
(SELECT Spalte FROM Tabelle)  
WHERE NOT EXISTS  
(SELECT Spalte FROM Tabelle)  
WHERE A BETWEEN X AND Y  
WHERE A IN (X, Y)
```

#### Alternative 2

```
WHERE X NOT IN  
(SELECT Spalte FROM Tabelle)  
WHERE 0 <>  
(SELECT COUNT(*) FROM Tabelle)  
WHERE 0 =  
(SELECT COUNT(*) FROM Tabelle)  
WHERE A >= X AND A <= Y  
WHERE A = X OR A = Y
```

### 5.5.10 Mengenoperationen auf Tabellen

SQL arbeitet mengenbezogen und daher ist es nicht verwunderlich, dass die Ergebnisse einer SELECT-Anweisung (vgl. Abschnitt 5.5.1) noch mal auf oberster Ebene mit den Mengenoperationen Vereinigung, Durchschnitt und Differenz versehen werden können.

Tabelle 5.32

### Bezeichnung der Mengenoperationen auf Tabellen

Mengenoperation	SQL	Oracle	MySQL
Vereinigung	UNION	UNION	UNION
Durchschnitt	INTERSECT	INTERSECT	
Differenz	EXCEPT	MINUS	

```
<Anfragemengenausdruck> ::=  
| <SELECT Grundform>  
| <Anfragemengenausdruck> UNION <Anfragemengenausdruck>  
| <Anfragemengenausdruck> INTERSECT <Anfragemengenausdruck>  
| <Anfragemengenausdruck> / EXCEPT | MINUS | <Anfragemengenausdruck>;
```

Die beteiligten SELECT-Ausdrücke müssen die gleichen Spaltendefinitionen haben. Die Spalten müssen nicht gleich heißen, zwingend ist nur, dass die Anzahl übereinstimmt, dass die Datentypen und Inhalte kompatibel sind sowie die Reihenfolge übereinstimmt. Diese Eigenschaft nennt man vereinigungskonform und gilt auch für die Differenz und den Durchschnitt.

### Beispiele

- Gesucht sind alle Nachnamen von Personen, die Angestellte oder Kunden sind!

```
SELECT Nachname FROM Angestellte
UNION
SELECT Nachname FROM Kunden;
```

- Gesucht sind alle Nachnamen von Angestellten, die auch Kunden sind!

```
SELECT Nachname FROM Angestellte
INTERSECT
SELECT Nachname FROM Kunden;
```

- Gesucht sind alle Nachnamen von Angestellten, die aber keine Kunden sind!

```
SELECT Nachname FROM Angestellte
MINUS
SELECT Nachname FROM Kunden;
```

Während Vereinigung und Durchschnitt kommutativ sind, macht die Reihenfolge der Anfragen bei der Differenz sehr wohl einen Unterschied. Vertauscht man im letzten Beispiel die Angestellten und die Kundentabelle, dann heißt die Anfrage: Gesucht sind alle Nachnamen von Kunden, die keine Angestellten sind.

### 5.5.11 Anfrageausdruck mit WITH-Klausel

Die SQL-Anfrage, wie wir sie bis jetzt kennengelernt haben (vgl. Abschnitt 5.5.10), ist immer noch nicht vollständig. Es fehlt ein Konstrukt zur Definition temporärer Hilfsichten, auf die dann in der Anfrage wie auf die persistenten Sichten (CREATE VIEW vgl. Abschnitt 5.3.5) zugegriffen werden kann. Zwei Ziele werden mit diesen Hilfsichten verfolgt. Zum einen dienen sie dazu, die Anfrage übersichtlicher und strukturierter zu programmieren, und zum anderen werden damit rekursive Anfragen formulierbar. Die Möglichkeit zur Rekursion ist aber nicht auf Hilfsichten beschränkt, auch persistente Sichten können rekursive Anfragen beinhalten: CREATE [RECURSIVE] VIEW... Da die Konzepte der rekursiven Sichten und Hilfsichten sehr analog sind, beschränken wir uns hier auf die Erläuterungen zu den Hilfsichten.

Im Byce & Co.-Schema gibt es einen Fall von Rekursion, die Strukturtabelle. Bis zur Einführung der WITH-Klausel waren rekursive Anfragen wie „Welche mittelbaren und unmittelbaren Unterteile gehören zu welchen Oberteilen?“ nur sehr umständlich zu formulieren und auch nur für eine feste Anzahl von Ebenen möglich. Für n Ebenen im Strukturbaum muss man dann in den FROM- und WHERE-Klauseln die Tabelle n+1-mal miteinander verknüpfen (vgl. Abschnitt 5.5.13). Mit der neuen Form der rekursiven Anfrage wird die Anfrage intuitiver, übersichtlicher und sie kann für eine unbekannte Anzahl rekursiver Ebenen angewendet werden.

```
<Anfrageausdruck> ::= [ <WITH Klausel> ] <Anfragemengenausdruck>;
<WITH Klausel> ::=  
    WITH [ RECURSIVE ] <WITH Element> [ , <WITH Element> ]...
<WITH Element> ::=  
    Anfragename [ ( Spaltenname [ , Spaltenname ]... ) ]  
    AS ( <Anfrageausdruck> ) [ <Suche oder Zyklus Klausel> ]
```

Der vollständige SELECT-Ausdruck besteht aus zwei Teilen, der optionalen WITH-Klausel und dem <Anfragemengenausdruck>, wobei dieser eine SELECT-Anfrage ist, wie wir sie im Abschnitt 5.5.10 definiert haben, mit einer FROM-Klausel wie aus Abschnitt 5.5.9, also Anfragen in ihrer vollständigen Komplexität. Neu ist die WITH-Klausel, für die im Falle rekursiver Hilfsichten die Option RECURSIVE angegeben wird. Ihre Elemente sind die temporären Hilfsichten.

Eine solche Hilfsicht <WITH Element> wird mit einem Namen benannt, gefolgt von einer optionalen Liste mit Spaltennamen. Dem Schlüsselwort AS folgt eine Anfrage, mit der die Daten der Sicht zusammengestellt werden. Die Syntax gleicht der der CREATE VIEW-Anweisung aus Abschnitt 5.3.5. Zu beachten ist, dass dieser <Anfrageausdruck> selbst wieder WITH-Klauseln enthalten kann. Solche Schachtelungen sollten geübten Entwicklern vorbehalten sein, da man dabei schnell den Überblick verlieren kann.

Rekursive Anfragen sind immer zweigeteilt, die erste Anfrage ermittelt die Quelldaten, von denen dann die rekursiv abhängigen Daten abgeleitet werden. Die beiden Ergebnismengen werden dann mit einem UNION-Operator zusammengeführt. Da das Erkennen von nicht terminierenden rekursiven Anfragen ein sehr schwieriges Problem ist, bietet SQL die <Suche oder Zyklus-Klausel> an. Zum einen kann dort spezifiziert werden, ob eine Breitensuche (breadth first) oder eine Tiefensuche (depth first) durchgeführt werden soll. Zum anderen können aufgrund der aktuellen Datenlage Zyklen bei der Auswertung auftreten. Für die Strukturtabelle im Schema der Firma Byce & Co. würde das heißen, dass ein Teil aus sich selbst aufgebaut ist, was in den meisten Fällen sicherlich ein Konstruktionsfehler ist. Rundfahrten bei Zugverbindungen sind dagegen sicherlich normal und gewollt. In beiden Fällen ist es für das Datenbanksystem schwierig zu erkennen, ob eine Schleife einfach nur mehrfach oder unendlich oft durchlaufen wird. In der Klausel kann dann der Programmierer Endkriterien bestimmen.<sup>50</sup>

### Beispiel

Für das Stücklistenproblem der Strukturtabelle wird die rekursive Anfrage so aufgebaut, dass zuerst die Teilenummern der Oberteile, die selbst nicht als Unterteile verwendet werden, ermittelt werden. Für diese Ausgangsmenge werden dann rekursiv alle zugehörigen Unterteile ermittelt. Die Anfrage besteht aus der Hilfsicht „Stueckliste“ und der SELECT-Anfrage an diese Hilfsicht.

<sup>50</sup> (vgl. [ANSI SQL 2003b], [Melton et al. 2002] und [Celko 2005])

