Stand: 20. Oktober 2000

A Relationale Datenbanken und SQL

Inhalt

A.1 Definitionen

Unter einer *Datenbank* ist eine auf Dauer angelegte Datenorganisation zu verstehen, und unter dem Begriff *Datenorganisation* werden alle Verfahren zusammengefasst, die dazu dienen, Daten zu strukturieren, zu speichern und verfügbar zu halten. In relationalen Datenbanken sind die Strukturelemente *Relationen*, zwischen denen logische Abhängigkeiten bestehen. Relationen manifestieren sich in der Praxis fast immer als *Tabellen*, d.h. die beiden Begriffe Relation und Tabelle können fast immer synonym verwenden werden. Im Folgenden wird vorzugsweise der Terminus Tabelle verwendet.
Ein Datenbankverwaltungssystem (DBMS – DataBase Management System) oder kurz Datenbanksystem (DBS) ist ein Programmsystem zur Verwaltung der Daten von Datenbanken. RDBMS ist die relationale DBMS-Variante. Sie ist der heute vorherrschende Datenbanktyp.
Datenbanksprachen dienen dem Zugriff auf die Bestände und Strukturen einer Datenbank. Sie ermöglichen in solchen Beständen das Suchen, Löschen, Verändern und Einfügen von Daten in Tabellen (`SELECT`, `DELETE`, `UPDATE` und `INSERT`) bzw. die Definition, Änderung und Elimination der Tabellen selbst (`CREATE`, `ALTER`, `DROP`). Die in Klammern stehenden Begriffe entstammen der prominentesten unter diesen Sprachen, nämlich SQL, die in nahezu allen RDBMS üblich ist.
Die Tabellen relationaler Datenbanken lassen sich mittels SQL zu virtuellen Tabellen, die auch als Sichten oder Views bezeichnet werden, zusammenbinden.
Die Elemente relationaler Datenbanken sind also Tabellen (oder Relationen, s.o.), die zueinander in Beziehung stehen. Jede Tabelle hat einen Tabellennamen und besteht aus *Spalten*. Und jede Spalte hat ihrerseits einen Spaltennamen und besteht aus in *Zeilen* angeordneten Werten (Spalten heißen manchmal auch Attribute und Spaltenwerte Ausprägungen der Attribute).
Mit den Werten von Spalten ist der Begriff *Domäne* verbunden. Darunter ist die Menge der erlaubten Werte in einer Spalte, also ihr Wertebereich zu verstehen.
Zur Verdeutlichung folgt ein einfaches Beispiel in Form der Tabelle `Personen` mit den Spalten `nachname`, `vorname` und `pcode` (`pcode` stellt eine Kodierung dar, über die jede der Zeilen in der Tabelle eindeutig identifiziert ist, vgl. Abschnitt 2.7).

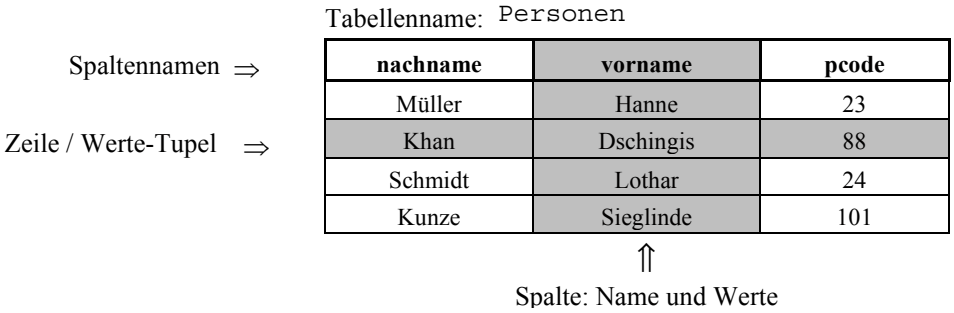


Abb. 1-1: Bezeichnungskonventionen in Tabellen

Zunächst sei das Vorangegangene in einer der üblicheren Schreibweisen auch etwas formaler zusammengefasst:

```
Tabelle (spalte1, spalte2, spalte3, ...)
```

ist eine Tabelle mit in der Klammer aufgelisteten Spalten, z.B.

```
Personen (nachname, vorname, pcode)
```

`spalte1`, `spalte2`, `...` bzw. `nachname`, `vorname`, `pcode` sind demgemäß die Spalten der Tabelle `Tabelle` bzw. `Personen`. Sie sind jeweils homogene *Mengen* von Zeichenketten, Zahlenwerten etc. Die Elemente solcher Mengen bilden die Werte der Spalten. Wendet man die Mengenschreibweise auf Spalten der Beispieltabelle an, so ergeben sich:

```
nachname   = { Müller, Khan, Schmidt, Kunze }
vorname    = { Lothar, Dschingis, Hanne, Sieglinde }
pcode      = { 101, 23, 88, 24 }
```

und daraus die Tabelle `Personen`, dargestellt als Menge `{...}` von n-Tupeln `(...)`, als Teilmenge der Kombintion von Spaltenwerten:

```
Personen( nachname, vorname, pcode )
          = {   (Müller, Hanne, 23),
                (Khan, Dschingis, 88),
                (Schmidt, Lothar, 24),
                (Kunze, Sieglinde, 101)  }
```

An Tabellen können Operationen ausgeführt werden, und in den Operationen werden unterschieden:

- *unäre* oder *monadische Operatoren*, die nur auf eine Tabelle, und
- *binäre* oder *dyadische Operatoren*, die auf zwei Tabellen einwirken.

Das Ergebnis aller auf Tabellen ausgeführter Operationen sind selbst wieder Tabellen (Abgeschlossenheitseigenschaft der Tabellen). Auf Resultattabellen von Operationen sind wiederum Operationen ausführbar etc. Operationen können also nach Belieben verschachtelt werden.

A.2 Beziehungen zwischen Tabellen

Tabellen dürfen nicht isoliert betrachtet werden, denn zwischen ihnen bestehen gewöhnlich *Beziehungen* oder Assoziationen. So gehören in Abb. 1-2 die beiden Zeilen `kcode=2` und `kcode=4` in der Tabelle Kurse zu der Zeile mit `dcode=3` in der Tabelle `Dozenten`. Zu einer Zeile in Dozenten können also grundsätzlich keine (`dcode=27`) oder eine (`dcode=5`) oder `2` oder allgemein `N` Zeilen in `Kurse` gehören. Man sagt, dass zwischen den beiden Tabellen eine Beziehung des Typs `1:N` bzw. eine `1:N`-Beziehung besteht.
Eine Sammlung von Tabellen wird erst durch solche Beziehungen zwischen den Tabellen zur relationalen Datenbank, und auf derartigen Beziehungen zwischen Tabellen baut insbesondere die noch zu besprechende Tabellenoperation Join (Abschnitt 2.5) auf.

`Dozenten`

dcode	nachname	vorname
2	Leutner	Brigitte
3	Gernhardt	Wolfgang
4	Weizenbaum	Josephine
5	Duffing	Julienne
9	Mergel	Börries
10	Ludwig	Luigi
27	Meyer-Böricke	Julius

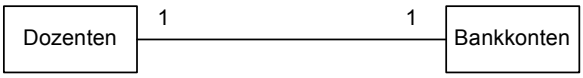
`Kurse`

kcode	dcode	bezeichnung	datum
1	10	Objektorientierte	27.04.98
2	3	JavaScript	29.06.98
3	2	JDBC	30.06.98
4	3	HTML	13.07.98
5	5	GUI-Programmie	09.06.98

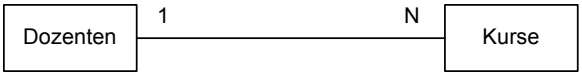
Abb. 1-2: Beziehungen zwischen zwei Tabellen

Die Beziehungstypen im einzelnen sind:

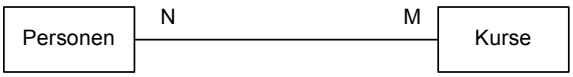
- die `1:1`-Beziehung:
  jedem Dozenten ist genau ein Bankkonto zugeordnet.



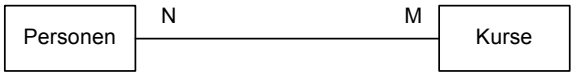
- die `1:N`-Beziehung:
  ein Dozent leitet mehrere Kurse, und jeder Kurs wird von höchstens einem bzw. genau einem Dozenten geleitet.



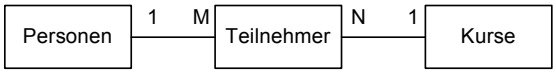
- die `N:M`-Beziehung:
  eine Person nimmt teil an mehreren Kursen, und an einem Kurs können mehrere Personen teilnehmen.



Das Ergebnis von N:M-Beziehungen zwischen Tabellen ist nicht direkt in Tabellenform darstellbar. Sie müssen daher unter Zwischenschaltung einer weiteren, *assoziativen* Tabelle in zwei 1:N-Beziehungen nach dem folgendem Vorgehensmuster aufgelöst werden: Eine Person kann mehrmals Kursteilnehmer sein, und für einen Kurs können mehrere Teilnehmer gebucht haben.
In der Musterdatenbank Kurse ist es das die assoziative Tabelle Teilnehmer, die, wie im folgenden Bild gezeigt, die N:M-Beziehung zwischen Kurse und Dozenten



in eine 1:N- und eine 1:M-Beziehung auflöst:



Beim Entwerfen einer relationalen Datenbank werden also zum einen die Daten kategorisiert und in Tabellen gesammelt und zum anderen diese Tabellen zueinander in Beziehung gesetzt, wobei auch die Typen all dieser Beziehungen festgelegt werden müssen. Dieses Vorgehen ist Bestandteil des so genannten Datenmodellierungsprozesses.

