5 SQL — Structured Query Language

5.1 Allgemeines zu SQL

SQL ist eine auf dem relationalen Datenmodell basierende konkrete Datenbanksprache. Das relationale Datenmodell ist ein logisches (abstraktes) Datenmodell, das man sich am besten als (high-level) abstrakte Datenbanksprache vorstellt (vgl. 1.7). So gesehen ist SQL eine *Konkretisierung des relationalen Datenmodells*. Die Abfragesprache von SQL basiert auf der relationalen Algebra und dem Tupelkalkül (vgl. 3.2 und Vorspann zu 3.3). SQL umfaßt DML (Abfragen und Mutationen) und DDL. *Zumindest ab*

Sprachentwicklung. SQL wurde im IBM-Forschungslabor in San José (Kalifornien) entwickelt. Die ersten Sprachvorschläge stammen aus 1974. Zu dieser Zeit wurde die Sprache noch SEQUEL (Structured English Query Language) genannt. Schon 1975 wurde ein erster Prototyp implementiert, nämlich SQL-XRM. Aufbauend auf den Erfahrungen mit der ersten Sprachversion und dem dazugehörigen Prototypen wurde die Sprache zu SEQUEL2 weiterentwickelt. Auch ein entsprechender Prototyp, System R, wurde in den Jahren 1976–1977 gebaut. Der Name SEQUEL mußte dann aus rechtlichen Gründen zu SQL (Structured Query Language) geändert werden.

Endbenutzerorientierung. Ein wichtiges Entwicklungsziel war die Verwendbarkeit von SQL auch durch den *Endbenutzer* (zur Formulierung von "ad-hoc queries"). Das wurde dadurch erreicht, daß SQL nicht *prozedural*, sondern *deskriptiv* ist. Die Benutzerfreundlichkeit wird natürlich auch durch die Mengenorientierung des relationalen Modells und seine Datenstruktur ("Relation ≈ Tabelle") unterstützt.

Die ersten SQL-Produkte. Von Mitte 1977 bis Ende 1979 wurde System R intern bei IBM und bei drei ausgewählten Kunden getestet. Die SQL-Benutzerschnittstelle von System R wurde dabei als mächtig und relativ leicht erlernbar eingestuft. Aufgrund dieser Bewährung im praktischen Einsatz fiel dann bei IBM die Entscheidung, ein marktreifes, auf der System-R-Technologie basierendes SQL-Produkt zu entwickeln. Das Ergebnis war SQL/DS (1981) und DB2 (1983). Das erste kommerziell verfügbare SQL-System, nämlich ORACLE, wurde aber schon 1980 von Relational Software Inc. (heute ORACLE Corporation) auf den Markt gebracht. Weitere Hersteller folgten mit SQL-Systemen und bald hatte sich SQL als De-facto-Industriestandard etabliert.

Standardisierung. SQL ist gewissermaßen auch ein De-jure Standard. Dabei kann man die folgenden Standard- und damit Sprachgenerationen unterscheiden:

- SQL-86 und SQL-89. SQL-86 wurde 1986 als ANSI Standard verabschiedet. Dieser wurde 1987 auch als (internationaler) ISO-Standard übernommen. 1989 folgte SQL-89, eine nur geringfügig erweiterte Fassung ("SQL with integrity enhancement"), wobei die Erweiterungen vor allem die Integritätsbedingungen

(`PRIMARY KEY`, `FOREIGN KEY`) betroffen haben. Auch dieser erweiterte Standard wurde von ISO übernommen. Als Überbegriff für SQL-86 und SQL-89 hat sich das Kürzel SQL1 eingebürgert.

- SQL-92. Seit 1987 arbeiten ANSI und andere nationale Standardbehörden (BSI, DIN, JISC, SCI, ...) im Rahmen von ISO koordiniert zur Weiterentwicklung des SQL-Standards zusammen. Diese Zusammenarbeit mündete 1992 in der Publikation des SQL-92 Standarddokuments ("SQL2"). SQL-92 sieht drei Sprachebenen vor: Entry, Intermediate und Full SQL, wobei die jeweils niedrigere Ebene eine Teilmenge der darüberliegenden Ebene darstellt. Entry SQL ist dabei fast völlig deckungsgleich mit SQL-89, das nur durch ganz wenige Sprachkonstrukte erweitert worden ist.
- SQL-99. Das Standarddokument dieser dritten Sprachversion ("SQL3") kam Ende 1999 heraus. SQL-99 — die offizielle Schreibweise ist SQL:1999 — hat einen für jede standardkonforme Implementierung verbindlichen Sprachkern Core SQL, der eine maßvolle Erweiterung von Entry SQL darstellt.* Darüberhinaus sind die Sprachelemente von SQL-99 sogenannten *Features* zugeordnet, wobei eine recht feine Granularität eingehalten wird. Eine Implementierung kann nun nach Belieben über Core SQL hinausgehende Features auswählen, um auf diese Weise für bestimmte Anwendungsbereiche maßgeschneiderte (standardkonforme) Produkte bereitzustellen. Dabei bietet SQL-99 als Orientierungshilfe noch sogenannte *Packages* an, durch die bestimmte zusammengehörige Features zusammengefaßt werden.
- SQL:2003. Die Standardrevision des Jahres 2003 enthält jedenfalls den gesamten Sprachumfang von SQL:1999. Obwohl der Funktionsumfang etwas erweitert worden ist (z.B. `MERGE`, `CREATE TABLE ... AS Query`, `BIGINT`, ...) und sogar ein neuer Teil (SQL/XML) hinzugekommen ist, steht doch die Konsolidierung des bestehenden Sprachumfanges im Vordergrund. SQL:2003 ist noch weitgehend unimplementiert.

Benutzerschnittstellen. SQL bietet mindestens die folgenden Schnittstellen an:

- direktes SQL. Dabei handelt es sich um eine interaktive Sprachversion, typischerweise für den Endbenutzer. Beipielsweise wird eine Abfrageanweisung eingegeben und die Ergebnisrelation† wird am Bildschirm angezeigt.
- eingebettetes SQL. Damit können SQL-Anweisungen aus einem Anwendungsprogramm, das beispielsweise in C geschrieben ist, abgesetzt werden. Diese Schnittstelle wird typischerweise vom Anwendungsprogrammierer verwendet.

Bemerkungen. Die Terminologie von SQL deckt sich nicht immer mit der des relationalen Modells. Zum Beispiel sagt man in SQL 'Tabelle' statt 'Relation', 'Spalte' statt 'Attribut' und 'Zeile' statt 'Tupel' (wahrscheinlich waren den Sprachentwicklern die Begriffe 'Relation', 'Tupel', 'Attribut' usw. zu benutzerfern) und wir werden uns im weiteren dieser Terminologie anschließen.