```
WITH RECURSIVE
Stueckliste (Teilenr, Position, Menge) AS
  (SELECT OTeil, 0, NULL
  FROM Struktur
  WHERE OTeil NOT IN (SELECT DISTINCT UTeil FROM Struktur)
UNION ALL
  SELECT Struktur.UTeil, Struktur.Position, Struktur.Menge
  FROM Stueckliste, Struktur
  WHERE Stueckliste.Teilenr = Struktur.OTeil )
SELECT * FROM Stueckliste;
```

### 5.5.12 SQL-Tuning-Maßnahmen von SELECT-Anweisungen

Auch auf dieses Thema kann an dieser Stelle nur in Form von Beispielen eingegangen werden. Es werden einige typische SQL-Anweisungen aufgezeigt, die zu langen Laufzeiten führen können. Technisches Tuning, d.h. Tuning, welches sich auf ein spezielles Datenbanksystem bezieht, kann gar nicht behandelt werden<sup>51</sup>. Eine gute Einführung in das Thema Tuning ist auch in Fritze<sup>52</sup> enthalten.

Wichtig beim Tuning<sup>53</sup> ist:

- Die Wahl der Indizes, die zu den häufigsten Anfragen passen
- Die Wahl der treibenden Tabelle, d.h. derjenigen Tabelle, die zuerst gelesen wird (je nach System durch die Reihenfolge der Tabellen in der FROM-Klausel)
- Vermeidung von langsamer Auswertung von Unterabfragen
- Vermeidung von unnötigen Sortiervorgängen
- Verwendung von OR statt UNION
- Vermeidung von Verknüpfungen von zu vielen Tabellen oder Views
- Vermeiden von SELECT-Anweisungen mit DISTINCT, die falsche WHERE-Bedingungen kaschieren
- Untersuchung der Initialisierungsparameter der Datenbank, die falsch eingestellt sein können
- Es werden sehr wenige lesende Zugriffe ausgeführt, mehr Einfüge-, Lösch- und Änderungsanweisungen. Dies muss beim Tuning berücksichtigt werden (wenige Indizes anlegen).

Ein Beispiel für einen falsch gewählten Index ist, dass die indizierte Spalte zu wenig verschiedene Werte hat (Maß: Anzahl unterschiedliche Werte/Anzahl Werte sollte größer als 0.8 sein) oder viele NULL-Werte enthält, die überhaupt nicht indiziert werden. Auch wenn ein Index über mehrere Spalten geht, die Abfrage sich aber nur auf eine Spalte bezieht, wird der Index nicht benutzt.

Generell sind Tabellen-Joins in der Regel schneller als Unterabfragen mit IN oder EXISTS, da bei den Unterabfragen kein Index verwendet wird. Stattdessen findet ein vollständiges Lesen der kompletten Tabelle statt, was natürlich bei großen Tabellen katastrophale Auswirkungen haben kann.

<sup>51</sup> vgl. [Ahrends 2006] und [Loney 2005]

<sup>52</sup> vgl. [Fritze 2003, Kap. 12]

<sup>53</sup> vgl. auch [Gurry 2002]

### Beispiele Tuning

Die Anfrage

```
SELECT * FROM Angestellte a
WHERE EXISTS (SELECT * FROM Abteilungen b
  WHERE a.Abt_Nr = b.Abt_Nr);
```

ist sicher bei genügend großen Tabellen langsamer als die folgende Abfrage:

```
SELECT * FROM Angestellte a, Abteilungen b
WHERE a.Abt_Nr = b.Abt_Nr;
```

Die Anfrage

```
SELECT *
FROM Grosse_Tabelle a, Kleine_Tabelle b
WHERE a.Id = b.ID;
```

ist schneller als

```
SELECT *
FROM Kleine_Tabelle b, Grosse_Tabelle a,
WHERE a.Id = b.ID;
```

da die treibende Tabelle die rechte Tabelle in der FROM-Klausel ist.

Falls ein Index auf Vorname **und** Nachname existiert, kann eine Anfrage der Gestalt

```
SELECT Nachname, Ang_Nr FROM Angestellte
WHERE Nachname = 'Meier';
```

nicht unterstützt werden.

Auch ein Index auf die Spalte „Nachname“ hilft wenig, falls die Anfrage von der Gestalt

```
SELECT Nachname, Ang_Nr FROM Angestellte
WHERE Nachname = UPPER('meier')
OR Nachname LIKE 'M%';
```

ist. Funktionen (UPPER) oder LIKE-Prädikate in der WHERE-Klausel können nämlich nicht auf einen Index zugreifen.

Die Anweisung

```
SELECT DISTINCT a.Ang_Nr, a.Nachname, a.Abt_Nr, b.Name
FROM Angestellte a, Abteilungen b ;
```

sollte durch

```
SELECT a.Ang_Nr, a.Nachname, a.Abt_Nr, b.Name
FROM Angestellte a, Abteilungen b
WHERE a.Abt_Nr = b.Abt_Nr;
```

ersetzt werden.

## 5.5.13 Oracle-Besonderheiten in der SQL-DQL-Syntax

### 5.5.13.1 Rekursivität in der SELECT-Anweisung

Im SQL-Standard ist seit 1999 für rekursive Anfragen eine WITH RECURSIVE-Klausel vorgesehen, wie wir im vorigen Kapitel gesehen haben. Dieser Ansatz wird in ähnlicher Form u.a. von DB/2 verwendet, nicht aber von MySQL und Oracle. Oracle hat die WITH-Klausel nur ohne Rekursion umgesetzt und benutzt weiterhin die schon seit langem implementierte START WITH CONNECT-BY-Klausel, die in diesem Kapitel vorgestellt wird.

Wir greifen wieder das Stücklistenproblem der Strukturtabelle in unserer Beispieldatenbank Byce & Co. auf, in der alle Bestandteile eines Artikels oder einer Baugruppe enthalten sind. Rekursive Beziehungen kommen in der Praxis häufig vor.

Der Stücklistenbaum hat die Stücklistenauflösung des Artikels mit der TNr= 60, des SCOTT ATACAMA TOUR-Fahrrads, zum Inhalt. Möchte man ohne rekursive Auswertungsformen zum Beispiel herausfinden, welche Bestandteile der Artikel mit der TNr = 60 hat, muss man folgende Abfrage starten:

```
SELECT t.TNr, s.UTeil
FROM Teile t, Struktur s
WHERE t.TNr = s.0Teil AND t.TNr = 60;
```

**Ergebnis:**

TNr	UTeil
60	2
60	41
60	62

Geht man noch eine Stufe weiter, löst man also die in den Baugruppen enthaltenen Teile noch mal auf, braucht man folgende Abfrage:

```
SELECT t.TNr, s1.UTeil, s2.UTeil
FROM Teile t, Struktur s1, Struktur s2
WHERE t.TNr = s1.0Teil AND s1.UTeil = s2.0Teil
AND t.TNr = 60;
```

**Ergebnis:**

TNr	UTeil	UTeil
60	2	3
60	2	4
60	41	42
60	41	43
60	41	44
60	41	45
60	62	40
60	62	49

Die Einbeziehung einer weiteren Stufe liefert alle dreistufigen Bestandteile des Stücklistenbaums:

```
SELECT t.TNr, s1.UTeil, s2.UTeil, s3.UTeil
FROM Teile t, Struktur s1, Struktur s2, Struktur s3
WHERE t.TNr = s1.0Teil AND s1.UTeil = s2.0Teil
AND s2.UTeil = s3.0Teil
AND t.TNr = 60;
```

**Ergebnis:**

TNr	UTeil	UTeil	UTeil
60	62	40	3
60	62	40	34

Es ist klar, dass die Abfragen so beliebig kompliziert werden, außerdem ist die Stücklistentiefe immer noch beschränkt. In unserem Beispiel wurden nur drei Stufen einbezogen und man bekommt in jeder Abfrage nur eine Stufe, nicht alle Stufen zugleich. Will man alle Stufen anzeigen, muss man mit OUTER JOIN-Anfragen arbeiten.

Oracle bietet mit der CONNECT-BY-Klausel ein Verfahren an, mit dem man solche hierarchischen Beziehungen unabhängig von der Tiefe des Baums durchlaufen kann.

### Beispiel

```
SELECT OTeil, UTeil
FROM Struktur
START WITH OTeil = 60
CONNECT BY PRIOR UTeil = OTeil;
```

<CONNECT BY Klausel> ::= [START WITH Spaltenname = Konstante]  
CONNECT BY <CONNECT Bedingung>

- START WITH ist optional und legt den oder die Datensätze fest, mit denen die Abfrage beginnen soll.
- Die CONNECT-Bedingung bestimmt die Verbindung zwischen dem übergeordneten und dem untergeordneten Datensatz.
- PRIOR legt den Vaterknoten in der CONNECT-BY-Bedingung fest.
- LEVEL ist eine Pseudospalte, in der die Tiefe der Auflösung festgehalten wird.

### Beispiele

- Welche Teile sind im Artikel TNr 60 in welcher Stücklistentiefe enthalten?

```
SELECT OTeil, UTeil, LEVEL
FROM Struktur
START WITH OTeil = 60
CONNECT BY PRIOR UTeil = OTeil;
```

Ergebnis:

OTeil	UTeil	LEVEL
60	2	1
2	3	2
2	4	2
60	41	1
41	42	2
41	43	2
41	44	2
41	45	2
60	62	1
62	40	2
40	3	3
40	34	3
62	49	2

■ Wie ist die maximale Stücklistentiefe des Artikels 60?

```
SELECT MAX (LEVEL)
  FROM Struktur
 WHERE OTeil = 60
 CONNECT BY PRIOR UTeil = OTeil;
```

Ergebnis:

MAX(LEVEL)
3

### 5.5.13.2 Weitere Besonderheiten

Die SELECT-Anweisung bietet eine FOR-UPDATE-OF-Option, mit der Daten während des Lesens für andere Zugriffe gesperrt werden (vgl. Kapitel 8). Mit der DECODE-Funktion lassen sich Werte in Abhängigkeit von anderen Werten setzen. DECODE ersetzt damit mehrere IF THEN-Anweisungen, die in PL/SQL eingeführt werden. Anstelle der DECODE-Funktion kann auch die CASE-Funktion genutzt werden.

Oracle setzte das SQL-JOIN-Konzept erst recht spät, dafür aber uneingeschränkt um. In der Vor-JOIN-Zeit gab es aber auch die Möglichkeit, OUTER-JOINS zu formulieren. Ein „+“-Zeichen rechts oder links positioniert in der WHERE-Bedingung, führt dann zu einem rechten oder linken OUTER-JOIN. Die WITH-Klausel ist ein sehr junges Konzept bei Oracle und dementsprechend noch nicht vollständig umgesetzt. Rekursive Hilfsichten, die dort Inline Views genannt werden, sind nicht erlaubt. Als Anfragen in den Hilfsichten sind anders als bei SQL nur beliebig komplexe SELECT-Ausdrücke ohne eigene WITH-Klausel zugelassen.

### 5.5.14 MySQL-Besonderheiten in der SQL-DQL-Syntax

Die IF- und die CASE-Anweisung entsprechen den obigen Oracle-Anweisungen DECODE und CASE und liefern Werte in Abhängigkeit von anderen Werten. Die IFNULL-Funktion ist eine verkürzte Form von IF und liefert einen Ausdruck anstelle eines anderen Ausdrucks zurück, wenn dieser einen NULL-Wert hat. Mit der LIMIT-Klausel kann die Anzahl der Datensätze in einer Abfrage (und sogar einer UPDATE-Anweisung) beschränkt werden:

```
SELECT * FROM Angestellte LIMIT 5;
```

In MySQL ist eine umfangreiche Volltextsuche enthalten, allerdings nur für MyISAM-Tabellen.

Die FOR-UPDATE-Option sperrt ähnlich wie unter Oracle Daten während des Lesens für andere Zugriffe.

MySQL bietet als JOIN-Operatoren den CROSS JOIN sowie die beiden Operatoren OUTER und NATURAL JOIN mit den Optionen LEFT und RIGHT in der SQL-konformen Syntax und Funktionalität. Abweichend ist hier der INNER JOIN, oder auch nur kurz JOIN geschrieben, ein Synonym für CROSS JOIN. Der [NATURAL] FULL OUTER JOIN fehlt. Zusätzlich gibt es den STRAIGHT JOIN, der identisch mit dem CROSS JOIN ist, mit der Ausnahme, dass die linke Tabelle immer zuerst gelesen wird. Dieser Operator ist für den Fall gedacht, dass der DB-Optimierer die falsche Strategie wählt. Die SELECT-Anfrage ist ohne eine WITH-Klausel implementiert. Und eine Unteranfrage ist nur ein Anfragemengenausdruck wie aus Abschnitt 5.5.9, der auch nicht in der FROM-Klausel verwendet werden kann.

## 5.6 Die Datenadministrationssprache (DAL, Data Administration Language)

Neben den Aufgaben des Datenbankentwicklers, der sich hauptsächlich mit dem Anlegen, Ändern und Löschen von Tabellen und den Daten, die in den Tabellen enthalten sind, beschäftigt, kann SQL auch noch für das Aufgabengebiet des Datenbankadministrators eingesetzt werden. Diesen Bereich von SQL nennt man DAL. Leider ist dieser wenig standardisiert, dafür aber umso umfangreicher. Wir beschränken uns auf einige Grundlagen, die in den gängigen Datenbanksystemen gleich oder ähnlich sind.

### 5.6.1 Anlegen und Löschen von Benutzerrechten

#### 5.6.1.1 Die GRANT-Anweisung

Mit dieser Anweisung lassen sich sehr differenziert auf einzelnen Tabellen Benutzerrechte vergeben.