A.3 Einfache Operationen auf Tabellen

Zwei Möglichkeiten, Tabellen zu bearbeiten, bieten sich unmittelbar an, nämlich das zeilenweise und das spaltenweise Auswählen von Werten. Die entsprechenden Operationen auf eine Tabelle heißen *Projektion* für die Spaltenwahl und *Selektion* bzw. Restriktion für die Auswahl von Zeilen. Beide wirken jeweils auf eine Tabelle, sind also so genannte unäre oder monadische Operationen. Das Ergebnis sind wiederum Tabellen.
In den drei folgenden Abschnitten sollen aus der untenstehenden Tabelle Personen beispielhaft ausgewählt werden:

- die beiden Spalten nachname und pcode (A.3.1);
- die Zeilen mit allen Werten von nachname, die mit "K" beginnen (A.3.2);
- die Spalte vorname und daraus nur diejenigen Zeilen, in denen die Werte von nachname gleich "Khan" sind (A.3.3).

nachname	vorname	pcode
Müller	Hanne	23
Khan	Dschingis	88
Schmidt	Lothar	24
Kunze	Sieglinde	101

A.3.1 Projektion

Durch Projektion einer Tabelle wird aus den vorhandenen Spalten ausgewählt. Das Resultat ist wiederum eine Tabelle, die im Allgemeinen weniger Spalten als die ursprüngliche hat.
Im folgenden Beispiel wird so aus der Tabelle `Personen` mit den drei Spalten `nachname`, `vorname` und `pcode` durch Projektion eine Tabelle mit den beiden Spalten `nachname` und `pcode` erzeugt.

nachname	vorname	pcode
Müller	Hanne	23
Khan	Dschingis	88
Schmidt	Lothar	24
Kunze	Sieglinde	101

⇒

nachname	pcode
Müller	23
Khan	88
Schmidt	24
Kunze	101

Abb. 1-3: Projektion

In der Datenbanksprache SQL hat diese Operation den Namen `SELECT`. `SELECT` leitet in SQL aber außerdem eine ganze Klasse von Ausdrücken ein, die alle eine Auswahl in einer oder mehreren verknüpften Tabellen bewirken (Query oder Abfrage), ist also sowohl das Schlüsselwort für Abfragen als auch der Name des Projektionsoperators.
Die SQL-Anweisung für die im Bild gezeigte Projektion ist:

```
SELECT nachname, pcode      wähle die Spalten nachname und pcode
FROM Personen               aus der Tabelle Personen
```

Viele der geschilderten und in diesem Kapitel noch zu schildernden Sachverhalte lassen sich mit SQL einfach und präzise darstellen. Dazu wird nur eine kleine Untermenge von SQL benötigt. Aus diesem Grunde wird im Vorgriff auf Abschnitt A.7 und folgende diese Untermenge, im wesentlichen die `SELECT`-Anweisung in ihren einfachsten Varianten bereits hier eingeführt.
Man kann in der Regel eine SQL-Anweisung mittels Leerstellen und Zeilenumbrüchen freizügig formatieren, d.h. die beiden folgenden Schreibweisen sind mit der vorangehenden völlig gleichwertig:

```
SELECT nachname, pcode FROM Personen
SELECT nachname,
       pcode
FROM   Personen
```

A.3.2 Selektion

Selektion in einer Tabelle bewirkt die Auswahl von Zeilen. Das Resultat ist wiederum eine Tabelle, die höchstens gleich viele, in der Regel aber weniger Zeilen hat als die ursprüngliche.
Im Beispiel wird so aus der vierzeiligen Tabelle Personen durch Selektion eine Tabelle mit nur zwei Zeilen.

nachname	vorname	pcode
Müller	Hanne	23
Khan	Dschingis	88
Schmidt	Lothar	24
Kunze	Sieglinde	101

⇒

nachname	vorname	pcode
Khan	Dschingis	88
Kunze	Sieglinde	101

Abb. 1-4: Selektion

Die dem Bild entsprechende SQL-Anweisung lautet:

```
SELECT * FROM Personen
WHERE nachname LIKE 'K%'          nachname fängt mit K an
```

Die Tabelle `Personen` wird mit `SELECT * FROM Personen` also erst auf sich selbst projiziert, und aus dieser Tabelle werden dann mittels `WHERE` diejenigen Zeilen selektiert, in denen jeweils der entsprechende Wert in der Spalte `nachname` mit dem Buchstaben `K` beginnt (`LIKE 'K%'`). Man beachte, dass in SQL-Strings als Joker das Prozentzeichen `%` und nicht der sonst üblichere Asteriskus `*` verwendet wird.

A.3.3 Selektion und Projektion

Die Kombination von Selektion und Projektion in SQL entsprechend dem folgenden Bild sollte keine Schwierigkeiten mehr bereiten. Im wesentlichen muss nur der Asteriskus nach `SELECT` durch eine Spaltenliste ersetzt werden, um eine entsprechend reduzierte Tabelle zu erhalten.

nachname	vorname	pcode
Müller	Hanne	23
Khan	Dschingis	88
Schmidt	Lothar	24
Kunze	Sieglinde	101

⇒

vorname
Dschingis

Abb. 1-5: Selektion und Projektion

```
SELECT vorname
FROM Personen
WHERE nachname LIKE 'Khan'
```

A.4 Tabellen verbinden: Die Join-Operation

A.4.1 Grundlegendes zu Join

Ähnlich wie die Zeilen einer Tabelle als Kombination von Spaltenwerten darstellbar sind, kann das Resultat von Join-Operationen als Kombination von Tabellenzeilen aller beteiligten Tabellen beschrieben werden. Die Kombination wird durch Selektion gesteuert,

also durch Abhängigkeiten von Spalten der beteiligten Tabellen untereinander, beispielsweise über die `WHERE`-Klausel. In der überwiegenden Zahl der Fälle werden Tabellen über gleiche Spaltenwerte zusammengefügt. Dazu ein Beispiel in SQL (dem man zugleich entnehmen kann, wie gleiche Spaltenbezeichnungen in unterschiedlichen Tabellen durch Qualifizieren mit dem Tabellennamen unterschieden werden):

```
SELECT *
FROM   Kurse, Dozenten
WHERE  Kurse.dcode = Dozenten.dcode
```

Die beiden Tabellen `Kurse` und `Dozenten` werden in allen folgenden Beispielen verwendet und daher in der folgenden Abb. 1-6 einleitend vorgestellt.

`Dozenten`

dcode	nachname	vorname
2	Leutner	Brigitte
3	Gernhardt	Wolfgang
4	Weizenbaum	Josephine
5	Duffing	Julienne
9	Mergel	Börries
10	Ludwig	Luigi
27	Meyer-Böricke	Julius

`Kurse`

kcode	dcode	bezeichnung	datum
1	10	Objektorientierte	27.04.98
2	3	JavaScript	29.06.98
3	2	JDBC	30.06.98
4	3	HTML	13.07.98
5	5	GUI-Programmie	09.06.98

Abb. 1-6: Basistabellen für Join-Operationen

Die obenstehende SQL-Anweisung, die auf diese beiden Tabellen angewandt wird, hat als Resultat diese Tabelle:

Kurse. kcode	Kurse. bezeichnung	Kurse. datum	Kurse. dcode	Dozenten. dcode	Dozenten. nachname	Dozenten. vorname
3	JDBC	30.06.98	2	2	Leutner	Brigitte
2	JavaScript	29.06.98	3	3	Gernhardt	Wolfgang
4	HTML	13.07.98	3	3	Gernhardt	Wolfgang
5	GUI-Programmierung	09.06.98	5	5	Duffing	Julienne
1	Objektorientierte Progra	27.04.98	10	10	Ludwig	Luigi

Die Operation verbindet die beiden Tabellen zu einer einzigen Tabelle und trägt daher die Bezeichnung Join.
In die SQL-Anweisung kann noch eine Projektion einbezogen werden, mit der überflüssige oder unerwünschte Spalten eliminiert werden können:

```
SELECT Kurse.datum, Dozenten.nachname,
       Dozenten.vorname, Kurse.bezeichnung
FROM Kurse, Dozenten
WHERE Kurse.dcode = Dozenten.dcode
```

Das Resultat dieser Anweisung ist:

Kurse. Datum	Dozenten. nachname	Dozenten. vorname	Kurse. bezeichnung
30.06.98	Leutner	Brigitte	JDBC
29.06.98	Gernhardt	Wolfgang	JavaScript
13.07.98	Gernhardt	Wolfgang	HTML
09.06.98	Duffing	Julienne	GUI-Programmierung mit Java
27.04.98	Ludwig	Luigi	Objektorientierte Programmierung mit Java

Die Join-Operation hat eine Vielzahl von Varianten, weil unterschiedliche Spaltenkombinationen projiziert werden können und vom Gleichheitsoperator abweichende Operatoren in der `WHERE`-Klausel möglich sind. Für einige dieser Varianten haben sich feste Bezeichnungen eingebürgert. Diese Varianten sind in Form von beispielhaften SQL-Anweisungen im Folgenden zusammengestellt. Jede Variante ist außerdem mit einem Beispiel veranschaulicht, das die beiden am Anfang des Abschnittes gezeigten Tabellen `Kurse` und `Dozenten` als Grundlage hat.