---

* Eine fundierte Darstellung von Core SQL findet man in: W. Panny, A. Taudes: Einführung in den Sprachkern von SQL-99, Springer-Verlag, Berlin, 2000

† In SQL sagt man 'Tabelle' statt 'Relation'.

Allerdings gibt es in SQL zwei Arten von Tabellen: Solche, die dem relationalen Modell entsprechen (Menge von Tupeln, in der SQL-Diktion *set of rows*). In diesem Fall können gleiche Zeilen nicht mehrfach vorkommen. Wir werden in diesem Fall von eigentlichen Tabellen sprechen.
Daneben gibt es auch Tabellen, in denen mehrfach vorkommende Zeilen zugelassen sind (in der SQL-Diktion heißt das *multiset of rows*). Wir werden in diesem Fall von uneigentlichen Tabellen sprechen. Diese uneigentlichen Tabellen entsprechen nicht dem relationalen Modell und erhöhen die Komplexität von SQL.

5.2 SQL-DDL: Definition eines Datenbankschemas

In diesem Abschnitt werden die wichtigsten SQL-Anweisungen zur Datendefintion ("SQL-DDL") vorgestellt. SQL spricht in diesem Zusammenhang von der *Schemasprache*. Mit Hilfe dieser Anweisungen kann ein *Datenbankschema* definiert werden. Dabei ist das Datenbankschema eine Umsetzung des konteptuellen (oder logischen) Schemas des Drei-Ebenen-Architekturkonzepts (vgl. 1.6.2) und besteht aus einer Menge von (Definitionen oder Beschreibungen von) *Basisrelationen und Integritätsbedingungen* (vgl. 3.3.1). Neben der konzeptuellen Ebene müssen auch die *externen Schemata* des Drei-Ebenen-Architekturkonzepts umgesetzt werden, was auf die Definition von virtuellen Tabellen oder *Views* hinausläuft (vgl. 3.3.2).
Einige typische Anweisungen der SQL-DDL (also der Schemasprache) sind:

Anweisung	Bedeutung
`CREATE SCHEMA`	Definition eines SQL-Schemas
`DROP SCHEMA`	Entfernen der Definition eines SQL-Schemas
`CREATE TABLE`	Definition einer Basistabelle
`ALTER TABLE`	Ändern der Definition einer Basistabelle
`DROP TABLE`	Entfernen der Definition einer Basistabelle
`CREATE VIEW`	Definition eines Views
`DROP VIEW`	Entfernen der Definition eines Views
`CREATE DOMAIN`	Definition einer Domäne
`DROP DOMAIN`	Entfernen der Definition einer Domäne

Die wichtigste davon ist zweifellos die `CREATE TABLE` - Anweisung. Zu jeder `CREATE`-Anweisung gibt es eine entsprechende `DROP`-Anweisung, mit der die Definition (der *Deskriptor* des Objekts) wieder entfernt werden kann. Für manche Objekte gibt es zusätzlich noch eine `ALTER`-Anweisung, mit Hilfe derer die Definition geändert werden kann. Ein SQL-Schema kann als eine Art Behälter aufgefaßt werden, der die Definitionen aller Basistabellen, Views und weiterer Objekte enthält, die einem Benutzer gehören. Jeder Benutzer hat ein SQL-Schema. Um ein solches SQL-Schema definieren zu können, braucht man ganz besondere Berechtigungen, die

üblicherweise nur der Datenbankadministrator hat. Wir gehen hier daher nicht näher auf die dazugehörigen Anweisungen ein.

Zur Spezifikation der Syntax verwendet der SQL-Standard eine erweiterte Version der BNF ("Backus Normal Form" oder "Backus Naur Form"). Auch wir werden in diesem Kapitel zur Spezifikation der Produktionsregeln eine etwas abgewandelte Version dieser Notation verwenden. Betrachten wir als Beispiel die folgenden Produktionsregel:

column-definition ::=
  column-name {data-type | domain-name} [column-constraint ...]
column-constraint ::=
  `NOT NULL` | `UNIQUE` | `PRIMARY KEY` | reference-constraint
reference-constraint ::=
  `REFERENCES` table-name [(reference-column)]

Auf der linken Seite steht immer eine *Variable*. Dann kommt der *Definitionsoperator* ::= . Auf der rechten Seite können Variable, *Endsymbole* (auch *Terminalsymbole* genannt) sowie gewisse *Metazeichen* vorkommen. Beispiele für Variable sind: column-definition, column-name oder data-type. Variable werden hier also kleingeschrieben und durch den Schrifttyp (variable) kenntlich gemacht. Beispiele für Endsymbole sind: `NOT NULL`, `UNIQUE`, aber auch `(` und `)`. Für Variable auf der rechten Seite müssen entsprechende Produktionsregeln angewendet werden, bis nur mehr eine Folge von Endsymbolen übrigbleibt, welche das aus der urspünglichen Variablen resultierende (oder aus ihr abgelitete) Sprachkonstrukt darstellt.

Die Metazeichen sind (neben dem schon oben besprochenen Definitionsoperator): |, { }, [ ] und ... sowie !!

Durch | werden Alternativen voneinander getrennt. Ein column-constraint kann zum Beispiel `NOT NULL` oder `UNIQUE` oder `PRIMARY KEY` oder ein reference-constraint sein.

Durch { } können syntaktische Elemente zusammengefaßt werden. Zum Beispiel muß in einer column-definition auf den column-name ein data-type oder ein domain-name folgen.

Das durch [ ] geklammerte Element ist optional. Beispielsweise ist (reference-column) in einem reference-constraint optional.

Das unmittelbar vor ... stehende Element kann beliebig oft wiederholt werden. Beispielsweise steht A ... für A oder AA oder AAA und so weiter und so fort.

Außerdem leitet der Standard eine verbale Spezifikation oder einen Kommentar auf der rechten Seite durch !! ein.