```
<GRANT Anweisung> ::= GRANT <Privileg> [ ON { Tabellenname | Sichtname } ]
                           TO <Person>;
<Person> ::= / PUBLIC | Benutzername /
<Privileg> ::= / ALL | SELECT | DELETE | INSERT
                           | UPDATE {Spaltenname [, Spaltenname ]...} /
```

**Beispiele**

```
GRANT ALL ON Teile TO PUBLIC;
GRANT SELECT ON Teile TO Hugo;
GRANT UPDATE (Typ, Bezeichnung) ON Teile TO Fritz;
```

Wer kann Benutzerrechte vergeben?

- Der Datenbankentwickler kann auf selbst eingerichteten Tabellen Rechte vergeben.
- Der Datenbankadministrator kann auf allen Tabellen Rechte vergeben.
- Der Datenbankadministrator richtet auch die Benutzer selbst ein.

SQL2003 bietet mit CREATE ROLE die Möglichkeit, eine Rolle zu erzeugen, der Benutzerrechte zugewiesen werden können. Dieser Rolle können dann über GRANT-Anweisungen einzelne Benutzer zugeordnet werden. Dieses Rollenkonzept vereinfacht die Rechteverwaltung gerade bei großen Nutzerzahlen und einem detaillierten Rechtekonzept gravierend.

**5.6.1.2 Die REVOKE-Anweisung**

Mit der REVOKE-Anweisung werden Datenbankprivilegien wieder aus der Datenbank entfernt:

```
<REVOKE Anweisung> ::= REVOKE <Privileg> ON [ Tabellename | Sichtname ]
FROM <Person>;
```

**Beispiel**

```
REVOKE ALL ON Teile FROM Hugo;
```

**5.6.1.3 Andere DAL-Anweisungen**

Die COMMIT-Anweisung macht Datenbankzugriffe erst dauerhaft wirksam, die ROLLBACK-Anweisung spielt temporäre Anweisungen wieder auf einen konsistenten Zustand zurück. Beide Anweisungen gehören zwar zur DAL-Sprache, werden aber erst in Kapitel 8, welches Transaktionen behandelt, ausführlich dargestellt. Die übrigen Bestandteile der DAL-Sprache sind sehr stark herstellerabhängig und es sei daher auf die Originalliteratur verwiesen.

**5.6.1.4 Oracle- und MySQL-Besonderheiten der DAL-Sprache**

Während bei Oracle einzelne Datenbankbenutzer über CREATE USER erzeugt werden, die dann gemeinsam zu einer Datenbank (CREATE DATABASE ...) gehören, fehlt bei MySQL hier eine Ebene. Einem Datenbankbenutzer unter Oracle wird automatisch ein Datenbankschema zugeordnet, d.h., die Tabellen des Benutzers sind über Benutzernamen/Tabellennamen auch von anderen Benutzern der Datenbank ansprechbar. Der Besitzer einer Tabelle kann auf den von ihm angelegten Tabellen Rechte an andere Benutzer der Datenbank vergeben.

Das SQL-Rollenkonzept wurde bei Oracle erfolgreich umgesetzt. Es wird von den Anwendern gut angenommen. Leider fehlt noch eine Implementierung unter MySQL.

Die Zuweisung von Rechten ist bei Oracle über den Standard hinaus sehr viel ausdifferenzierter, da es neben den oben genannten Objektprivilegien auch Systemprivilegien (GRANT CREATE TABLE, GRANT ALTER TABLE u.a.) gibt.

Zwar können mit CREATE DATABASE unter MySQL auch neue Datenbanken erzeugt und mit dem GRANT-Befehl neue Benutzer angelegt werden, aber anders als bei Oracle wird für einen Benutzer nicht automatisch ein eigenes Datenbankschema angelegt. Bei beiden gibt es keinen CREATE SCHEMA-Befehl wie beim SQL-Standard.

Unter Oracle kann man mit

```
ALTER USER Benutzername IDENTIFIED BY Passwort;
```

einem Benutzer ein Passwort zuweisen, unter MySQL passiert dies mit der Anweisung

```
SET PASSWORD = PASSWORD('Passwort');
```

Unter MySQL kann man einen Benutzer auch mit RENAME umbenennen.

**5.7 SQL und die Objekte der Relationalen Algebra**

Wie in Kapitel 4 gesehen, gibt es relationale Objekte, relationale Operatoren und relationale Integritätsregeln, die die relationale Algebra ausmachen. SQL ist eine Umsetzung dieser relationalen Algebra in die Praxis. Dieses Kapitel befasst sich damit, inwiefern die Anforderungen der relationalen Algebra in SQL umgesetzt sind.

**Tabelle 5.33**

Relationale Objekte	
Objekt in der Relationalen Algebra	SQL-Darstellung
Relation	Tabelle, wird mit der CREATE TABLE-Anweisung erzeugt
Domäne	CREATE TABLE-Anweisung mit Datentypen und CONSTRAINTS SQL: CREATE DOMAIN
Attribut	Spalte einer Tabelle
Tupel	Zeile einer Tabelle/Datensatz
Primärschlüssel	CONSTRAINT PRIMARY KEY
Zweitschlüssel	CONSTRAINT UNIQUE
Fremdschlüssel	CONSTRAINT FOREIGN KEY

Tabelle 5.34	
Relationale Operationen	
<b>Operation</b>	<b>Umsetzung in der SQL-SELECT-Anweisung</b>
Projektion	Spaltenliste der SELECT-Klausel SELECT Spalte_3, Spalte_2 FROM Tabelle;
Selektion	WHERE-Bedingung SELECT * FROM Tabelle WHERE A > B;
Kartesisches Produkt	FROM-Klausel ohne WHERE-Klausel SELECT Tabelle_1.*, Tabelle_2.* FROM Tabelle_1, Tabelle_2;
Vereinigung	UNION (für zwei Tabellen mit gleicher Spaltendefinition) SELECT * FROM Tabelle_1 UNION SELECT * FROM Tabelle_2;
Differenz	EXCEPT oder MINUS SELECT * FROM Tabelle_1 EXCEPT SELECT * FROM Tabelle_2;
Durchschnitt	INTERSECT SELECT * FROM Tabelle_1 INTERSECT SELECT * FROM Tabelle_2;
NATURAL JOIN	NATURAL JOIN-Operator oder: Für die Tabellen der FROM-Klausel werden in der WHERE-Bedingung alle Spalten, die gleich heißen, miteinander auf Gleichheit verglichen und mit AND verknüpft und doppelte Spalten in der SELECT-Liste herausprojiziert. SELECT Artikel.TNr, AuftragsNr, Menge FROM Artikel, Auftragspositionen WHERE Artikel.TNR = Auftragspositionen.TNR;
THETA-JOIN	JOIN-Operator mit beliebiger Bedingung oder: Für die Tabellen der FROM-Klausel kann eine beliebige WHERE-Bedingung formuliert werden. SELECT k.Kun_Nr, Nachname, Bestelldatum FROM Auftraege a, Kunden k WHERE k.Kun_Nr = a.Kun_Nr AND Bestelldatum IS NOT NULL;
OUTER-JOIN	LEFT, RIGHT, FULL OUTER-JOIN-Operator SELECT k.Kun_Nr, Nachname, Bestelldatum FROM Auftraege a RIGHT OUTER JOIN Kunden k ON k.Kun_Nr = a.Kun_Nr;

**Fazit:**

- SQL2003, Oracle und MySQL entsprechen bei den relationalen Objekten den Anforderungen der relationalen Algebra. Das Domänenkonzept ist bei Oracle und MySQL jedoch nur unvollständig umgesetzt.
- Relationale Integritätsregeln sind durch Integritätsbedingungen (vgl. Abschnitt 5.3.2) und Datenbanktrigger (vgl. Abschnitt 7.3) vollständig abgedeckt.
- Die Anforderungen, die aus den Operatoren der relationalen Algebra resultieren, sind in SQL2003, Oracle und MySQL mit dem SELECT-Befehl umgesetzt.
- Der SELECT-Befehl bietet sogar noch weitergehende Möglichkeiten: GROUP BY, HAVING und ORDER BY gehen über die Anforderungen der relationalen Algebra hinaus, genauso wie die Aggregatfunktionen COUNT, SUM, MIN, MAX und AVG.
- Ein grundlegender Unterschied bleibt zwischen der relationalen Algebra und SQL. SQL lässt Duplikate zu und die relationale Algebra nicht. Durch die DISTINCT-Option in der SELECT-Klausel und beim UNION kann auch unter SQL duplizatfrei gearbeitet werden.

**5.8 Data-Dictionaries**

Eine Forderung von E.F. Codd war (vgl. Abschnitt 1.1.5), alle Informationen über Metadaten mit den Mitteln einer relationalen Datenbank, also wieder mit Relationen zu beschreiben. Diese Metadaten nennt man auch das DATA DICTIONARY. Natürlich sind diese Relationen, die beim Installieren einer Datenbank abgelegt sind, herstellerspezifisch und sehr umfangreich. Es werden kurz das Oracle- und das MySQL-Dictionary vorgestellt.

**5.8.1 Oracle-Dictionary**

Einige Oracle-DICTIONARY-Sichten, die nur zur Einsicht in die zugrunde liegenden Verwaltungstabellen des Datenbankmanagementsystems dienen, sollen hier aufgeführt werden.

**Tabelle 5.35**

Die drei Typen von DICTIONARY-Sichten	
Beginn	Inhalt
USER_...	enthalten alle Objekte, die dem eingeloggten Benutzer gehören.
ALL_...	enthalten alle Objekte, auf die der eingeloggte Benutzer Zugriffsberechtigung hat.
DBA_...	enthalten alle Objekte, die es in der Datenbank gibt.

In der nachfolgenden Tabelle werden einige „USER\_-“-Sichten aufgeführt. Die entsprechenden Sichten gibt es auch mit den Präfixen „DBA\_-“ und „ALL\_-“. Das Data-Dictionary ist nur als Sicht für jeden Benutzer sichtbar, um die Daten vor unberechtigtem Zugriff zu schützen. Änderungen sind nur durch die Ausführung von CREATE, ALTER und DROP-Befehlen für Datenbankobjekte möglich. Wenn Sie die Spaltendefinitionen der Sichten benötigen, erhalten Sie diese in SQL\*PLUS mit dem Befehl „DESCRIBE Sichtname“. Alle Sichtnamen, die zur Meta-Datenbank gehören und auf die Sie Zugriff haben, lesen Sie mit der SQL-Anweisung:

```
SELECT Table_Name FROM Dictionary
WHERE Table_Name LIKE 'USER_%' ORDER BY Table_Name;
```

Tabelle 5.36

Die Views des Oracle-Data-Dictionary	
Informationen über	sind enthalten in
DATA-DICTIONARY	DICTIONARY, DICT_COLUMNS
Alle Objekte	USER_OBJECTS, USER_OBJECT_SIZE, USER_OBJECT_TABLES
Tabellen	USER_TABLES, USER_ALL_TABLES, USER_CATALOG, USER_TAB_COLUMNS
Sichten	USER_VIEWS
Integritätsbedingungen	USER_CONSTRAINTS, USER_CONS_COLUMNS
Indizes	USER_INDEXES, USER_IND_COLUMNS
Prozeduren	USER_SOURCE, ALL_ARGUMENTS, USER_ERRORS USER_LIBRARIES, USER_DEPENDENCIES
Sequenzen	USER_SEQUENCES
Trigger	USER_TRIGGERS, USER_TRIGGER_COLS, USER_DEPENDENCIES, USER_OBJECT_SIZE
Benutzerrechte	USER_TAB_PRIVS
Benutzer	USER_USERS, USER_PASSWORD_LIMITS, USER_SYS_PRIVS, USER_JOBS, USER_FREE_SPACE

## 5.8.2 MySQL: INFORMATION\_SCHEMA

Das MySQL-Data-Dictionary gehört zur Datenbank INFORMATION\_SCHEMA, die in jeder Installation enthalten ist.

Tabelle 5.37

Die wichtigsten Objekte	
Informationen über	sind enthalten in
Datenbanken	SCHEMATA
Tabellen	TABLES
Spalten der Tabellen	COLUMNS
Indizes	STATISTICS
Globale Benutzerrechte	USER_PRIVILEGES
Rechte auf Datenbankschemata	SCHEMA_PRIVILEGES
Rechte auf Tabellen	TABLE_PRIVILEGES
Rechte auf Spalten	COLUMN_PRIVILEGES
Verfügbare Zeichensätze	CHARACTER_SETS
Tabellenbedingungen	TABLE_CONSTRAINTS
Schlüsselbedingungen	KEY_COLUMN_USAGE
Gespeicherte Prozeduren	ROUTINES
Sichten	VIEWS
Datenbanktrigger	TRIGGERS
Speichermaschine	ENGINES
Tabellenpartitionen	PARTITIONS
Ereignisse	EVENTS

Ähnlich wie unter Oracle lassen sich mit der Anweisung „SHOW Name“ die Spalten der einzelnen Data-Dictionary-Tabellen abfragen. Der Inhalt des Data-Dictionary lässt sich dann wie gewohnt über eine SELECT-Anweisung bestimmen.

Beispiel