A.4.2 Theta-Join

Zusammenfügen über gleiche Spaltenwerte ist zwar die häufigste, aber nicht die einzige Art, Tabellen zu verknüpfen. Spaltenwerte können auch über beliebige andere Bedingungsoperatoren zueinander in Beziehung gesetzt werden. Darauf beruhende Operationen werden als Theta- oder θ-*Join* bezeichnet, wobei Theta bzw. θ beliebige Bedingungsoperatoren symbolisieren.

```
SELECT * FROM Personen, Dozenten
WHERE Personen.nachname θ Dozenten.nachname
```

Ein Beispiel für den Stringvergleichsoperator θ = `NOT LIKE`:

```
SELECT * FROM Personen, Dozenten
WHERE Personen.nachname NOT LIKE Dozenten.nachname
```

Personen. pcode	Personen. nachname	Personen. vorname	Dozenten. dcode	Dozenten. nachname	Dozenten. vorname
23	Müller	Hanne	2	Leutner	Brigitte
88	Khan	Dschingis	2	Leutner	Brigitte
24	Schmidt	Lothar	2	Leutner	Brigitte
101	Kunze	Sieglinde	2	Leutner	Brigitte
23	Müller	Hanne	3	Gernhardt	Wolfgang
88	Khan	Dschingis	3	Gernhardt	Wolfgang
usw.	...	...	...	...	...

Abb. 1-7: Verknüpfung mit θ-Join (θ = `NOT LIKE`)

Die Ergebnistabelle verfügt über alle Spalten der beteiligten Tabellen, verbindet aber nur diejenigen Zeilen, in denen die Werte der beiden Spalten `Personen.nachname` und `Dozenten.nachname` ungleich sind. (Bei numerischen Werten würde der Ungleichheitsoperator `!=` an die Stelle von `NOT LIKE` gesetzt werden.)

A.4.3 Equi Join

Equi Join ist ein θ-Join mit dem Gleichheitszeichen an der Stelle von θ bei Zahlen oder `LIKE` bei Strings, z.B.

```
SELECT * FROM Dozenten, Kurse
WHERE Kurse.dcode = Dozenten.dcode
```

Das Resultat besteht aus allen Spalten der beteiligten Tabellen, also auch den beiden inhaltlich gleichen Spalten, die in der `WHERE`-Klausel verwendet sind.
Das gleiche Ergebnis erhält man auch bei Verwendung des `INNER JOIN`-Operators, den allerdings nicht jedes Datenbanksystem kennt (Access kennt ihn, Oracle dagegen nicht):

```
SELECT * FROM Dozenten
INNER JOIN Kurse
ON Dozenten.dcode = Kurse.dcode;
```

Für beide SQL-Ausdrücke ist das Ergebnis das gleiche, nämlich:

Dozenten. dcode	Dozenten. nachname	Dozenten. vorname	Kurse. kcode	Kurse. dcode	Kurse. bezeichnung	Kurse. datum
2	Leutner	Brigitte	3	2	JDBC	30.06.98
3	Gernhardt	Wolfgang	2	3	JavaScript	29.06.98
3	Gernhardt	Wolfgang	4	3	HTML	13.07.98
5	Duffing	Julienne	5	5	GUI-Programmierung	09.06.98
10	Ludwig	Luigi	1	10	Objektorientierte Prog	27.04.98

Abb. 1-8: Verknüpfung mit Equi Join

A.4.4 Natural Join

```
SELECT nachname, vorname, Kurse.*
FROM Dozenten, Kurse
WHERE Kurse.dcode = Dozenten.dcode
```

Ein *Natural Join* ähnelt dem Equi Join, das Resultat beinhaltet aber nur eine der in der `WHERE`-Klausel angegebenen Spalten (`Kurse.dcode` und `Dozenten.dcode` haben identische Werte, d.h. eine der beiden Spalten ist überflüssig und wird weggelassen). Dies wird sozusagen als die natürliche Weise angesehen, wie Tabellen zu verbinden sind.
Das Ergebnis der Operation ist die folgende Tabelle:

Dozenten. nachname	Dozenten. vorname	Kurse. kcode	dcode	Kurse. bezeichnung	Kurse. datum
Leutner	Brigitte	3	2	JDBC	30.06.98
Gernhardt	Wolfgang	2	3	JavaScript	29.06.98
Gernhardt	Wolfgang	4	3	HTML	13.07.98
Duffing	Julienne	5	5	GUI-Programmierung mit Java	09.06.98
Ludwig	Luigi	1	10	Objektorientierte Programmierung m	27.04.98

Abb. 1-9: Verknüpfung mit Natural Join

A.4.5 Outer Join

Outer Join ist wie alle Joins eine binäre oder dyadische Operation, mittels derer alle Zeilen der einen Tabelle mit einer passenden Auswahl von Zeilen der anderen Tabelle verbunden werden. Dabei kann es geschehen, dass zu Tabellenzeilen keine Entsprechungen in der anderen Tabelle existieren, also Lücken entstehen. Solche Lücken werden in der Resultattabelle mit Nullwerten aufgefüllt. Nullwerte zeigen an, dass ein Wert fehlt.
Die Syntax ist

```
... linkeTabelle XXX [OUTER] JOIN rechteTabelle ON bedingung
```

`XXX` kann den Wert `LEFT` oder `RIGHT` haben. Mit `OUTER` kann optional ein Outer Join deutlich symbolisiert werden, ohne irgendwelche sonstigen Wirkungen zu haben. `LEFT` und `RIGHT` geben an, welche der Tabellen als Ganzes verwendet wird, nämlich die von der linken Seite des Operators bei `LEFT` und die von der rechten Seite bei `RIGHT`. Im folgenden Beispiel

```
SELECT * FROM Dozenten
LEFT [OUTER] JOIN Kurse ON Dozenten.dcode=Kurse.dcode
```

besteht das Ergebnis also aus allen Zeilen der links stehenden Tabelle `Dozenten` verbunden nur mit den Zeilen der rechts stehenden Tabelle `Kurse`, für die die Bedingung in der `ON`-Klausel zutrifft. Genau das gleiche Ergebnis erhält man, wenn `LEFT` durch `RIGHT` ersetzt und außerdem `Kurse` und `Dozenten` vertauscht werden:

```
SELECT * FROM Kurse
RIGHT [OUTER] JOIN Dozenten ON Kurse.dcode=Dozenten.dcode
```

Die Resultattabelle enthält alle Zeilen der rechts stehenden Tabelle `Dozenten` und nur die Zeilen der links stehenden Tabelle `Kurse`, für die die Bedingung in der `ON`-Klausel zutrifft.
Der folgende SQL-Ausdruck zeigt, dass ein Outer Join auch durch Vereinigung (`UNION`) zweier getrennt berechneter Tabellen darstellbar ist.

```
(SELECT *
FROM Dozenten, Kurse
WHERE Kurse.dcode = Dozenten.dcode)
UNION
SELECT Dozenten.*, null, null, null, null, null, null
FROM Kurse, Dozenten
WHERE Dozenten.dcode NOT IN (SELECT dcode FROM Kurse)
```

`null` steht als Symbol für typgerechte Nullwerte.
Outer Join ist also keine elementare Operation wie etwa Equi Join bzw. Natural Join, sondern dient lediglich einer vereinfachten Schreibweise für eine häufig benötigte Operation. In den folgenden beiden Beispielen sind ein `LEFT JOIN` und ein `RIGHT JOIN` gezeigt. Als erstes Beispiel wird in Abb. 1-10 das Ergebnis des folgenden SQL-Ausdrucks für `LEFT JOIN` angezeigt:

```
SELECT * FROM Dozenten
LEFT JOIN Kurse ON Dozenten.dcode=Kurse.dcode
```

Dozenten. dcode	Dozenten. nachname	Dozenten. vorname	Kurse. kcode	Kurse. dcode	Kurse. bezeichnung	Kurse. datum
2	Leutner	Brigitte	3	2	JDBC	30.06.98
3	Gernhardt	Wolfgang	2	3	JavaScript	29.06.98
3	Gernhardt	Wolfgang	4	3	HTML	13.07.98
4	Weizenbaum	Josephine				
5	Duffing	Julienne	5	5	GUI-Programmier	09.06.98
9	Mergel	Börries				
10	Ludwig	Luigi	1	10	Objektorientierte	27.04.98
27	Meyer-Böricke	Julius				

Abb. 1-10: Verknüpfung mit Left Outer Join

Im zweiten Beispiel ist `LEFT JOIN` durch `RIGHT JOIN` ersetzt, das Ergebnis dafür ist in Abb. 1-11 zu sehen:

```
SELECT * FROM Dozenten
RIGHT JOIN Kurse ON Dozenten.dcode=Kurse.dcode
```

Dozenten. dcode	Dozenten. nachname	Dozenten. vorname	Kurse. kcode	Kurse. dcode	Kurse. bezeichnung	Kurse. datum
2	Leutner	Brigitte	3	2	JDBC	30.06.98
3	Gernhardt	Wolfgang	2	3	JavaScript	29.06.98
3	Gernhardt	Wolfgang	4	3	HTML	13.07.98
5	Duffing	Julienne	5	5	GUI-Programmieru	09.06.98
10	Ludwig	Luigi	1	10	Objektorientierte Pr	27.04.98

Abb. 1-11: Verknüpfung mit Right Outer Join

A.4.6 Inner Join

Inner Join entspricht vollständig dem weiter oben ausgeführten Equi Join:

```
SELECT * FROM Dozenten
INNER JOIN Kurse ON Dozenten.dcode=Kurse.dcode
```

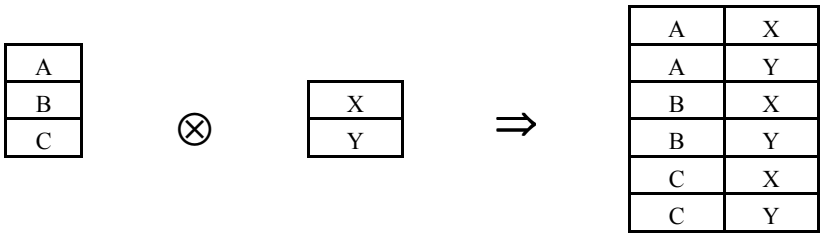
`INNER` darf, anders als `OUTER`, *nicht* weggelassen werden!