5.2.1 Definition von Basistabellen

Eine Basistabelle wird durch `CREATE TABLE` bzw. eine table-definition definiert. Dadurch wird ein Deskriptor der Basistabelle angelegt. Die Syntax einer etwas vereinfachten Version wird durch die folgende BNF spezifiziert:

table-definition ::=
  CREATE TABLE table-name
  (column-definition [{, column-definition}...] [{, table-constraint}...]);

Nach dem Namen der zu definierenden Tabelle kommt eine öffnende Klammer, dann kommt mindestens eine *Spaltendefinition*. Nach den Spaltendefinitionen können noch *Tabellenbedingungen* definiert werden.* Es muß aber keine Tabellenbedingungen geben. Schließlich kommt noch die schließende Klammer und der Strichpunkt. Jede SQL-Anweisung muß durch einen Strichpunkt abgeschlossen werden.

column-definition ::=
  column-name {data-type | domain-name} [column-constraint ...]

Eine *Spaltendefinition* beginnt mit dem Namen der zu definierenden Spalte, welcher der Datentyp oder die Domäne der Spalte folgen muß. Schließlich können noch eine oder mehrere Spaltenbedingungen kommen.

Wir haben die folgenden Datentypen (die wichtigsten sind der *numerische* und der *Zeichenketten-Typ*):

data-type ::=
    char-string-type
  | bit-string-type
  | num-type
  | datetime-type
  | interval-type

char-string-type ::=
    CHARACTER [(length)]
  | CHAR   [(length)]
  | CHARACTER VARYING (length)
  | CHAR VARYING   (length)
  | VARCHAR (length)

CHAR ist eine Kurzform von CHARACTER, ebenso ist VARCHAR eine Kurzform von CHARACTER VARYING bzw. CHAR VARYING. Die ersten beiden Alternativen und die letzten drei Alternativen haben also die gleiche Bedeutung. Durch die ersten beiden Alternativen wird eine Zeichenkette mit fixer Länge length definiert. Wenn die length-Angabe fehlt, wird length = 1 angenommen, es handelt sich dann also um ein einzelnes Zeichen.

Durch die letzten drei Alternativen wird eine Zeichenkette mit variabler Länge definiert. Der (hier zwingende) length-Wert gibt die maximale Länge der entsprechenden Zeichenkette an.

bit-string-type ::=
    BIT [(length)]
  | BIT VARYING (length)

* Es ist sogar erlaubt, Spaltendefinitionen und Tabellenbedingungen beliebig zu mischen. Es ist aber ratsam, sich an die obige Reihenfolge zu halten.

Dadurch werden Bitketten fixer bzw. variabler Länge definiert. Sonst ist alles analog zu den Zeichenketten geregelt.

num-type ::=
    exact-num-type
  | approximate-num-type

Beim numerischen Typ gibt es die Untertypen exakt-numerisch, was in der üblichen Terminologie *Festkomma-Typ* genannt wird, und approximativ-numerisch, was dem *Gleitkommatyp* entspricht.

exact-num-type ::=
    NUMERIC [(precision [,scale])]
  | DECIMAL [(precision [,scale])]
  | DEC   [(precision [,scale])]
  | INTEGER
  | INT
  | SMALLINT

Beim Festkomma-Typ sind `DEC` bzw. `INT` Kurzformen von `DECIMAL` bzw. `INTEGER`, so daß tatsächlich nur drei Möglichkeiten übrigbleiben. `INTEGER` und `SMALLINT` spezifizieren ganze Zahlen. Bezüglich der Wertebereiche legt der Standard nur fest, daß der Wertebereich von `INTEGER` mindestens so groß sein muß, wie der von `SMALLINT`.

Bei `NUMERIC` und `DECIMAL` kann man die gewünschte Anzahl der dezimalen Stellen durch den precision-Wert bestimmen. Der optionale scale-Wert gibt allfällige Nachkommastellen an (scale $\leq$ precision). Wenn scale weggelassen wird, handelt es sich um ganze Zahlen. Wenn auch die precision-Angabe fehlt, handelt es sich um ganze Zahlen mit implementationsdefinierter Stellenanzahl. Der Unterschied zwischen `NUMERIC` und `DECIMAL` besteht darin, daß die SQL-Implementierung bei `DECIMAL` auch eine größere Stellenanzahl vorsehen kann als der precision-Wert angibt, während sie bei `NUMERIC` vollständig an den precision-Wert gebunden ist.

approximate-num-type ::=
    FLOAT [(precision)]
  | REAL
  | DOUBLE PRECISION

Hier wird für die interne Darstellung der Zahlen ein Gleitkommaformat verwendet. Der precision-Wert bei `FLOAT` gibt die geforderte binäre Genauigkeit an, also die gewünschte Stellenanzahl der binären Mantisse. Wenn precision weggelassen wird, wird ein implementationsdefinierter Defaultwert genommmen.

Bei `REAL` und `DOUBLE PRECISION` werden implementationsdefinierte interne Gleitkommadarstellungen genommen. Die Genauigkeit muß aber bei `DOUBLE PRECISION` größer sein als bei `REAL`.

Ab SQL-92 gibt es auch Datentypen für zeitbezogene Daten, nämlich den `DATETIME`- und den `INTERVAL`-Typ.

datetime-type ::=
    DATE
  | TIME
  | TIMESTAMP

Ein DATETIME-Wert gibt einen *Zeitpunkt* an, wobei es sich um eine Datum (DATE), eine Tageszeit oder ein Datum zusammen mit einer Tageszeit (TIMESTAMP) handeln kann.

interval-type ::=
  INTERVAL interval-qualifier

Ein INTERVAL-Wert repräsentiert einen *Zeitraum*, also eine zeitliche Dauer.

interval-qualifier ::=
    start-field TO end-field
  | single-datetime-field

Bei der ersten Alternative für den interval-qualifier wird ein start-field und ein end-field angegeben, z.B. HOUR TO SECOND oder YEAR TO MONTH. Bei der zweiten Alternative wird nur ein einziges datetime-field spezifiziert.

start-field ::= datetime-field

end-field ::= datetime-field

single-datetime-field ::= datetime-field

Folgende datetime-fields sind vorgesehen:

datetime-field ::=
  YEAR | MONTH | DAY | HOUR | MINUTE | SECOND

Wenn start-field TO end-field gewählt wird, muß start-field eine größere Einheit als end-field spezifizieren. Wenn das start-field YEAR ist, darf das end-field nur MONTH sein. MONTH ist als start-field verboten. Diese beiden Einschränkungen sind notwendig, weil das in die Einheit von end-field umgerechnete Zeitintervall sonst nicht eindeutig wäre.