```
SELECT Table_Name, Table_Type
FROM Information_Schema.Tables
WHERE Table_Schema = 'test'
ORDER BY Table_Name;
```

## ZUSAMMENFASSUNG

In diesem umfangreichen Kapitel des Buchs wurden ausführlich die wichtigsten SQL-Anweisungen aus dem neuen Standard SQL2003 behandelt. Trotzdem reicht der Platz bei weitem nicht aus, eine vollständige Darstellung der Sprache zu erreichen, was bei der Größe des aktuellen Standards nicht weiter verwundert. So gibt es im deutschsprachigen Raum auch noch kein Buch, das wie [Date 1998] den Standard SQL2003 vollständig wiedergibt. Außergewöhnlich ausführlich wurde neben den DDL-Anweisungen CREATE und DROP und den DML-Anweisungen INSERT, UPDATE und DELETE die umfangreiche SELECT-Anweisung auch mit rekursiver WITH-Klausel vorgestellt. Eine Besonderheit ist sicher die detaillierte Behandlung der Konzepte der Integritätsicherung und auch der Vergleich der beiden SQL-Dialekte MySQL und Oracle. Es zeigen sich viele Gemeinsamkeiten, aber auch Unterschiede in der Syntax der beiden Dialekte, angefangen bei den unterschiedlichen Datentypen, bis hin zu den einzelnen SQL-Anweisungen und dem Data Dictionary.

## Weiterführende Literatur

Wer an genaueren Informationen interessiert ist, kann an dieser Stelle, was die SQL-Dialekte von Oracle und MySQL angeht, am besten auf die Herstellerliteratur [Oracle 2005] und [MySQL 2006] zurückgreifen. Als umfangreiche Quellen zum Standard-SQL sind [Groff et al. 2002] sowie [Melton et al. 2002] zu nennen. Tuning-Aspekte werden z.B. in [Fritze 2002] und speziell für Oracle in [Ahrends 2006] und [Gurry 2002] behandelt.

## Übungsaufgaben

### Aufgaben zum Abschnitt 5.3 (Data Definition Language):

- 1** Erstellen Sie ein SQL-Skript, in dem alle CREATE TABLE-Anweisungen Ihres Datenmodells Rollo enthalten sind!  
Erstellen Sie ein Skript, das jeweils
    - a. unter Oracle lauffähig ist,
    - b. unter MySQL lauffähig ist.
  - 2** Erstellen Sie ein SQL-Skript, das alle SQL-Anweisungen für Ihre Rollo-Datenbank enthält, die zur Definition von Indizes erforderlich sind, die nicht Primärschlüsseln und Zweitenschlüsseln entsprechen!  
Erstellen Sie ein Skript, das jeweils
    - a. unter Oracle lauffähig ist,
    - b. unter MySQL lauffähig ist.
  - 3** Erzeugen Sie einen Nummerngenerator, der für die Personen, die Spiele und die Spieler in Ihrer Rollo-Datenbank je eine fortlaufende Nummer bereitstellt!  
Erstellen Sie ein Skript, das jeweils
    - a. unter Oracle lauffähig ist,
    - b. unter MySQL lauffähig ist (Ersatz für Sequence: Auto\_Increment-Spalte).
  - 4** Erstellen Sie ein SQL-Skript, das alle Tabellen aus Ihrer Datenbank Rollo wieder entfernt. Wo finden Sie die Information, welche Tabellen es überhaupt in dem Schema bzw. in der Datenbank gibt? Vergewissern Sie sich vor der Ausführung der Löschanweisungen, dass Ihnen die notwendigen Skripte zur Wiederherstellung der Tabellen zur Verfügung stehen.  
Erstellen Sie ein Skript, das jeweils
    - a. unter Oracle lauffähig ist,
    - b. unter MySQL lauffähig ist.
- Für das Testen der nachfolgenden Integritätsbedingungen benötigen Sie Kenntnisse aus der SQL-DML, die im Abschnitt 5.4 erläutert wird.
- 5** Erstellen Sie ein SQL-Skript, das die Primär-, Eindeutigkeits- und Fremdschlüssel, die Sie auf der Rollo-Datenbank geplant haben, enthält! Benutzen Sie bitte die ALTER TABLE-Anweisung und den unmittelbaren Prüfungszeitpunkt. Testen Sie Ihre Schlüsselbedingungen zumindest für jeweils eine der Schüsselarten.  
Erstellen Sie ein Skript, das jeweils
    - a. unter Oracle lauffähig ist,
    - b. unter MySQL lauffähig ist.



 **6** Welche Vorteile hat bei der Erstellung der Fremdschlüsselbedingungen die Verwendung des ALTER TABLE-Befehls gegenüber dem CREATE TABLE-Befehl?

**7** Programmieren Sie ALTER TABLE-Anweisungen für die Byce & Co.-Datenbank unter Oracle, um folgende Integritätsbedingungen zu implementieren. Schreiben Sie Testfälle für die Bedingungen, die zeigen, dass richtige Daten weiterhin gespeichert und fehlerhafte Daten abgewiesen werden. Beachten Sie insbesondere die Prüfungszeitpunkte.

- Bei den Angestellten sind als Geschlechtskennzeichen nur die beiden Werte „w“ und „m“ zugelassen, wobei auch Großbuchstaben möglich sind. Die Prüfung soll unmittelbar erfolgen.
- Bei den Kunden soll in der Spalte „Geschlecht“ zudem ein „j“ zugelassen sein, was als Abkürzung für „juristische Person“ bei Firmen eingetragen wird. Auch hier soll eine unmittelbare Prüfung stattfinden.
- Als Auftragstyp sind nur die beiden Werte „Angebot“ und „Auftrag“ erlaubt. Groß- und Kleinschreibung soll keinen Unterschied machen bei der Eingabe. Das Transaktionsende ist der Prüfungszeitpunkt.
- Bei den Gehaltsklassen muss die obere Grenze größer sein als die untere Grenze. Die Prüfung soll unmittelbar erfolgen.
- Bei der Auftragsabwicklung muss die Reihenfolge der Termine beachtet werden. Zuerst kommt die Bestellung, dann wird geliefert und schließlich die Rechnung geschrieben. Aktionen können aber auch am gleichen Tag erfolgen. Die Prüfung findet zum Transaktionsende statt.
- Da Teile entweder eingekauft oder selbst gefertigt werden, sind entweder der Einkaufspreis und die Lieferzeit oder die Herstellkosten und die Herstelldauer zu erfassen. Es können zwar sowohl die Herstell- als auch die Lieferinformationen fehlen, es dürfen aber nicht Herstell- und Lieferinformationen zusammen angegeben werden. Wenn eine der Lieferinformationen angegeben wird, dann muss auch die andere Lieferinformation gefüllt sein. Das Gleiche gilt für die Herstellinformationen. Für diese Prüfungen sind eine leere Spalte und der numerische Nullwert gleich zu behandeln. Damit die Bedingung übersichtlich bleibt, verwenden Sie bitte die NVL-Funktion. Prüfen Sie diese Bedingung zum Transaktionsende. (Diese Integritätsbedingung ist sicherlich schon etwas anspruchsvoller, aber durchaus praxisnah.)

 **8** Programmieren Sie ALTER TABLE-Anweisungen für die Rollo-Datenbank unter Oracle, um folgende Integritätsbedingungen zu implementieren. Schreiben Sie Testfälle, die zeigen, dass richtige Daten weiterhin gespeichert und fehlerhafte Daten abgewiesen werden. Beachten Sie insbesondere die Prüfungszeitpunkte.

- Es darf kein Land in einem Spiel gegen sich selbst spielen. Die Prüfung soll unmittelbar erfolgen.
- Die Anzahl reservierter Karten darf maximal so groß sein wie die Anzahl gewünschter Karten. Prüfen Sie diese Bedingung zum Transaktionsende.

c. Aus Sicherheitsgründen muss für jede Person eine Ausweisnummer angegeben werden. Die Prüfung soll unmittelbar erfolgen.

d. Eine Kartenbestellung kann entweder mittels Kreditkarte bezahlt werden oder durch eine Abbuchung von einem Konto. Daher muss entweder eine Kreditkartennummer oder eine Bankverbindung bestehend aus Bankleitzahl und Kontonummer angegeben werden. Prüfen Sie diese Bedingung zum Transaktionsende.

**9** Es soll die Angabe des Spielergebnisses in der Rollo-Datenbank auf syntaktische Korrektheit geprüft werden. Programmieren Sie die dazu erforderlichen ALTER TABLE-Anweisungen unter Oracle. Schreiben Sie wieder die erforderlichen Testfälle. Geprüft wird unmittelbar. Es wird die Annahme getroffen, dass mehr als zweistellige Zahlen für Torschüsse einer Nation ausgeschlossen werden können.

- Für das Ergebnis eines Spiels ist als Trennzeichen der Doppelpunkt zulassen, also z.B. 3:4 oder 10:0.
- Das Trennzeichen Doppelpunkt darf nicht an erster Stelle stehen.
- Steht das Trennzeichen Doppelpunkt an zweiter Stelle, dann darf das Ergebnis 3 oder 4 Zeichen haben. Steht es an dritter Stelle, dann darf das Ergebnis 4 oder 5 Stellen haben.
- Vor und nach dem Trennzeichen dürfen nur ein bis zweistellige Zahlen stehen.

Hinweise: Hilfreich sind hier die SQL-String-Funktionen wie INSTR, SUBSTR, LENGTH ... oder auch der LIKE-Operator.

**10** Schreiben Sie eine CREATE TABLE-Anweisung für eine Tabelle „Test“ mit den Spalten A, B und C mit folgenden Eigenschaften:

- Alle Spalten sollen vom Datentyp NUMBER sein.
- A ist der Primärschlüssel.
- Stellen Sie über CHECK-Constraints sicher, dass genau eines der Attribute „B“ und „C“ NULL ist; das heißt, wenn B nicht NULL ist, muss C NULL sein und umgekehrt.

**11** Warum lassen sich die Aufgaben 7 bis 10 nicht mit MySQL lösen?

**12** Betrachten Sie die angegebenen SQL-Anweisungen der Relation „Hierarchie“:

```
DROP TABLE Hierarchie;
CREATE TABLE Hierarchie (
    Angestellter VARCHAR(20) NOT NULL,
    Vorgesetzter VARCHAR(20) NOT NULL,
    PRIMARY KEY (Angestellter),
    FOREIGN KEY (Vorgesetzter) REFERENCES Hierarchie ON DELETE CASCADE);
DELETE FROM Hierarchie;
```

Nach einigen INSERT-Anweisungen hat die Tabelle folgenden Inhalt:

Angestellter	Vorgesetzter
Schulz	Schulz
Meier	Schulz
Müller	Meier
Schmidt	Schulz
Tulpe	Müller
Real	Tulpe
Taris	Müller

- a. Erklären Sie allgemein die Wirkung des Löschens eines Datensatzes aus der erzeugten Tabelle!
- b. Wie wirkt sich das Löschen des Datensatzes (Meier, Schulz) aus? Welche Datensätze bleiben übrig?

#### Aufgaben zum Abschnitt 5.4 (Data Manipulation Language):

Weitere Aufgaben, mit denen Sie DML-Anweisungen üben können, finden Sie bei den Integritätsprüfungsaufgaben unter „Aufgaben zum Abschnitt 5.3“.

- 13** Erstellen Sie ein Skript zur Lösung der nachfolgenden Aufgaben, das jeweils unter Oracle und MySQL lauffähig ist.
  - a. Erstellen Sie ein SQL-Skript, das alle Daten aus der Rollo-Datenbank löscht!
  - b. Schreiben Sie ein SQL-Skript, das in jede Tabelle Ihrer Datenbank mindestens einen Datensatz einfügt!
  - c. Schreiben Sie ein SQL-Skript, das alle SQL-Skripte, die zum Anlegen Ihrer Datenbank Rollo dienen, nacheinander aufruft und den Ablauf mitprotokolliert.
- 14** Lösen Sie folgende Aufgaben für Ihr Rollo-Modell unter Oracle und MySQL.
  - a. Fügen Sie in die Tabelle „Nationen“ eine Spalte „Anzahl Spieler“ ein! Belegen Sie diese Spalte mit der tatsächlichen Anzahl der Spieler aus der Tabelle „Spieler“ mithilfe einer UPDATE-Anweisung!
  - b. Alle Spieler und Personen des Rollo-Systems bekommen eine Email-Adresse. Belegen Sie mit einer UPDATE-Anweisung diese Email-Adresse für jeden Spieler und jede Person mit der Standardadresse Vorname.Nachname@rollo.de!
- 15** In die Tabelle „Nationen“ aus Ihrer Rollo-Datenbank sind versehentlich einige Nationen doppelt eingefügt worden. Schreiben Sie eine SQL-Anweisung, die die doppelten Datensätze (Spalte: Nationname ist doppelt) wieder löscht.



- 16** In das Rollo-System sollen alle vorhandenen Karten eingepflegt werden. Schreiben Sie ein INSERT STATEMENT, das mittels einer SELECT-Anweisung für jede Preiskategorie, die Ihr System hat (mindestens zwei Preiskategorien), zehn Karten einfügt, die noch frei sind, d.h. weder BESTELL\_ID noch PERSONEN\_ID haben.

Hinweis: Benutzen Sie die Tabelle „Klein“!

```
CREATE TABLE Klein ( k Integer);
INSERT INTO Klein (k) VALUES (0);
INSERT INTO Klein (k) VALUES (1);
INSERT INTO Klein (k) VALUES (2);
INSERT INTO Klein (k) VALUES (3);
INSERT INTO Klein (k) VALUES (4);
INSERT INTO Klein (k) VALUES (5);
INSERT INTO Klein (k) VALUES (6);
INSERT INTO Klein (k) VALUES (7);
INSERT INTO Klein (k) VALUES (8);
INSERT INTO Klein (k) VALUES (9);
```

#### Aufgaben zum Abschnitt 5.5 (Data Query Language):

- 17** Welche Tabellen, Views und Indizes sind unter Ihrer Kennung/in Ihrer Datenbank angelegt? Welche Spalten haben die unter Ihrer Kennung angelegten Tabellen? Formulieren Sie Anfragen in Oracle-SQL und MySQL, die diese Fragen beantworten.
- 18** Formulieren Sie Anfragen an die Byce & Co.-Datenbank unter Oracle und MySQL, die diese Fragen beantworten:
  - a. Bestimmen Sie alle Angestellten aus der Abteilung mit der Abteilungsnummer 2, die mehr als 5000 € verdienen.
  - b. Bestimmen Sie alle Mountainbikes, die weniger als 2000 € kosten oder einen Jahresumsatz von weniger als 100 Stück hatten.
  - c. Bestimmen Sie alle Artikeltypen!
  - d. Welchen Namen hat die Abteilung mit der Abteilungsnummer 2?
  - e. Welche Artikel (Typ = „Artikel“) liegen im Hauptlager und haben einen Bestand > 0?
  - f. Welche Materialien werden von Lieferanten aus Dortmund geliefert?
  - g. Welche Angestellten stammen aus Dortmund und gehören zur Abteilung Vertrieb?
  - h. Welche Angestellten verdienen zwischen 50000 € und 80000 € im Jahr?
  - i. Welche Angestellten haben einen Nachnamen, der mit W beginnt und kein e enthält?
  - j. Welche Angestellten haben einen Vornamen, der als zweiten Buchstaben ein u hat?
  - k. Welche Angestellten betreuen Aufträge der Kunden aus Gummersbach?
  - l. Welche Angestellten sind zwischen dem 1.1.90 und dem 1.1.97 eingestellt worden?

- m. Finden Sie die Abteilungsnummern der Abteilungen in Dortmund, in denen es Angestellte gibt, die weniger als 2000 € im Monat verdienen.
- n. Finden Sie die Namen der Angestellten, die den gleichen Beruf und das gleiche Gehalt wie der Angestellte Hugo Schmidt haben.
- o. Welche Teile, die an Kunden verkauft wurden, lagern am gleichen Ort, an dem die Kunden wohnen?
- p. Welcher Lieferant liefert alle Materialien?
- q. Welcher Kunde kauft alle Artikel?
- 19** Formulieren Sie Anfragen an die Byce & Co.-Datenbank unter Oracle und MySQL, die diese Fragen beantworten:
- Erzeugen Sie eine nach Gehalt aufsteigend sortierte Liste aller Angestellten mit Nachnamen, Vornamen, Gehalt und der Gehaltsklasse (Tabelle: Geh\_Klassen)!
  - Welche Kunden in der Tabelle „Kunden“ stehen alphabetisch hinter dem Kunden mit dem Namen „Mueller“?
  - Von welchen Berufen gibt es mehr als drei Angestellte?
  - Berechnen Sie das durchschnittliche Gehalt aller Angestellten der gleichen Abteilung! Geben Sie auch den Namen der Abteilung mit aus!
  - Ermitteln Sie die Summe über alle Lieferungen je Teil und Lieferant!
  - Welche Kunden haben mindestens einen Artikel bestellt, den auch Herr Bernhardt Falk bestellt hat?
  - Listen Sie Nachnamen, Vornamen, Gehalt und Abteilungsnamen der Informatiker auf, die in Köln beschäftigt sind!
- 20** Ergänzen Sie die Byce & Co.-Datenbank unter Oracle um folgende Datenbankobjekte:
- Erstellen Sie eine View, in der alle Attribute enthalten sind, die für eine Rechnung benötigt werden.
  - Erstellen Sie eine physikalische Kopie der Angestelltentabelle unter dem Namen Angestellte\_Kopie!
- 21** Formulieren Sie Anfragen an die Byce & Co.-Datenbank unter Oracle und MySQL, die diese Fragen beantworten:
- Welche Angestellten bearbeiten keine Aufträge?
  - Welche Angestellten verdienen mehr als der Durchschnitt aller Gehälter?
  - Welche Angestellten verdienen mehr als der Durchschnitt aller Gehälter von Angestellten der gleichen Abteilung?
  - Ermitteln Sie alle Orte, in denen Kunden oder Angestellte wohnen oder beide!
  - Ermitteln Sie die Produktnamen derjenigen Rohstoffe (Teile.Typ = 'Material'), die nicht zur Produkterzeugung verwendet werden! Geben Sie diese Produktbezeichnungen in Großbuchstaben aus!

- Welche Kunden haben Artikel bestellt, die nicht auf Lager sind?
  - In welchen Abteilungen sind alle Berufe der Unternehmung vertreten?
  - In welchen Abteilungen sind die meisten Berufe vertreten?
  - In welchen Teilen findet die Silberfarbe (TNr = 3) (nur eine Stufe) Verwendung?
  - In welchen Teilen findet die Silberfarbe (TNr = 3) (über alle Stufen) Verwendung?
  - Welche Artikel sind in allen Lagern vorhanden?
- 22** Welche Ergebnisse liefern die folgenden fünf SELECT-Anweisungen? Erläutern Sie die Unterschiede!
- ```
SELECT Ang_Nr , Gehalt
FROM Angestellte a
WHERE Gehalt > (SELECT AVG(Gehalt)
                  FROM Angestellte b
                  WHERE a.Abt_Nr = b.Abt_Nr );
```
  - ```
SELECT Ang_Nr , Gehalt
FROM Angestellte
WHERE Gehalt > (SELECT AVG(Gehalt)
                  FROM Angestellte);
```
  - ```
SELECT Ang_Nr , Gehalt
FROM Angestellte
WHERE Gehalt > ALL (SELECT AVG(Gehalt)
                      FROM Angestellte
                      GROUP BY Abt_Nr);
```
  - ```
SELECT Ang_Nr , Gehalt
FROM Angestellte
WHERE Gehalt > ANY (SELECT AVG(Gehalt)
                      FROM Angestellte
                      GROUP BY Abt_Nr);
```
  - ```
SELECT Ang_Nr , Gehalt
FROM Angestellte
WHERE Gehalt > (SELECT AVG(AVG(Gehalt))
                  FROM Angestellte
                  GROUP BY Abt_Nr);
```

- 23** Konstruieren Sie eine Tabelle, mindestens eine INSERT-Anweisung und eine SELECT-Anweisung

a. für die die Anfrage