A.5 Weitere Operationen auf Tabellen

Alle noch fehlenden Tabellenoperationen werden in diesem Abschnitt ihrer Bedeutung entsprechend nur sehr kurz aufgeführt. Mit Ausnahme der Produktbildung (`TIMES`) sind diese Operationen allerdings nicht bei allen Datenbanken realisiert.

Produkt (`TIMES`)

Produktbildung ergibt eine Tabelle, die aus all den Zeilen besteht, die man erhält, wenn man jede Zeile einer Tabelle mit jeder einer anderen Tabelle kombiniert (so genanntes kartesisches Produkt oder Kreuzprodukt).



So wie bei der Join-Operation ist auch die Produktbildung oft nicht durch einen eigenen Operatornamen gekennzeichnet, sondern wird meist automatisch dann ausgeführt, wenn in der `FROM`-Klausel mehr als eine Tabelle angegeben wird.
SQL:

```
... FROM tabellenListe
```

(ohne `WHERE`-Klausel!)

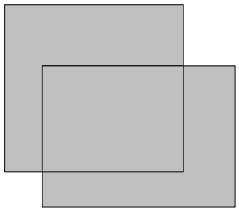
Ein Beispiel dazu:
Das Ergebnis des SQL-Ausdrucks

```
SELECT bezeichnung, nachname FROM Kurse, Dozenten
```

ist die folgende Tabelle, in der jede Zeile der Tabelle `Kurse` mit jedem Wert der Tabelle `Dozenten` kombiniert ist und das Ergebnis auf die Spalten `Kurse.bezeichnung` sowie `Dozenten.nachname` projiziert werden:

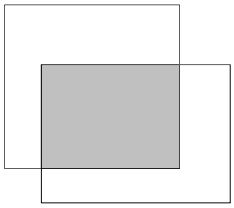
Kurse.bezeichnung	Dozenten.nachname
Objektorientierte Programmierung mit Java	Weizenbaum
JavaScript	Weizenbaum
JDBC	Weizenbaum
HTML	Weizenbaum
GUI-Programmierung mit Java	Weizenbaum
Servlets	Weizenbaum
WWW	Weizenbaum
Objektorientierte Programmierung mit Java	Ludwig
JDBC	Ludwig
...	...

Vereinigung (`UNION`)



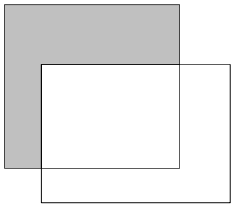
Die Vereinigung zweier Tabellen ergibt eine Tabelle, die diejenigen Zeilen der beiden Tabellen enthält, die zumindest in einer der beiden enthalten sind.
Ein Beispiel für die `UNION`-Operation ist im Zusammenhang mit Outer Joins in Abschnitt A.4.5 zu finden.

Durchschitt (`INTERSECT`)



Ergibt eine Tabelle, die aus allen Zeilen besteht, die sowohl in der einen als auch in der anderen Tabelle enthalten sind.

Differenz (`EXCEPT`)

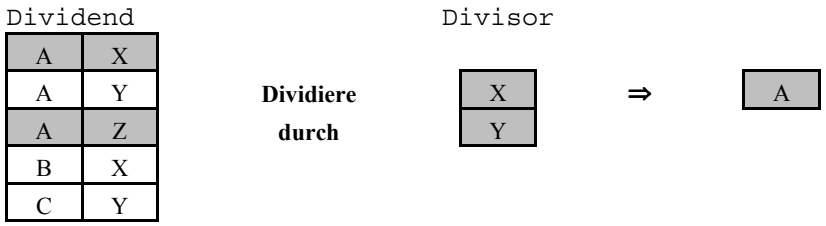


Differenzbildung aus zwei Tabellen ergibt eine Tabelle mit allen Zeilen, die in der ersten, aber nicht in der zweiten Tabelle enthalten sind.

Division (`DIVIDE`, `DIVIDEBY`)

Was eine Division zweier Tabellen bedeutet, kann am einfachsten anhand eines Beispiels gezeigt werden:
Gegeben sind zwei Tabellen mit zwei bzw. einer Spalte (Dividend und Divisor). Eine der Spalten des Dividenden und die Divisorspalte haben den gleichen Typ, d.h entnehmen

ihre Werte dem gleichen Wertevorrat, nämlich X, Y, Z in der einen und X, Z in der anderen Spalte.



Nach Ausführung der Division erscheint ein Wert aus der nicht-gemeinsamen Spalte der Dividendentabelle in der Ergebnistabelle nur dann, wenn jedem der Divisorwerte mindestens einmal dieser Wert zugeordnet ist.

A.6 Schlüssel

Eine wichtige Eigenschaft einer Datenbanktabelle ist Eindutigkeit, d.h. dass keine Zeile mehrfach vorkommt, sich also nicht wiederholt. Ist die Tabelle frei von Wiederholungen, so werden die Zeilen durch die Werte von mindestens einer Spalte und von höchstens allen Spalten der Tabelle eindeutig. Spalten, die alleine für die Eindeutigkeit der ganzen Zeile sorgen, die Zeile also identifizieren, nennt man Schlüssel.
Grundsätzlich können in einer Tabelle unterschiedliche Spalten bzw. Spaltenkombinationen als Kandidaten zur Identifizierung der Zeilen herangezogen werden. Diejenige Spalte oder Spaltenkombination, die konkret zur Identifizierung herangezogen wird, heißt Primärschlüssel.
Ein *Primärschlüssel* ist demnach also eine Spalte (oder eine Spaltengruppe) in einer Tabelle. Er ist alleine hinreichend dafür, dass alle Zeilen eindeutig sind. Es gibt also keine zwei gleichen Schlüsselwerte in einer Tabelle mit Primärschlüsselspalte(n). Wird ein Schlüssel von mehr als einer Spalte repräsentiert, so spricht man auch von einem Kombinations- oder Verbundschlüssel. Jede Tabelle kann höchstens einen Primärschlüssel haben. (Eine weitere übliche Bezeichnung für Primärschlüssel ist Identifikationsschlüssel.) Beispiele für Primärschlüssel sind die Spalte `dcode` in der Tabelle `Dozenten` und `kcode` in `Kurse`.
Sekundärschlüssel sind Spalten, über deren Werte Zeilengruppen eindeutig gekennzeichnet sind. Alle Zeilen einer Gruppe enthalten also gleiche Sekundärschlüsselwerte. Eine Tabelle kann mehrere Sekundärschlüsselspalten haben. Beispiel: die Spalte `typ` in der Tabelle `Kurse`.
Fremdschlüssel dienen dagegen nicht der Identifikation von Zeilen und Zeilengruppen, sondern sind Zeiger oder Verweise auf Schlüssel in anderen Tabellen. In Abb. 1-12 ist `dcode` in `Dozenten` ein Primärschlüssel, und `dcode` in `Kurse` verweist als Fremdschlüssel auf die entsprechenden Zeilen in der Tabelle `Dozenten`.

Dozenten

dcode	nachname	vorname
2	Leutner	Brigitte
3	Gernhardt	Wolfgang
4	Weizenbaum	Josephine
5	Duffing	Julienne
9	Mergel	Börries
10	Ludwig	Luigi
27	Meyer-Böricke	Julius

⇑

Primärschlüssel

Kurse

kcode	dcode	datum	bezeichnung
1	10	27.04.98	Objektorientiert
2	3	29.06.98	JavaScript
3	2	30.06.98	JDBC
4	3	13.07.98	HTML
5	5	09.06.98	GUI-Programmi

⇑

Fremdschlüssel:

Verweis auf `dcode` in der Tabelle `Dozenten`

Abb. 1-12: Primär- und Fremdschlüssel

Join-Operationen erfolgen demgemäß meist über die Verknüpfung von Primär- und Fremdschlüsseln, wie bereits in den vorangehenden Beispielen implizit gezeigt wurde.