Durch ein column-constraint (*Spaltenbedingung*) wird eine an eine einzelne Spalte gebundene Integritätsbedingung definiert. Tatsächlich stellt eine Spaltenbedingung die Kurzform einer entsprechenden table-constraint (*Tabellenbedingung*) für den Fall dar, daß nur eine einzige Spalte involviert ist.

column-constraint ::=
    PRIMARY KEY
  | reference-constraint
  | UNIQUE
  | NOT NULL
  | check-constraint

Durch PRIMARY KEY wird festgelegt, daß es sich bei der Spalte um den (einelementigen) Primärschlüssel der Tabelle handelt. Durch UNIQUE wird festgelegt,

daß kein in der Spalte auftretende Wert mehrfach vorkommen darf. Durch `NOT NULL` können für die Spalte `NULL-Werte` ausgeschlossen werden. Durch ein check-constraint (`CHECK`-Bedingung) kann für die entsprechende Spalte eine allgemeine, auf eine Suchbedingung hinauslaufende Integritätsbedingung definiert werden. Wenn die Suchbedingung für eine Zeile den Wahrheitswert *falsch* liefert, dann wird die `CHECK`-Bedingung von dieser Zeile verletzt (und die die Verletzung verursachende Anweisung — es kann sich dabei nur um eine Mutationsanweisung handeln — wird zurückgewiesen).

check-constraint ::=
  `CHECK` (search-condition)

Schließlich wird durch ein reference-constraint festgelegt, daß es sich bei der entsprechenden Spalte um einen (einelementigen) Fremdschlüssel handelt.

reference-constraint ::=
  `REFERENCES` table-name [(reference-column)]

reference-column ::= column-name

Der Tabellenname gibt natürlich den Namen der referenzierten Tabelle an. Die optionale reference-column gibt den Namen der referenzierten Spalte an. Gemäß dem relationalen Modell muß es sich dabei um den Primärschlüsssel der referenzierten Tabelle handeln, in welchem Fall die Spezifikation der reference-column eine Fleißaufgabe ist und daher auch entfallen kann. In SQL ist es aber auch erlaubt, daß die referenzierte Spalte nur eine `UNIQUE`-Spalte ist und in diesem Fall muß die reference-column angegeben werden.

table-constraint ::=
    `PRIMARY KEY` (column-list)
  | `FOREIGN KEY` (referencing-columns) `REFERENCES` table-name [(column-list)]
  | `UNIQUE` (column-list)
  | check-constraint

Das table-constraint (*Tabellenbedingung*) ist die allgemeinere Form der Definition einer Integritätsbedingung. Jede Spaltenbedingung kann äquivalentwerweise auch als Tabellenbedingung definiert werden. Die Form der Tabellenbedingung ist aber dann zwingend, wenn mehrer Spalten in die Bedingung involviert sind, wenn also der Primärschlüssel, der Fremdschlüssel oder die `UNIQUE`-Bedingung mehrere Spalten umfaßt oder die `CHECK`-Bedingung Spaltenreferenzen auf mehr als eine Spalte enthält. Ansonsten ist alles analog zur Spaltenbedingung geregelt.†

column-list ::=
  column-name [{, column-name}...]

referencing-columns ::= column-list

† Bezüglich der `NOT NULL` - Spaltenbedingung gilt, daß der Standard diese als Kurzform von `CHECK(column-name IS NOT NULL)` definiert. `NOT NULL` ist dabei die negierte Form des `NULL`-Prädikats (s.u.).

```
domain-definition ::=
  CREATE DOMAIN domain-name [AS] data-type [check-constraint];
```

Mit Hilfe der `CREATE DOMAIN` - Anweisung kann eine Domäne definiert werden, was in SQL auf einen Datentyp mit einer optionalen `CHECK`-Bedingung hinausläuft. In der Suchbedingung einer solchen `CHECK`-Bedingung wird der Domänenwert durch das Wortsymbol `VALUE` angesprochen. Spaltenreferenzen dürfen keine vorkommen.
Beispiele:
```
CREATE DOMAIN Gewicht AS DECIMAL(8,2) CHECK ( VALUE >= 0 );
CREATE DOMAIN ProzentAngabe AS REAL
        CHECK ( VALUE >= 0 AND VALUE <= 100.0 );
```

Wenden wir uns nun der search-condition (*Suchbedingung*) zu. Im Augenblick brauchen wir sie, weil eine `CHECK`-Bedingung im wesentlichen aus einer Suchbedingung besteht.† Diese ist ein sogenannter Boolscher oder logischer Ausdruck. Ein solcher liefert einen Wahrheitswert und dabei können mittels der boolschen Operatoren (logischen Junktoren) `OR`, `AND` und `NOT` auch Wahrheitswerte verknüpft werden.

```
search-condition ::=
    boolean-term
  | search-condition OR boolean-term

boolean-term ::=
    boolean-factor
  | boolean-term AND boolean-factor

boolean-factor ::=
  [NOT] boolean-primary
```

Die boolschen Operatoren `OR`, `AND` und `NOT` haben dabei die üblichen Prioritäten: `NOT` hat die höchste Priorität, dann kommt `AND`, die niedrigste Priorität hat `OR`. Innerhalb der gleichen Prioritätsstufe wird von links nach rechts ausgewertet. Um eine andere Auswertungsreihenfolge festzulegen, können zusätzlich Klammern gesetzt werden. Die Suchbedingung entspricht im wesentlichen der *Selektionsbedingung* unserer relationenalgebraischen Sprache (vgl. 3.2.2).

```
boolean-primary ::=
  predicate | ( search-condition )
```

Die in einer Suchbedingung auftretenden Wahtheitswerte müssen ja irgendwo herkommen und die "ultimativen Wahrheitswertlieferanten" sind die sogenannten *Prädikate* (predicates). Die (wichtigsten) SQL-Prädikate sind:

† Das typische Vorkommnis der Suchbedingung ist aber das in der `FROM`-Klausel der `SELECT`-Abfrage (s.u.).

predicate ::=
comparison-predicate
| between-predicate
| like-predicate
| null-predicate
| in-predicate
| exists-predicate

Die einzelnen Prädikate werden erst im Rahmen von 5.3 besprochen.