```
SELECT * FROM Tabelle
WHERE Spalte <> ANY (SELECT Spalte FROM Tabelle);
```

nicht die komplette Tabelle liefert.

- b. Konstruieren Sie eine Tabelle, mindestens eine INSERT-Anweisung und eine SELECT-Anweisung, für die die Anfrage

```
SELECT * FROM Tabelle
WHERE Spalte = ALL (SELECT Spalte FROM Tabelle);
```

nicht die leere Menge zurückgibt.

- 24** Schreiben Sie eine SELECT-Anweisung, die die Selektivität des Felds „Nachname“ in der Angestelltentabelle der Byce & Co. misst (Selektivität = Anzahl unterschiedlicher Werte dividiert durch Anzahl der Werte).

- a. Durch eine SELECT-Anweisung an die Tabelle Angestellte  
b. Durch eine SELECT-Anweisung auf das Data Dictionary von Oracle (USER\_TAB\_COLUMNS)  
c. Ist eine Abfrage wie unter b) auch bei MySQL möglich?

- 25** Beantworten Sie die folgenden Anfragen an die Rollo-WM-Datenbank:

- a. In welchen Spielen (Ausgabe: Mannschaft\_1 : Mannschaft\_2, Ausführungsdatum, Spieltag, Anzahl gelbe Karten, Anzahl rote Karten), an denen Deutschland teilgenommen hat, gab es rote oder gelbe Karten?  
b. Welche Nationen haben in der Vorrunde gespielt?  
c. Welche Nationen haben ausschließlich in der Vorrunde gespielt?  
d. Welcher Spieler lebt nicht in dem Land, in dessen Nationalmannschaft er spielt?  
e. Listen Sie die Gruppen auf mit der Anzahl an Toren, die während der Vorrunde geschossen wurden?  
f. Wieviele Zuschauer gab es für jede Runde des Turniers (Typ des Spiels)?  
g. Welches Spiel hatte die meisten roten Karten?

- 26** Welche der folgenden Anfragen liefern bei dem gegebenen Datenbestand der Lieferantentabelle gleiche Ergebnisse und welche sind unabhängig vom Datenbestand? Begründen Sie Ihre Antwort.

- a. `SELECT COUNT(*) FROM Lieferanten;`  
b. `SELECT COUNT(Lief_Nr) FROM Lieferanten;`  
c. `SELECT COUNT(Name) FROM Lieferanten;`  
d. `SELECT COUNT(TelefonNr) FROM Lieferanten;`  
e. `SELECT COUNT(Zeitstempel) FROM Lieferanten;`

- 27** Welche der folgenden SELECT-Anweisungen sind semantisch äquivalent zu a) und welches ist ihre Semantik? Erläutern Sie, warum die anderen unterschiedliche Ergebnisse liefern und welches deren Semantik ist! Wenn möglich, formulieren Sie die Anfragen so um, dass die Semantik der Anfrage a) entspricht.

a.

```
SELECT Nachname, Vorname, Nationname
FROM Spieler
WHERE Spieler_ID NOT IN (SELECT Spieler_ID FROM Tore);
```

b.

```
SELECT Nachname, Vorname, Nationname
FROM Spieler
WHERE NOT EXISTS (SELECT Spieler_ID FROM Tore);
```

c.

```
SELECT Nachname, Vorname, Nationname
FROM Spieler
WHERE Spieler_ID <> ANY (SELECT Spieler_ID FROM Tore);
```

d.

```
SELECT Nachname, Vorname, Nationname
FROM Spieler
WHERE Spieler_ID <> ALL (SELECT Spieler_ID FROM Tore);
```

e.

```
SELECT Nachname, Vorname, Nationname
FROM Spieler, Tore
WHERE Spieler.Spieler_ID <> Tore.Spieler_ID;
```

f.

```
SELECT Nachname, Vorname, Nationname
FROM Spieler
MINUS
SELECT Nachname, Vorname, Nationname
FROM Spieler INNER JOIN Tore USING (Spieler_ID);
```

- 28** Was ist die Semantik der folgenden Rollo-Anfragen?

a.

```
SELECT Nachname, Vorname, Nationenname
FROM Spieler
WHERE Nationenname = Land;
```

b.

```
SELECT s.Nachname, s.Vorname, s.Nationname,
      p.Mannschaft_1, p.Mannschaft_2, t.Minute
FROM Spieler s, Tore t, Spiele p
WHERE s.Spieler_Id = t.Spieler_Id
  AND t.Spiel_Id = p.Spiel_Id
  AND t.Minute > 90;
```

c.

```
SELECT SUM(Anzahl_rote_Karten)
FROM Spiele
WHERE Typ = 'Vorrunde';
```



d.  
SELECT Typ, SUM(Anzahl\_rote\_Karten)  
FROM Spiele  
GROUP BY Typ;

e.  
SELECT Mannschaft\_1, Mannschaft\_2, Termin  
FROM Spiele  
WHERE Anzahl\_Zuschauer = (SELECT MIN(Anzahl\_Zuschauer)  
FROM Spiele);

**Aufgabe zum Abschnitt 5.6 (Data Administration Language):**

- 29 Erstellen Sie ein SQL-Skript, mit dem Ihr Berechtigungskonzept aus dem Datenbankschema des Systems Rollo umgesetzt wird.

Weitere Kontrollfragen zu diesem Kapitel finden Sie unter der Companion-Webseite des Pearson-Verlages <http://www.pearson-studium.de/> auf der Begleitseite unseres Buches. Wählen Sie dort bitte im Multiple-Choice-Test das Fach „DBS“ und den Punkt „Kapitel5/Die Datenbanksprache SQL2003“ aus.

Zusätzlich wird auf der Begleitwebseite noch ein SQL-Trainer angeboten, mit dem Sie online SELECT-Abfragen lösen können und ein Tool, welches automatisch SELECT-Abfragen in Operatorbäume umwandelt. Auch die SQL-Skripte zur Erzeugung der Datenbankschemata Rollo sowie Byce & Co. finden Sie auf der Begleitwebseite.