A.7 SQL-Übersicht

SQL ist ein internationaler Standard für den Zugriff auf relationale Datenbanken. Als Begriff wird SQL oft gleichbedeutend mit Relational verwendet, z.B. SQL-Datenbank für relationale Datenbank. SQL gibt die adäquaten sprachlichen Mittel für den Umgang mit relationalen Datenbanken an die Hand, weshalb - gewissermaßen als Vorgriff - bereits im vorangehenden Kapitel 2 (relationale Datenbanken) SQL zur Beschreibung relationaler Operationen verwendet wurde.
Es könnte der Eindruck erweckt werden, dass eine SQL-Anweisung immer dasselbe bewirkt, also beispielsweise die Anwendung von SQL-Anweisungen auf ein und dieselbe Datenbank auf direktem Wege zu den gleichen Ergebnissen führt wie über z.B. Perl und ODBC-Treiber. Das ist nicht immer der Fall, ganz im Gegenteil können unterschiedliche Ergebnisse nie ganz ausgeschlossen werden.
Weiter ist mangelnde Kompatibilität der Resultate von SQL-Anweisungen, ausgeführt in unterschiedlichen Datenbanksystemen, eher Regel als Ausnahme. Zwar ist SQL die syntaktische Grundlage, aber meist in Form irgendeines Dialektes. Außerdem kommen zum Teil erhebliche Abhängigkeiten von der jeweiligen Rechnerplattform hinzu.
Allerdings ist SQL Grundlage von fast jedem relationalen DBMS, und die Anpassung an den jeweiligen SQL-Dialekt erfordert in der Regel nur geringen zeitlichen Aufwand.
SQL ist eine Sprache, die „aus einer Menge von Einrichtungen zur Definition, zum Zugriff und zur anderweitigen Verwaltung von SQL-Daten“ besteht ([Date 98]). Sie ist eine Entwicklung der Standardisierungsorganisationen ANSI und ISO und wurde 1992 als SQL/2 oder SQL/92 zum „International Standard ISO/IEC 9075:1992, Database Language SQL“ unverändert übernommen als DIN 66315. Der Name der Sprache ist SQL. SQL war die Abkürzung für Structured Query Language, die ihrerseits aus der Structured

English Query Language oder SEQUEL der Firma IBM entwickelt wurde (SQL wird deshalb immer noch oft wie SEQUEL ausgesprochen). SQL ist aber mittlerweile über eine reine Abfragesprache weit hinaus gediehen.
Ziel des Kapitels ist, den Einblick in die einfachsten Grundlagen von SQL weiter zu vertiefen. Der Schwerpunkt liegt auf der Handhabung oder Manipulation von SQL-Daten, die insbesondere durch die Anweisungen `SELECT`, `INSERT`, `UPDATE` und `DELETE` vertreten ist (Abschnitt A.8). Eine weitere wichtige Gruppe von Anweisungen betrifft die Definition der Daten, also das Erzeugen und Manipulieren der Tabellen und Tabellenstrukturen selbst, die kurz in Abschnitt A.9 behandelt werden.
Nicht zum SQL-Standard gehörig, aber in kommerziellen DBMS schon lange üblich sind gespeicherte Prozeduren (stored procedures), siehe Abschnitt A.7.6.

A.7.1 Skalare Operatoren und Funktionen

SQL unterstützt die üblichen arithmetischen Operatoren `+`, `-`, `*`, `/` und `mod`. Beispielsweise lassen sich so aus numerischen Spaltenwerten Zeile für Zeile Summen, Produkte etc. als errechnete Werte einer neuen, virtuellen Spalte (siehe auch Abschnitt A.7.4) bilden.
Welche skalaren Funktionen in einer Datenbank verfügbar sind, wird für zwei Datenbanksysteme als Beispiele im Folgenden gezeigt, zunächst für Access Version 8 (ACCESS 3.5 Jet):

Numerische Funktionen

```
ABS,ATAN,CEILING,COS,EXP,FLOOR,LOG,MOD,POWER,RAND,
SIGN,SIN,SQRT,TAN
```

Zeichenkettenfunktionen

```
ASCII,CHAR,CONCAT,LCASE,LEFT,LENGTH,LOCATE,LOCATE_2,
LTRIM,RIGHT,RTRIM,SPACE,SUBSTRING,UCASE
```

Zeit-/Datumsfunktionen

```
CURDATE,CURTIME,DAYOFMONTH,DAYOFWEEK,DAYOFYEAR,HOUR,
MINUTE,MONTH,NOW,SECOND,WEEK,YEAR
```

Etwas umfangreicher sind Die Funkionen bei Oracle8 Personal Edition:
Numerische Funktionen

```
ABS, CEIL, COS, COSH, EXP, FLOOR, LN, LOG, MOD, POWER, ROUND,
SIGN, SIN, SINH, SQRT, TAN, TANH, TRUNC, AVG, COUNT, GLB,
LUB, MAX, MIN, STDDEV, SUM, VARIANCE
```

Zeichenkettenfunktionen

```
CHR, INITCAP, LOWER, LPAD, LTRIM, NLS,_INITCAP, NLS,_LOWER,
NLS,_UPPER, REPLACE, RPAD, RTRIM, SOUNDEX, SUBSTR, SUBSTRB,
TRANSLATE, UPPER, ASCII, INSTR, INSTRB, LENGTH, LENGTHB,
NLSSORT, CHARTOROWID, CONVERT, HEXTORAW, RAWTOHEX, ROWIDTOCHAR,
TO_CHAR, TO_DATE, TO_LABEL, TO_MULTI_BYTE, TO_NUMBER,
TO_SINGLE_BYTE
```

Systemfunktionen

```
DUMP, GREATEST, GREATEST_LB, LEAST, LEAST_UB, NVL, UID,
USER, USERENV, VSIZE
```

Zeit-/Datumsfunktionen

```
ADD_MONTHS, LAST_DAY, MONTHS_BETWEEN, NEW_TIME, NEXT_DAY,
ROUND, SYSDATE, TRUNC
```

Sonst sei betreffs skalarer Funktionen auf die Literatur über konkrete Datenbanken und auf Handbücher dazu verwiesen.

A.7.2 Aggregatfunktionen

Aggregate Functions oder *Aggregatfunktionen* sind „Sammel"funktionen wie Summation, Durchschnittsbildung, Maximum- und Minimumbestimmung etc. Man bezeichnet sie gelegentlich auch als *Gruppenfunktionen* oder *statistische Funktionen*. Diese Funktionen reduzieren eine Aggregation bzw. Sammlung von Werten auf einen einzigen Wert. SQL stellt eingebaute Aggregatfunktionen wie `COUNT()` und `COUNT(*)` zum Abzählen von Spalten, `SUM()`, `AVG()` zum Summieren bzw. Mitteln über Spalten und `MIN()`, `MAX()` zur Bestimmung von Minimum und Maximum in Spalten bereit. Zusammen mit der `GROUP BY`-Klausel können Tabellen damit in Gruppen aufgeteilt werden und innerhalb solcher Gruppierungen Summation, Maximumbestimmung u.a. vorgenommen werden. Beispielsweise liefert

```
SELECT COUNT(*) FROM Kurse
```

die Gesamtzahl der Zeilen der Tabelle `Kurse`.

A.7.3 Cursor

Ein *Cursor* definiert einen Mechanismus, der erlaubt, eine Tabelle z.B. das Resultat einer Abfrage, Zeile für Zeile zu durchlaufen; er ist also etwas wie ein beweglicher Zeiger auf die Zeilen einer Tabelle. Cursorprimitive sind Vorwärtsbewegen des Cursors (`FETCH` in SQL) sowie Anlegen und Schließen eines Cursors. Cursor spielen auch bei der Datenbankprogrammierung mit PHP oder VBScript/ASP eine wichtige Rolle.

A.7.4 Virtuelle Tabellen

Aus Tabellen eines RDBMS können neue Tabellen zusammengestellt werden. Diese virtuellen Tabellen interpretieren gewissermaßen die zugrundeliegenden Tabellen aus einem bestimmten Blickwinkel und heißen daher auch *View*, *Viewed Table* oder *Sicht*. So erlaubt beispielsweise die virtuelle Tabelle

```
SELECT nachname, vorname, bezeichnung
FROM Dozenten, Kurse WHERE Kurse.dcode=Dozenten.dcode
```

den Blick auf die von den Dozenten angebotenen Kurse.

Virtuelle Tabellen können ähnlich wie reale Tabellen weiterverarbeitet werden, also z.B. zu Zielen von SQL-Anweisungen werden. Sie können „virtuelle“ Spalten beinhalten, das sind Spalten, in denen beispielsweise über Stückzahl, Einzelpreis und Umsatzsteuer Gesamtpreise zeilenweise bestimmt werden. So sind „Abfragen“ in MS Access solche virtuelle Tabellen, die über ODBC wie reale Tabellen behandelt werden können. Einige Einschränkungen müssen allerdings beachtet werden. Beispielsweise ist das Ändern oder Löschen von Werten in virtuellen Spalten sinnlos und daher unterbunden.

A.7.5 Transaktionen

Eine SQL-Transaktion ist eine Folge von Operationen. Eine SQL-Transaktion ist unteilbar oder *atomar*.
Die durch eine Transaktion vorgesehenen Änderungen werden insgesamt entweder durch COMMIT zur Ausführung freigegeben oder insgesamt durch ROLLBACK verworfen. Tritt ein Fehler während der Abarbeitung der SQL-Anweisungen einer Tranaktion auf, müssen alle bereits erfolgten Änderungen vollständig rückgängig gemacht werden. Änderungen werden erst nach COMMIT wirksam und damit für andere Transaktionen bzw. Datenbankoperationen sichtbar.
Ein Beispiel für eine Transaktion ist die Buchung von einem auf ein anderes Konto. Die beiden Operationen, Buchen von einem Konto, Buchen des gleichen Betrages auf ein anderes Konto, müssen entweder beide ausgeführt werden, oder es muss für beide die Ausführung unterbleiben. Sie sind also unteilbar oder atomar in einer Transaktion zusammengefasst.