5.2.3 Ein Beispiel für ein Datenbankschema

```
CREATE DOMAIN PersonalNr AS INTEGER;
CREATE DOMAIN ProjektNr  AS INTEGER;
CREATE DOMAIN ProzentAngabe AS REAL
       CHECK (VALUE>=0 AND VALUE<=100.0);

CREATE TABLE angestellte
     ( ANG_NR  PersonalNr PRIMARY KEY,
       NAME    VARCHAR(30),
       WOHNORT VARCHAR(30),
       ABT_NR  INTEGER );

CREATE TABLE projekt
     ( P_NR      ProjektNr PRIMARY KEY,
       P_NAME    VARCHAR(15),
       P_FILIALE VARCHAR(30),
       P_LEITER  PersonalNr REFERENCES angestellte );

CREATE TABLE ang_pro
     ( P_NR          ProjektNr  REFERENCES projekt,
       ANG_NR        PersonalNr REFERENCES angestellte,
       PROZ_ARBZEIT  ProzentAngabe,
       PRIMARY KEY (P_NR, ANG_NR) );
```

5.2.4 Definition von Sichten

Eine virtuelle Tabelle bzw. ein View wird durch `CREATE VIEW` bzw. eine view-definition definiert. Die Syntax wird durch die folgende Produktionsregel spezifiziert:

view-definition ::=
`CREATE VIEW` table-name
[(column-name {, column-name}...)]
`AS` query-expression
[`WITH CHECK OPTION`];

Auf `CREATE VIEW` folgt der Tabellenname der zu definierenden virtuellen Tabelle. Man beachte, daß Basis- und virtuelle Tabellen in SQL einen gemeinsamen Namensraum haben. Darauf folgt eine optionale Spaltenliste. Mit Hilfe dieser Spaltenliste kann man die Spalten des Views benennen bzw. umbenennen. Wenn eine solche Spaltenliste angegeben wird, dann muß für jede Spalte der Ergebnistabelle des *Abfrageausdrucks* (query-expression) ein Spaltenname vorgesehen werden.

Der Abfrageausdruck stellt die *Viewformel* dar und gibt an, wie der View aus der (den) im Abfrageausdruck enthaltenen Tabelle(n) abzuleiten ist. Normalerweise haben die Spalten der Ergebnistabelle einen Spaltennamen und wenn die `CREATE VIEW` - Anweisung keine Spaltenliste aufweist, übernehmen die Spalten des Views diese Spaltennamen. Es kann aber manchmal vorkommen, daß eine oder mehrere Spalten der Ergebnistabelle der Viewformel keinen Namen haben. In diesem Fall *muß* eine Spaltenliste angegeben werden, um den sonst namenlosen Spalten einen Namen zu geben. Auch wenn alle Spalten der Ergebnistabelle der Viewformel einen Namen haben, bleibt es dem Benutzer unbenommen, die Spalten durch Spezifikation einer Spaltenliste umzubenennen. Auf Syntax und Semantik des Abfrageausdrucks wird im Rahmen von 5.3 eingegangen.

Die `CHECK`-Option hat damit zu tun, ob gewisse Mutationen (auf einen mutierbaren View) zugelassen werden oder nicht und wir werden darauf bei der Besprechung der Mutationsanweisungen von SQL in 5.4 eingehen.

Beispiel:
```
CREATE VIEW angestellteAusKarlsruhe AS
  SELECT ANG_NR, NAME, WOHNORT, ABT_NR
  FROM   angestellte
  WHERE  WOHNORT = 'Karlsruhe';
```

5.2.5 Grundbausteine der SQL-Sprache

In diesem Abschnitt wird auf einige Sprachelemente eingegangen, die zu den Grundbausteinen der SQL-Sprache gehören und bisher noch nicht behandelt worden sind. Dieser Abschnitt wurde eingefügt, um die Spezifikation einigermaßen vollständig zu halten. Um die Sache nicht ausufern zu lassen, werden wir dabei nicht immer bis zu den BNFs auf den untersten Ebenen gehen, sondern uns auch mit Beispielen oder verbalen Spezifikationen zufrieden geben.

Beginnen wir einmal mit den folgenden Namen:

table-name ::=
  [schema-name.] identifier

schema-name ::= identifier

Eine *Tabellenname* ist in der Regel ein einfacher Bezeichner. Wahlweise kann man den Tabellennamen auch mit dem *Schemanamen* des Besitzers der Tabelle qualifizieren (der normalerweise mit der Benutzerkennung des Besitzers identisch ist). Wenn man mit seinen eigenen Tabellen arbeitet, genügt der unqualifizierte Tabellenname, weil defaultmäßig der Schemaname des Besitzers als angenommen wird.

domain-name ::=
  [schema-name.] identifier

Für *Domänennamen* ist alles völlig analog zu den Tabellennamen geregelt.

column-name ::= identifier

Ein *Spaltenname* ist ein (einfacher) Bezeichner.

identifier ::=
    regular-identifier
  | delimited-identifier

Ab SQL2 gibt es zwei Arten von *Bezeichnern* (identifier), nämlich *reguläre Bezeichner* (regular-identifier) und *begrenzte Bezeichner* (delimited-identifier). Zum Sprachkern gehören nur die regulären Bezeichner, auf die wir uns daher hier beschränken wollen. Diese müssen mit einem Buchstaben beginnen, dem eine beliebige Folge aus Buchstaben, Ziffern und dem Underscore-Symbol '_' folgen kann. Es gilt das Prinzip der "Unerheblichkeit der Kleinschreibung": Kleinbuchstaben werden gewissermaßen in die entsprechenden Großbuchstaben übersetzt. Eine standardkonforme SQL-Implementierung muß für reguläre Bezeichner eine Länge von mindestens 18 Zeichen zulassen. Die folgenden Bezeichner sind somit gültige reguläre Bezeichner (die Bezeichner in der ersten Spalte sind überdies äquivalent):

```
GEWICHT     ABC789     A
Gewicht     ABT_NR     REGULAR_IDENTIFIER
gewicht     x__1       ang_pro
```

Für jeden Datentyp kann man Konstante dieses Typs verwenden. SQL verwendet für Konstante die Bezeichnung *Literal* (literal).

literal ::=
    num-literal
  | char-string-literal
  | bit-string-literal
  | datetime-literal
  | interval-literal

num-literal ::=
    exact-num-literal
  | approximate-num-literal

Die folgenden Beispiele zeigen einige gültige *Festkommaliterale* (exact-num-literals):

```
  0      50      314159
  1      50.     0.12345
100      50.0     .12345
```

Und hier sind ein paar Beispiele für *Gleitkommaliterale* (approximate-num-literals):

```
 0.314E1      2.998E5
 3.140E0      9.460E12
31.400E-1     1.661E-24
```

Die Gleitkommaliterale in der ersten Spalte sind alle äquivalent und haben den Wert 3.14. Die Beispiele in der zweiten Spalte zeigen, daß die Gleitkommanotation in erster Linie zur Darstellung von Zahlen gedacht ist, deren Absolutbetrag entweder sehr groß (9.46×10^{12}) oder sehr klein ist (1.661×10^{-24}).

Zeichenketten-Literale werden auf beiden Seiten durch ein einfaches Hochkomma (Quote-Symbol) ‘’’ begrenzt. Wenn das Zeichenketten-Literal ein Quote-Symbol enthalten soll, so sind dafür zwei Quote-Symbole unmittelbar nebeneinander zu setzen. Einige Beispiele:

```
'Oesterreich'       '+ ist ein SQL-Sonderzeichen'
'Wie geht''s?'      'DON''T'
'12.331'            '$%&*_'
```

Die folgenden Literale sind gültige Bitketten-Literale (bit-string-literals):

```
B'0'
B'1'
B''
B'001100'
B'00001111'
B'00101010000101010101000010101000001010100001010101100001010100001'
```

Beim `DATETIME`-Literal (datetime-literal) gibt es natürlich für jeden `DATETIME`-Untertyp entsprechende Literale.

datetime-literal ::=
- date-literal
- time-literal
- timestamp-literal

Ein `DATE`-Literal spezifiziert ein *Datum*. Für ein `DATE`-Literal muß das Format `DATE 'yyyy-mm-dd'` eingehalten werden. Es muß sich dabei um ein gültiges (gregorianisches) Datum zwischen dem 1. Jänner des Jahres 1 und dem 31. Dezember des Jahres 9999 handeln. Die folgenden Literale sind gültige `DATE`-Literale:

```
DATE '1999-12-31'
DATE '2006-02-28'
DATE '2013-09-30'
DATE '2008-02-29'
```

Ein `TIME`-Literal spezifiziert eine gültige *Tageszeit*. Für ein `TIME`-Literal muß das Format `TIME 'hh:mm:ss'` eingehalten werden. Die folgenden Beispiele zeigen gültige `TIME`-Literale:

```
TIME '11:30:00'
TIME '09:20:59'
TIME '17:02:25'
TIME '00:00:00'
TIME '23:59:59'
```

Schließlich gibt es noch TIMESTAMP-Literale, die eine Kombination eines DATE- mit einem TIME-Literal darstellen, wodurch eine *Datumsangabe* mit einer gültigen *Zeit* versehen werden kann. Wie die folgenden Beispiele zeigen, beginnt ein TIMESTAMP-Literal mit dem Datum, dann kommt die Zeitangabe. Zwischen der Datums- und der Zeitkomponente muß genau ein Leerzeichen stehen:

```
TIMESTAMP '2006-05-30 16:10:30'
TIMESTAMP '2006-12-31 23:59:59'
```

In einem TIMESTAMP-Literal kann der Sekundenwert auch schon in Core SQL bis zu 6 Nachkommastellen haben:

```
TIMESTAMP '2006-05-30 16:10:30.00'
TIMESTAMP '2006-12-31 23:59:59.999999'
```

Während ein DATETIME-Wert einen *Zeitpunkt* charakterisiert, repräsentiert ein INTERVAL-Wert einen *Zeitraum*. Beispielsweise liegt zwischen dem Zeitpunkt 10:30:00 und dem Zeitpunkt 9:20:10 ein Zeitraum von 1 Stunde, 9 Minuten und 50 Sekunden. Dieser Zeitraum kann natürlich ebenso durch 69 Minuten und 50 Sekunden angeben werden. Zwischen den Zeitpunkten 30. April 1994 und 1. Februar 1994 liegt ein Zeitraum von 88 Tagen. Wenn man andererseits zum Zeitpunkt 1. Februar 1994 einen Zeitraum von 1 Jahr und 3 Monaten hinzufügt, erhält man den Zeitpunkt 1. Mai 1995. Um solche Zeiträume — oder eben Zeitintervalle — angeben zu können, sind die INTERVAL-Literale vorgesehen, bei denen man zweckmäßigerweise YEAR-MONTH - Literale und DAY-TIME - Literale unterscheidet.

Die ersten drei der oben gegebenen Beispiele laufen auf DAY-TIME - Literale hinaus, nämlich auf: INTERVAL '1:9:50' HOUR TO SECOND bzw. INTERVAL '69:50' MINUTE TO SECOND bzw. INTERVAL '88' DAY. Das letzte Beispiel läuft auf das YEAR-MONTH - Literal INTERVAL '1-3' YEAR TO MONTH hinaus. Die Beispiele weisen schon darauf hin, warum zwischen DAY-TIME - und YEAR-MONTH - Intervallen unterschieden werden muß: Man kann *Monate* nicht ohne weiteres in *Tage* bzw. kleinere Einheiten (*Stunden, Minuten, Sekunden*) umrechnen. Haben beispielsweise 2 Monate 59, 60, 61 oder 62 Tage? Daher laufen die beiden Spielarten letzten Endes darauf hinaus, daß in keinem INTERVAL gleichzeitig ein Monats- und ein Tageswert vorkommen darf.

Ein YEAR-MONTH - Literal enthält entweder nur einen *Jahreswert* oder nur einen *Monatswert* oder beides. Die folgenden Beispiele decken alle Möglichkeiten ab:

```
INTERVAL '1'    YEAR
INTERVAL '3'             MONTH
TIMESTAM '1-9'  YEAR TO MONTH
```

Das INTERVAL-Literal kann wahlweise auch mit einem Vorzeichen versehen werden. Es sind also insbesondere auch *negative* INTERVAL-Literale möglich:

```
INTERVAL -'1'    YEAR
INTERVAL +'3'             MONTH
INTERVAL -'1-9'  YEAR TO MONTH
```

Der erste (und möglicherweise auch einzige) Wert im `INTERVAL`-Literal ist im Prinzip unbeschränkt. Wenn ein Jahreswert *und* ein Monatswert angegeben sind (wie in Zeile 3), dann muß zwischen den beiden Werten — wie von den `DATETIME`-Literalen gewohnt — ein Bindestrich '`-`' stehen, wobei der Wertebereich für die Monatsangabe `0 – 11` ist.