A.7.6 Persistent gespeicherte Module (PSM)

Ein Mangel des SQL/92-Standards ist, dass benutzerdefinierte Funktionen und Prozeduren nicht enthalten sind. Demgegenüber bieten professionelle Datenbankprodukte bereits seit langer Zeit solche Möglichkeiten in Form von gespeicherten Prozeduren (stored procedures) oder Funktionen (stored functions) an. Verallgemeinernd spricht man auch von Persistant Stored Modules oder, abgekürzt, PSM. „Persistant“ beschreibt den Sachverhalt, dass die Module dauerhaft in der Datenbank gespeichert sind. Der Rahmen dafür ist in der Regel ein firmenspezifischer prozeduraler Sprachanteil in SQL, z.B. PL/SQL bei Oracle8 (PL steht für Procedural Language).
Zwei Gründe sind es im wesentlichen, die zu diesen Konstrukten führten. Zum einen wird damit die konventionelle strukturierte Programmiermethodik erschlossen, und zum zweiten kann die Effizienz von Datenbankoperationen erheblich verbessert werden. Letzteres wird sofort einsichtig, wenn man bedenkt, dass z.B. die Datenbankfunktionen in ASP/ ADO auf „call level“ arbeiten. Jede SQL-Anweisung wird einzeln an die Datenbank versandt, dort ausgeführt und das Ergebnis zurückgeschickt, gegebenenfalls über das Internet. Mit PSM ist es möglich, Anweisungen und Ergebnisse zu blocken, mit entsprechend positiven Auswirkungen auf die Effizienz.

Auch hierzu ein Beispiel zur Illustration, und zwar für eine Oracle8-Datenbank:
Im unten stehenden Programm ist zunächst die gespeicherte Prozedur `Zaehle` in PL/SQL (Oracle Datenbank) angegeben. Als erstes sind dort der Prozedurname und ein formaler Parameter deklariert. Der Prozedurkörper besteht aus einer einzigen `SELECT`-Anweisung, mit der in der Tabelle `Kurse` mittels der Aggregatfunktion `COUNT(*)` alle Datensätze abgezählt werden und das Ergebnis per `INTO aus` über den `OUT`-Parameter `aus` an die aufrufende Instanz, etwa an ein VBScript- oder Perl-Programm, zurückgegeben wird.

```
-- Gespeicherte Oracle8-Prozedur
PROCEDURE Zaehle (aus OUT INTEGER) IS
BEGIN
  SELECT COUNT(*)
  INTO aus
  FROM Kurse;
END;
```

A.8 Datenmanipulation

Die SQL-Anweisungen zur Manipulation der Daten in einer Datenbank sind

- `SELECT` Aus einer oder mehreren Tabellen eine neue Tabelle zusammenstellen.
- `INSERT` Einfügen von neuen Zeilen in eine Tabelle.
- `UPDATE` Ändern von Zeilen in einer Tabelle.
- `DELETE` Löschen einer oder mehrerer Zeilen in einer Tabelle.

In der Regel können nur an „echten“ Tabellen Änderungen vorgenommen werden (also nicht ohne weiteres an virtuellen Tabellen oder Views!).

A.8.1 SELECT

Mit der `SELECT`-Anweisung können Tabellen – reale und virtuelle! – zu neuen Tabellen zusammengestellt werden. Dabei sind die verwendeten Tabellen das Rohmaterial, aus dem Zeilen und Spalten ausgewählt, die unterschiedlichen Tabellenoperationen wie Join und Union angewandt und neue, virtuelle Spalten berechnet werden können.
Die SELECT-Anweisung hat die folgende Syntax:

```
SELECT [ALL | DISTINCT] spaltenListe
FROM TabellenListe
[WHERE bedingungsAusdruck]
[GROUP BY spaltenListe
   [HAVING bedingungsAusdruck]  ]
[ORDER BY spaltenListe]
```

Hier und im Folgenden bedeuten

`[...]`	kann weggelassen werden;
`...|...`	Alternative, d.h. entweder der linksseitige oder der rechtsseitige Teil wird verwendet.

SELECT ... FROM ...

```
SELECT...FROM...
```

definiert die Spalten in den Tabellen, aus denen ausgewählt werden soll.

```
SELECT * FROM Tabelle1
```

wählt die ganze Tabelle `Tabelle1` aus.

```
SELECT spalte1, spalte2 FROM Tabelle1
```

ergibt dagegen eine Tabelle mit nur zwei Spalten aus `Tabelle1` (insofern hat das Schlüsselwort `SELECT` zusätzlich den Charakter einer so genannten Klausel, hier im Sinne einer Einschränkung).

Werden zwei (oder mehr) Tabellen in der `FROM`-Klausel angegeben, so wird aus diesen das kartesische oder Kreuzprodukt gebildet (`TIMES`-Operation), d.h. jede Zeile der einen Tabelle wird mit jeder Zeile der anderen kombiniert. (Bei mehr als zwei Tabellen wird analog die Zeilen aller Tabellen kombiniert.) Zu sinnvollen Ergebnissen kommt man allerdings erst durch Hinzunahme der `WHERE`-Klausel, also durch Untermengenbildung bzw. Selektion.

Beispiele:

`SELECT *` `FROM Kurse`	Ergibt die Tabelle `Kurse` selbst
`SELECT datum, bezeichnung` `FROM Kurse`	Die Resultattabelle besteht nur aus den Spalten `datum` und `bezeichnung` (Projektion).
`SELECT *` `FROM Kurse, Dozenten`	Die neue Tabelle ist die zeilenweise Kombination aus den Tabellen `Kurse` und `Dozenten`.

`SELECT` kann als Postfix `ALL` oder `DISTINCT` haben:

- `DISTINCT` heißt, dass bei gleichlautenden Zeilen nur eine in die neue Tabelle aufgenommen wird, und
- `ALL` heißt, dass alle Zeilen berücksichtigt werden, es also zu Wiederholungen kommen kann. `ALL` ist als Standardeinstellung wirksam für den Fall, dass es als Postfix weggelassen wird.

X	1
Y	2
Y	3
Y	2

⇒ **SELECT DISTINCT**

X	1
Y	2
Y	3

Abb. 1-13: SELECT DISTINCT

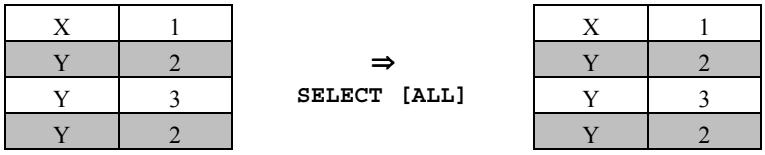


Abb. 1-14: SELECT ALL

Bei der Einbeziehung eines Primärschlüssels (siehe 2.7) in die Ergebnistabelle sind `ALL` und `DISTINCT` überflüssig, weil wirkungslos.

Die `WHERE`-Klausel

Mit so genannten Klauseln können Regeln und Einschränkungen für SQL-Ausdrücke festgelegt werden. So werden mit der `WHERE`-Klausel Zeilen aus der Ergebnistabelle von `SELECT...FROM...` ausgesondert, d.h. `WHERE` repräsentiert die Selektions-Operation:

```
... WHERE bedingungsAusdruck
```

Bedingungsausdrücke sind von booleschem Typ und entsprechend entweder wahr (`TRUE`) oder falsch (`FALSE`).
In den beiden folgenden Beispielen finden sich einfache Bedingungsausdrücke für die Tabellen Teilnehmer und Personen (`'L%'` bezeichnet alles, was mit dem Buchstaben `L` beginnt):

```
SELECT * FROM Teilnehmer WHERE kcode = 10
SELECT * FROM Personen WHERE nachname LIKE 'L%'
```

Man beachte, dass der Gleichheitsoperator unterschiedlich ist für Zahlen (`=`) und Zeichenketten (`LIKE`).
In einem weiteren Beispiel wird aus den Tabellen `Kurse` und `Dozenten` eine Verbundtabelle gebildet, und in dieser Tabelle eine Selektion durch Gleichheit von Primärschlüssel `dcode` in Dozenten und Fremdschlüssel `dcode` in `Kurse` vorgenommen (zu Schlüsseln siehe Abschnitt Kapitel A.6):

```
SELECT * FROM Kurse, Dozenten
WHERE Kurse.dcode = Dozenten.dcode
```

Dem Beispiel ist auch zu entnehmen, dass Felder gleichen Namens aber aus unterschiedlichen Tabellen dadurch unterscheidbar werden, indem sie mit ihrem Tabellennamen qualifiziert werden. Danach ist

`Kurse.dcode` die Spalte `dcode` in der Tabelle `Kurse` (Fremdschlüssel) und
`Dozenten.dcode` die Spalte `dcode` in der Tabelle `Dozenten` (Primärschlüssel).

Natürlich sind auch komplexere Ausdrücke durch Gebrauch logischer Operatoren möglich:

```
SELECT * FROM Kurse, Dozenten
WHERE Kurse.dcode = Dozenten.dcode AND
      nachname LIKE '_e%' OR nachname LIKE '_u%'
```

Das Zeichen _ ist Stellvertreter für ein beliebiges Einzelzeichen, d.h. `'_e%'` bezeichnet alle Strings mit `e` als Zeichen in der zweiten Zeichenposition.
Oder mit Klammersetzung:

```
SELECT * FROM Kurse, Dozenten
WHERE Kurse.dcode = Dozenten.dcode AND
     (nachname LIKE '_e%' OR nachname LIKE '_u%')
```

Zu beachten ist, dass das Setzen oder Weglassen von Klammern wegen der unterschiedlichen Rangigkeit der Operatoren in der Regel unterschiedliche Ergebnisse zur Folge hat.

Bedingungsoperatoren

Bedingungsoperationen ergeben immer die Werte TRUE oder FALSE.
Es gibt unäre, binäre und ternäre Bedingungsoperatoren. Die unären werden als Präfix oder Postfix vor bzw. nach ihrem Operanden notiert, die binären als Infix zwischen zwei Operanden.
Dazu jeweils ein Beispiel:

Infix:	`Nachname LIKE 'Meyer'`	(mittendrin)
	`(a > min) AND (a < max)`	
Präfix:	`NOT (a <> b)`	(am Anfang)
Postfix:	`(x > y) IS TRUE`	(am Ende)

Ternär ist der `BETWEEN-AND`-Operator, der mit dem von C oder Java her bekannten Bedingungsoperator ?: formal verglichen werden kann:

```
x BETWEEN min AND max
```

Dieser Ausdruck ist äquivalent mit dem Bedingungsausdruck

```
x >= min AND x <= max .
```

Im Folgenden sind summarisch die wichtigsten unären und binären SQL-Bedingungsoperatoren aufgeführt:

- Die binären Operatoren `AND`, `OR` und die unären Operatoren `NOT`, `IS TRUE`, `IS FALSE`, `IS NOT TRUE` und `IS NOT FALSE`, die auf Bedingungsausdrücke als Operanden wirken.
- Die elementaren binären Vergleichsoperatoren, die gleichfalls auf beliebige Ausdrücke wirken können:

  `=  <  >  <=  >=  <>`
- Auf Typkompatibilität der Operanden muss geachtet werden, d.h. eine Zeichenkette kann z.B. nicht mit einer Zahl verglichen werden.
- `LIKE` und `NOT LIKE` zum Vergleich von Zeichenketten. „Wild Card“- oder Jokerzeichen in Zeichenketten sind `%` für eine beliebige Zeichenkette und _ für ein beliebiges einzelnes Zeichen (also nicht die sonst üblicheren Zeichen `*` und `?`).
- `IN` und `NOT IN` zur Prüfung, ob ein Spaltenwert in einer Tabelle enthalten ist oder nicht.
- `UNIQUE` zur Prüfung, ob eine Zeile in einer Tabelle eindeutig ist.
- `EXISTS`, ob eine Tabelle existiert (eine „Tabelle“ mit 0 Zeilen ist in SQL nicht existent).

Welche Operatoren vor welchen Vorrang haben, ist ähnlich wie in Java oder anderen Programmiersprachen wie Perl und VBScript geregelt.

Aggregatfunktionen

Aggregatfunktionen in SQL sind:

`COUNT(spalte)`	Zählt die Zeilen in der Spalte `spalte` ab, nachdem Dubletten optional entfernt wurden (zu „optional“ siehe weiter unten).
`COUNT(*)`	Zählt alle Zeilen ab; eventuelle Dubletten sind nicht entfernbar.
`SUM(spalte)`	Spaltensumme nach optionaler Entfernung von Dubletten.
`MIN(spalte)`	Bestimmt den kleinsten Wert in der Spalte `spalte`.
`MAX(spalte)`	Bestimmt den größten Wert in der Spalte `spalte`.
`AVG(spalte)`	Mittelt die Werte der Spalte `spalte` nach optionaler Entfernung von Dubletten.

An die Stelle des Wertes spalte kann ein Ausdruck treten. Solche Ausdrücke müssen wie der Wert spalte von skalarem Datentyp sein, d.h. in SQL also entweder vom Typ einer Zeichen- bzw. Bitkette oder einer Zahl.

Bei einigen Datenbanksystemen, z.B. Oracle8, können in den Parameterklammern der Funktionen über das Präfix `DISTINCT` vor `spalte` eventuelle Dubletten vor der Funktionsausführung entfernt werden. Dagegen zählt `COUNT(*)` immer alle Zeilen, `DISTINCT` darf nicht vor den Asteriskus gestellt werden. Bei `MIN` und `MAX` ist `DISTINCT` erlaubt, aber wirkungslos.

Ein Beispiel für die `COUNT`-Funktion, angewandt auf die Spalte `typ` in der Tabelle `Kurse`:

```
SELECT COUNT(typ) FROM Kurse                  ergibt 6  und
SELECT COUNT(DISTINCT typ) FROM Kurse         ergibt 3
```

Aggregatfunktionen können nur in der `SELECT`- und in der `HAVING`-Klausel verwendet werden. Von besonderem Interesse ist ihr Zusammenspiel mit der `GROUP BY`-Klausel. Beispiele dazu finden sich in den folgenden Absätzen.

Wenn eine `SELECT`-Anweisung keine `GROUP BY`-Klausel hat, so wirken die Aggregatfunktionen auf alle Zeilen einer Tabelle.

Die `GROUP BY`- und die `HAVING`-Klausel

Als Beispiel diene die nun um eine Spalte `typ` und `zeit` erweiterte Tabelle `Kurse`. Die Spalte `typ` kennzeichnet die Veranstaltungstypen der Kurse (`S` für Seminar, `V` für Vorlesung etc.), gliedert die Kurse also in Gruppen. Ähnliches leistet `dcode`, die Kennung für die veranstaltenden Dozenten, oder `zeit`, der zeitliche Umfang eines Kurses. Die Tabelle hat also mehrere sinnvolle Sekundärschlüssel.

`Kurse`

kcode	typ	dcode	bezeichnung	datum	zeit
1	P	10	Objektorientierte Programmierung mit Java	27.04.98	10
2	S	3	JavaScript	29.06.98	5
3	P	2	JDBC	30.06.98	7,5
4	S	3	HTML	13.07.98	5
5	S	5	GUI-Programmierung mit Java	09.06.98	7,5
6	V	10	Servlets	10.06.98	7,5

Abb. 1-15: Tabelle `Kurse` mit den Sekundärschlüsseln `typ`, `dcode` und `zeit`

Eine Tabelle, die nur noch die Gliederungsbegriffe selbst beinhaltet (bei `typ` die unterschiedlichen Buchstaben), erhält man mit der folgenden SQL-Anweisung:

```
SELECT DISTINCT typ FROM Kurse
```

Das Ergebnis ist die von Dubletten bereinigte einspaltige Tabelle

typ
P
S
V

Statt der Verwendung von `DISTINCT` nach der `SELECT`-Klausel kann man auch die `GROUP BY`-Klausel verwenden, das Ergebnis ist das Gleiche:

```
SELECT typ FROM Kurse GROUP BY typ
```

Die zweite Form hat gegenüber der ersten den Vorzug, dass beispielsweise die verschiedenen Gruppen abgezählt werden können oder dass in den Gruppen einzeln summiert werden kann (in der ersten Form ist das nicht möglich).
Das ist in der folgenden SQL-Anweisung getan:

```
SELECT typ, COUNT(typ) AS Anzahl,
   MIN(zeit) AS MinZeit,
   MAX(zeit) AS MaxZeit
FROM Kurse GROUP BY typ
```

Das Ergebnis dieser Anweisung ist die Tabelle in der folgenden Abb. 1-16:

typ	Anzahl	MinZeit	MaxZeit
P	2	7,5	10
S	3	5	7,5
V	1	7,5	7,5

Abb. 1-16: Aggregatfunktionen und Gruppieren mittels `GROUP BY`

Das Beispiel zeigt auch, wie man zusätzliche, errechnete Spalten hinzufügen und mittels des Sprachelements `AS` benennen kann.
Wenn mehrere Spalten in der `GROUP BY`-Klausel angegeben werden, so wird hierarchisch gruppiert. Die erste Spalte ist dann die Hauptgruppe. Innerhalb der Hauptgruppe wird in den einzelnen Gruppierungen jeweils nach der 2. Spalte gruppiert etc.
Im Beispiel ist das anhand einer Tabelle gezeigt, in der Ergebnisse (`gewonnen`, `verloren`, `remis`) für jeden Einzelspieler (`spieler`) einer Mannschaft (`team`) Spiel für Spiel festgehalten sind.

Spiele

team	spieler	gewonnen	verloren	remis	spielnr
A	1	1			1
A	1		1		62
A	1		1		3
B	1	1			4
A	2			1	5
B	1		1		6
B	2		1		7
B	2		1		8
A	2	1			9

Abb. 1-17: Basistabelle für Beispiele mit hierarchischen Gruppierungen

Die Tabelle wird so ausgewertet, dass in einer Tabelle gruppenweise für jeden Spieler die Anzahl der gewonnenen, verlorenen und unentschiedenen Spiele berechnet wird. Dies leistet die folgende SQL-Anweisung:

```
SELECT team, spieler,
    SUM(gewonnen) AS gewonneneSpiele,
    SUM(verloren) AS verloreneSpiele,
    SUM(remis)    AS unentschiedeneSpiele
FROM Spiele
GROUP BY team, spieler;
```

Das Ergebnis ist die folgende Tabelle:

team	spieler	gewonnene Spiele	verlorene Spiele	unentschiedene Spiele
A	1	1	2	
A	2	1		1
B	1	1	1	
B	2		2	

Abb. 1-18: Hierarchische Gruppierungen: Resultat der SQL-Anweisung

Die HAVING-Klausel erlaubt eine Auswahl von ganzen Zeilengruppen, vergleichbar mit der WHERE-Klausel, die die Auswahl auf Zeilenbasis erlaubt. Verwendet werden können in der Klausel das Gruppierungsfeld selbst und die Aggregatfunktionen. In Beispielen:

```
... GROUP BY spalte HAVING spalte LIKE 'A%'
... GROUP BY spalte HAVING COUNT(*) > 4
... GROUP BY spalte HAVING MIN(spalte) > 0
```