Demgegenüber bezieht sich ein `DAY-TIME` - Literal auf eine zusammenhängende Teilfolge aus `DAY`, `HOUR`, `MINUTE`, `SECOND`. Die folgenden Beispiele decken im Prinzip alle Möglichkeiten ab:

```
INTERVAL   '2 23:59:59.123456'   DAY TO SECOND
INTERVAL   '2 23:59'             DAY TO MINUTE
INTERVAL   '1000 1'              DAY TO HOUR
INTERVAL   '80'                  DAY
INTERVAL   '23:59:59.123456'     HOUR TO SECOND
INTERVAL   '101:59'              HOUR TO MINUTE
INTERVAL   '72'                  HOUR
INTERVAL   '3:20'                MINUTE TO SECOND
INTERVAL   '15'                  MINUTE
INTERVAL   '9.87'                SECOND
```

Für jeden Datentyp gibt es (je nach dem Datentyp mehr oder weniger komplexe) Wertausdrücke (value-expressions).

```
value-expression ::=
    num-value-expression
  | char-value-expression
  | bit-value-expression
  | datetime-value-expression
  | interval-value-expression
```

Ein *numerischer Wertausdruck* (num-value-expression) repräsentiert einen numerischen Wert. Wie die folgenden regeln zeigen, kann man in einem numerischen Wertausdruck die gewohnten arithmetischen Operatoren verwenden:

```
num-value-expression ::=
    term
  | num-value-expression + term
  | num-value-expression - term

term ::=
    factor
  | term * factor
  | term / factor
```

Durch monadische Verwendung des '`-`'-Operators kann man das Vorzeichen des darauffolgenden *numerischen Primary* umkehren. Aus Symmetriegründen kann auch '`-`' monadisch verwendet werden, was aber keine Wirkung hat. Dabei stellen die

numerischen Primaries sozusagen die ultimativen Wertelieferanten der numerischen Wertausdrücke dar.

```
factor ::=
  [+ | -] num-primary
```

Die einzige auf Zeichenketten anwendbare Operation ist die *Verkettung* (concatenation). Der Verkettungsoperator wird in SQL durch '||' symbolisiert.

```
char-value-expression ::=
    char-factor
  | concatenation

concatenation ::=
  char-value-expression || char-factor

char-factor ::= char-primary
```

Auch bei Bitketten gibt es nur einen Verkettungsoperator '||'.

```
bit-value-expression ::=
    bit-factor
  | bit-concatenation

bit-concatenation ::=
  bit-value-expression || bit-factor

bit-factor ::= bit-primary
```

Es würde uns hier zu weit führen auf die Details der Arithmetik von DATETIME- und INTERVAL-Wertausdrücken (datetime-value- und interval-value-expressions) einzugehen. Aber die Grundidee ist klar: Man kann zu einem Zeitpunkt ein Zeitintervall addieren oder subtrahieren und das Ergebnis ist wieder ein Zeitpunkt (beispielsweise 2007-01-10 + 3 Tage ergibt 2007-01-13 oder 11:30:00 + 130 Sekunden ergibt 11:32:10). Andererseits ergibt die Summe oder Differenz zweier Intervalle wieder ein Intervall (z.B. 3 Stunden + 4 Stunden ergibt 7 Stunden). Ein Intervall erhält man aber auch als Differenz zweier Zeitpunkte (z.B. 2007-01-13 - 2007-01-10 ergibt 3 Tage). Auch bei den DATETIME- bzw. INTERVAL-Wertausdrücken gelangt man schließlich zu entsprechenden Primaries (datetime-primary bzw. interval-primary) als ultimativen Wertelieferanten.

Alle Primaries sind entweder ein *primärer Wertausdruck* (value-expression-primary) oder das Ergebnis einer Funktion des entsprechenden Typs (ein INTERVAL-Primary kann aber nur ein primärer Wertausdruck sein):

```
num-primary ::=
    value-expression-primary
  | num-value-function

char-primary ::=
    value-expression-primary
  | char-value-function
```

bit-primary ::=
    value-expression-primary
  | bit-value-function

datetime-primary ::=
    value-expression-primary
  | datetime-value-function

interval-primary ::= value-expression-primary

Ein *primärer Wertausdruck* ist typischerweise ein Literal, eine Spaltenreferenz oder ein geklammerter Wertausdruck. Es kann sich aber auch um eine skalare Unterabfrage oder eine Aggregatfunktion (vgl. 5.3.4) handeln. In jedem Fall muß natürlich der Datentyp zum Datentyp des entsprechenden Primaries passen.

value-expression-primary ::=
    literal
  | column-ref
  | aggregate-funct
  | scalar-subquery
  | (value-expression)

Eine *Spaltenreferenz* kann einfach oder qualifiziert sein. Der Qualifizierer kann dabei der dazugehörige Tabellenname oder ein sogenannter Korrelationsname (vgl. ??) sein. Häufig sagt man statt Korrelationssname auch *Aliasname*.

column-ref ::=
  [qualifier.]column-name

qualifier ::=
    table-name
  | correlation-name

Eine *skalare Unterabfrage* ist ein Unterabfrage, deren Ergebnistabelle nur aus einer einzigen Zeile und Spalte besteht (so daß man sie mit einem skalaren Wert gleichsetzen kann). Auf Unterabfragen wird in 5.3.8 eingegangen werden.

scalar-subquery ::= subquery

5.3 SQL-DML: Abfragen in SQL (Query Language)

angestellte $(\# = n)$

ANG_NR	NAME	WOHNORT	ABT_NR
3115	Meyer	Karlsruhe	35
3207	Müller	Mannheim	30
2814	Klein	Mannheim	32
3190	Maus	Karlsruhe	30
2314	Groß	Karlsruhe	35
1324	Schmitt	Heidelberg	35
1435	Mayerlein	Bruchsal	32
2412	Müller	Karlsruhe	32
2244	Schulz	Bruchsal	31
1237	Krämer	Ludwigshafen	31
3425	Meier	Pforzheim	30
2454	Schuster	Worms	31

projekt $(\# = k)$

P_NR	P_NAME	P_FILIALE	P_LEITER
761235	P-1	Karlsruhe	3115
770008	P-2	Karlsruhe	3115
770114	P-3	Heidelberg	1324
770231	P-4	Mannheim	2814

ang˙pro $(\# = m)$

P_NR	ANG_NR	PROZ_ARBZEIT
761235	3207	100
761235	3115	50
761235	3190	50
761235	1435	40
761235	3425	50
770008	2244	20
770008	1237	40
770008	2814	70
770008	2454	40
770114	2814	30
770114	1435	60
770114	1237	60
770114	2454	60
770114	3425	50
770114	2412	100
770231	3190	50
770231	2314	100
770231	2244	80
770231	3115	50
770231	1324	100

5.3.1 Die Syntax der SELECT-Abfrage

Das wichtigste SQL-Konstrukt zur Formulierung von Abfragen stellt die sogenannte SELECT-Abfrage (query-specification) dar. Die SELECT-Abfrage ist folgendermaßen aufgebaut:

```
query-specification ::=
  SELECT [DISTINCT | ALL] select-list
    from-clause
  [where-clause]
  [group-by-clause]
  [having-clause]
```

Nach dem Schlüsselwort SELECT kann DISTINCT oder ALL angegeben werden. Der Default ist ALL. Wenn also keine Angabe gemacht wird, ist ALL implizit. Bei DISTINCT ist sichergestellt, daß die Ergebnistabelle der SELECT-Abfrage keine mehrfachen Zeilen enthält ("eigentliche Tabellen"). Bei ALL können Duplikate auftreten ("uneigentliche Tabellen"), was eigentlich nicht dem Relationenmodell entspricht.

```
select-list ::=
    select-sublist [{, select-sublist}...]
  | *

select-sublist ::=
    derived-column | qualifier.*
```

Die SELECT-Liste (select-list) besteht typischerweise aus einer oder mehreren *Ergebnisspalten* (derived-columns). Es kann aber auch '*' spezifiziert werden (vgl. 5.3.3). Eine weitere Möglichkeit für select-sublist ist ein *qualifizierter Stern* (vgl. 5.3.6).

```
derived-column ::=
  value-expression [as-clause]
```

Die *Ergebnisspalten* der SELECT-Liste sind *Wertausdrücke* (value-expressions, vgl. 5.5.2), die optional mit einer AS-Klausel (as-clause) versehen sein können. Typischerweise handelt es sich dabei um Spaltenreferenzen (column-refs, vgl. 5.5.2).

```
as-clause ::=
  [AS] column-name
```

Mit Hilfe der optionalen AS-Klausel können die Ergebnisspalten benannt bzw. umbenannt werden (*Spaltenname* (column-name) wurde in 5.5.2 definiert).

```
from-clause ::=
  FROM table-ref [{, table-ref}...]

table-ref ::=
    table-primary
  | joined-table

table-primary ::=
    table-name [[AS] correlation-name]
  | ( joined-table )
```

correlation-name ::= identifier

where-clause ::=
  WHERE search-condition

group-by-clause ::=
  GROUP BY column-ref [{, column-ref}...]

having-clause ::=
  HAVING search-condition

!!HIER!!

---

direct-select-statement-multiple-rows ::=
  query-expression [order-by-clause];

order-by-clause ::=
  ORDER BY sort-spec [{, sort-spec}...]

sort-spec ::=
  column-ref [ASC | DESC]

query-expression ::=
    non-join-query-expression
  | joined-table

non-join-query-expression ::=
    query-specification
  | union-expression
  | except-expression
  | intersect-expression
  | (non-join-query-expression)

union-expression ::=
  query-expression UNION [DISTINCT | ALL] query-term

except-expression ::=
  query-expression EXCEPT [DISTINCT | ALL] query-term

intersect-expression ::=
  query-term INTERSECT [DISTINCT | ALL] query-primary

query-term ::=
    intersect-expression
  | query-primary

query-primary ::=
    query-specification
  | joined-table
  | (non-join-query-expression)

joined-table ::=
    cross-join
  | natural-join
  | qualified-join

cross-join ::=
  table-ref `CROSS JOIN` table-primary

natural-join ::=
    natural-inner-join
  | natural-outer-join

qualified-join ::=
    qualified-inner-join
  | qualified-outer-join

natural-inner-join ::=
  table-ref `NATURAL [INNER] JOIN` table-primary

qualified-inner-join ::=
  table-ref `[INNER] JOIN` table-ref join-specification

join-specification ::=
    `ON` join-condition
  | `USING` (join-column-list)

join-condition ::= search-condition

join-column-list ::= column-list

natural-outer-join ::=
  table-ref `NATURAL` { `LEFT` | `RIGHT` | `FULL` } `[OUTER] JOIN` table-primary

qualified-outer-join ::=
  table-ref { `LEFT` | `RIGHT` | `FULL` } `[OUTER] JOIN` table-ref join-specification

5.3.2 SELECT-Abfragen mit Suchbedingungen

Das wichtigste SQL-Konstrukt zur Formulierung von Abfragen stellt die sogenannte SELECT-Abfrage dar.

comparison-predicate ::=
  value-expression {= | <> | < | <= | > | >= } value-expression

between-predicate ::=
  value-expression [NOT] BETWEEN value-expression AND value-expression

like-predicate ::=
  match-value [NOT] LIKE pattern

match-value ::= char-value-expression

pattern ::= char-value-expression

null-predicate ::=
value-expression `IS` [`NOT`] `NULL`

in-predicate ::=
value-expression [`NOT`] `IN` { value-list | subquery }

value-list ::=
(literal [{, literal}...])

exists-predicate ::=
[`NOT`] `EXISTS` subquery

5.3.3 Spezielle Abfragen an eine Relation

Die spezifischen relationalen Operationen sind *Projektion*, *Selektion* und *Verbund (Join)*.

5.3.4 Aggregatfunktionen (`COUNT`, `SUM`, ...)

Die spezifischen relationalen Operationen sind *Projektion*, *Selektion* und *Verbund (Join)*.

5.3.5 Gruppierte Tabellen (`GROUP BY`)

Die spezifischen relationalen Operationen sind *Projektion*, *Selektion* und *Verbund (Join)*.

5.3.6 `SELECT`-Abfragen mit mehreren Tabellen in der `FROM`-Klausel (Produkt und Verbund)

Die spezifischen relationalen Operationen sind *Projektion*, *Selektion* und *Verbund (Join)*.

5.3.7 Explizite Verbundoperationen

Die spezifischen relationalen Operationen sind *Projektion*, *Selektion* und *Verbund (Join)*.

5.3.8 Geschachtelte `SELECT`-Abfragen (Unterabfragen)

Die spezifischen relationalen Operationen sind *Projektion*, *Selektion* und *Verbund (Join)*.

5.3.9 Abfragen mit `EXISTS`-Prädikat

Die spezifischen relationalen Operationen sind *Projektion*, *Selektion* und *Verbund (Join)*.

5.3.10 Mengenoperationen

Die spezifischen relationalen Operationen sind *Projektion*, *Selektion* und *Verbund (Join)*.

5.4 SQL-DML: Die Mutationen

5.2.1 Definition von Basistabellen

Kartesisches Produkt:

Seien $W_1, W_2, \ldots, W_n$ beliebige Mengen. Dann ist das kartesische Produkt $W_1 \times W_2 \times \ldots \times W_n$ folgendermaßen definiert:

$$W_1 \times W_2 \times \ldots \times W_n = \left\{ (w_1, w_2, \ldots, w_n) \,\middle|\, w_j \in W_j, j = 1, 2, \ldots, n \right\} .$$