In einem letzten Schritt wird der SQL-Ausdruck nun noch mit einer HAVING-Klausel ergänzt, so dass nur diejenigen Spieler angezeigt werden, die noch nie ein Spiel gewonnen haben:

```
SELECT team, spieler,
    SUM(gewonnen) AS gewonneneSpiele,
    SUM(verloren) AS verloreneSpiele,
    SUM(remis)    AS unentschiedeneSpiele
FROM Spiele
GROUP BY team, spieler
HAVING SUM(gewonnen) = 0;
```

team	spieler	gewonnene Spiele	verlorene Spiele	unentschiedene Spiele
B	2		2	

Abb. 1-19: Gruppierungen: Resultat der Anweisung mit HAVING-Klausel

Die ORDER BY-Klausel

Die `ORDER BY`-Klausel bringt Ordnung in Tabellen. Man kann auch mehrstufig ordnen, etwa nach Nachnamen und Vornamen. Wie die Klausel verwendet wird, ist aus den Beispielen ersichtlich.
Im ersten Beispiel wird nach `nachname` alphabetisch aufsteigend (`ASC`ending) sortiert:

```
SELECT * FROM Personen ORDER BY nachname [ASC]
101, Kunze, Sieglinde
19,  Müller, Maria
23,  Müller, Hanne
24,  Schmidt, Lothar
```

Im nächsten Beispiel wird ebenfalls nach `nachname` alphabetisch sortiert, aber nun absteigend (`DESC`ending) :

```
SELECT * FROM Personen ORDER BY nachname DESC
24,  Schmidt, Lothar
19,  Müller, Maria
23,  Müller, Hanne
101, Kunze, Sieglinde
```

Zuletzt wird aufsteigend nach `nachname` und, bei gleichen Werten von `nachname`, zusätzlich nach `vorname` sortiert:

```
SELECT * FROM Personen ORDER BY nachname, vorname
101, Kunze, Sieglinde
23,  Müller, Hanne
19,  Müller, Maria
24,  Schmidt, Lothar
```

A.8.2 INSERT

Neue Zeilen können in eine Tabelle mit der `INSERT`-Anweisung eingefügt werden. Dabei sind zwei Varianten anwendbar:

- `INSERT INTO ... VALUES ...` : In der Zieltabelle wird eine neue Zeile eingefügt und direkt mit den Werten einer Werteliste versehen.

- `INSERT INTO ... SELECT ...` : Mit `SELECT` ausgewählte Zeilen aus einer anderen Tabelle werden als neue Zeilen in die Zieltabelle importiert.

Eine Zeile direkt einfügen

In die gewählte Tabelle werden die Werte der Liste `werteListe` eingefügt, bei Weglassen der Spaltenliste in der Reihenfolge der Spalten in der Tabelle `Tabelle`, sofern keine entsprechende Spaltenliste angegeben wurde. Die Spaltenreihenfolge ist nach Standard optional, bei einigen wenigen DBS muss sie aber angegeben werden (MiniSQL in der Version 1 ist ein Beispiel dafür).

```
INSERT INTO Tabelle [(spaltenListe)] VALUES (werteListe)
```

Als Beispiel sollen der folgenden Tabelle zwei Zeilen hinzugefügt werden:

pcode	nachname	vorname
34	Hintze	Franz
88	Khan	Dschingis

Dazu müssen zwei aufeinanderfolgende `INSERT`-Anweisungen ausgeführt werden:

```
INSERT INTO Personen (vorname, nachname, pcode)
VALUES ('Leo', 'Kaiser', 91)
INSERT INTO Personen (vorname, nachname, pcode)
VALUES ('Sieglinde', 'Kunze', 101)
```

Das Ergebnis ist die folgende Tabelle:

pcode	nachname	vorname
34	Hintze	Franz
88	Khan	Dschingis
91	Kaiser	Leo
101	Kunze	Sieglinde

Aus einer anderen Tabelle einfügen

Mit `SELECT` ausgewählte Zeilen aus einer anderen Tabelle werden als neue Zeilen in die Zieltabelle nach `INTO` aufgenommen.

```
INSERT INTO Tabelle [(spaltenListe)] SELECT ...
```

Im Beispiel werden alle Dozenten, deren Nachname mit dem Buchstaben `M` beginnt, in die Tabelle `Personen` eingefügt.

```
INSERT INTO Personen (vorname, nachname)
SELECT vorname, nachname FROM Dozenten
WHERE nachname LIKE 'M%'
```

A.8.3 DELETE

Mit der DELETE-Anweisung können Zeilen aus Tabellen gelöscht werden. DELETE ist eine Operation, die auf eine Zeilen*menge* wirkt. Die Menge der zu löschenden Zeilen wird mittels der WHERE-Klausel in der DELETE-Anweisung festgelegt.
DELETE FROM tabelle [WHERE bedingungen]
Als Beispiel sollen in der folgenden Tabelle die vorletzte und letzte Zeile gelöscht werden:

pcode	nachname	vorname
34	Hintze	Franz
88	Khan	Dschingis
91	Kaiser	Leo
101	Kunze	Sieglinde

```
DELETE FROM Personen WHERE pcode=91 OR pcode=101
```

Oder alternativ

```
DELETE FROM Personen WHERE pcode>=91
```

pcode	nachname	vorname
34	Hintze	Franz
88	Khan	Dschingis

Achtung! Die Anweisung DELETE FROM Personen hinterlässt eine geleerte Tabelle.

A.8.4 UPDATE

Die UPDATE-Anweisung erlaubt, Werte in den Spalten einer Tabelle zu verändern. Wie DELETE ist auch UPDATE eine Operation, die auf Zeilenmengen einwirkt, d.h. die einschränkende Anwendung der WHERE-Klausel ist wie bei DELETE von existentieller Wichtigkeit.
Nach Angabe der Zieltabelle folgt auf SET eine Wertezuweisungsliste der Form spaltenName1=wert1, spaltenName2=wert2, etc. Wird WHERE weggelassen, so werden alle aufgeführten Spalten auf einen gleichbleibenden Wert gesetzt. Sonst werden nur diejenigen Zeilen in den Spalten geändert, für die die Bedingung in der WHERE-Klausel den Wert true hat.

```
UPDATE Tabelle SET spaltenWerteListe
[WHERE bedingungen]
```

In den folgenden Beispielen wird als Ausgangstabelle Personen verwendet:

pcode	Nachname	vorname
34	Hintze	Franz
88	Khan	Dschingis
91	Kaiser	Leo
101	Kunze	Sieglinde

Die Zeile mit `pcode=91` soll geändert werden:

```
UPDATE Personen SET nachname='Geiser', vorname='Theo'
WHERE pcode=91
```

pcode	nachname	vorname
34	Hintze	Franz
88	Khan	Dschingis
91	Geiser	Theo
101	Kunze	Sieglinde

Die folgende SQL-Anweisung trägt in allen Zeilen als `nachname` den Wert `'Geiser'` ein:

```
UPDATE Personen SET nachname='Geiser'
```

pcode	nachname	vorname
34	Geiser	Franz
88	Geiser	Dschingis
91	Geiser	Theo
101	Geiser	Sieglinde

A.9 Datendefinition

Mit den SQL-Anweisungen für die Datendefinition werden insbesondere Tabellen und Tabellenstrukturen erzeugt, geändert und ggf. wieder vernichtet. Die wichtigsten Anweisungen zu diesen Zwecken sind

- `CREATE TABLE` Eine neue Tabelle anlegen.
- `ALTER TABLE` Die Struktur einer bestehenden Tabelle verändern.
- `DROP TABLE` Eine Tabelle löschen.

Da das Anlegen von Tabellen bzw. Tabellenstrukturen und deren Pflege selten mit JDBC und meist mit anderen Mitteln erfolgt, sollen die entsprechenden SQL-Anweisungen nur kurz und ausschließlich anhand einfacher Beispiele erläutert werden.

A.9.1 CREATE TABLE

Eine Tabelle in einer Datenbank definieren:

```
CREATE TABLE TestPersonen
  ( tcode         INTEGER,
    vorname       CHAR(25),
    nachname      CHAR(25) )
```

Zu den verwendbaren Datentypen siehe Anhang B.

A.9.2 ALTER TABLE

Die Struktur einer Tabelle kann durch Hinzufügen neuer und durch Änderung des Typs bestehender Spalten manipuliert werden:

Eine neue Spalte in einer bestehenden Tabelle erzeugen

```
ALTER TABLE TestPersonen
ADD testSpalte INTEGER
```

Den Typ der Spalte ändern

```
ALTER TABLE TestPersonen
MODIFY testSpalte FLOAT NOT NULL
```

Die Spalte aus der Tabelle entfernen

```
ALTER TABLE TestPersonen
DELETE testSpalte
```

A.9.3 DROP TABLE

Eine Tabelle wird vollständig aus einer Datenbank entfernt mit

```
DROP TABLE TestPersonen
```

A.10 Literatur

Date, Chris J.: An Introduction to Database Systems.
Addison-Wesley 2000, 7. Auflage, ISBN 0-201-38590-2

Date, Chris J.; Darwen, Hugh: SQL - Der Standard.
Addison-Wesley 1998, ISBN 3-8273-1345-7

Rob, Peter; Coronel, Carlos: Database Systems. Design, Implementation, and Management. Course Technology 1999, ISBN 0-7600-1090-0

Steiner, René: Theorie und Praxis relationaler Datenbanken.
Vieweg 1996, ISBN 3-528-25427-0

van der Lans, Rick F.: Introduction to SQL.
Addison-Wesley 1994, 3. Auflage, ISBN 0-201-59618-0