3 Einführung in SQL

- Etwas SQL-Geschichte
- Der SQL-Markt
- Einführung
- DDL Datendefinitionsanweisungen
- DML Datenmanipulationsanweisungen
- Einbettung von SQL in 3GL-Sprachen
- SQL-Kritik

Kapitel 3, 4 und 5 geben einen Einblick in die Datenbanksprache SQL und deren Enwicklungsstand. Die einzelnen SQL-Implementierungen der Datenbankhersteller unterscheiden sich bezüglich des Funktionsumfangs erheblich. Aus diesem Grund wird hier bewusst keine konkrete SQL-Implementierung eines Herstellers beschrieben, sondern der »SQL-Standard«. Alle Hersteller sind bestrebt, sich früher oder später diesem Standard anzunähern. Die DBMS-Hersteller haben allerdings zusätzliche Erweiterungen zu diesem Standard implementiert und keiner unterstützt ihn vollständig. Nutzer von Datenbanksystemen sind gut beraten, wenn sie den SQL-Standard kennen und nur diesen nutzen. Sie sichern dadurch die Kompatibilität ihrer Programme mit zukünftigen Releases der Datenbankhersteller und erleichtern einen späteren Austausch des Datenbanksystems mit dem eines anderen Herstellers. Das Anwendungsprogramm wird somit unabhängiger vom eingesetzten DBMS. Des Weiteren ist es mit der Kenntnis des Standards möglich, Datenbanksysteme bezüglich ihres Enwicklungsstandes zu beurteilen. Das Problem dabei ist allerdings, dass kein einzelner »SQL-Standard« existiert. Der Standardisierungsprozess ist seit vielen Jahren in Arbeit und hat seit seinen Anfängen bisher drei entscheidende Versionen hervorgebracht. Diese werden in den nächsten drei Kapiteln erläutert.

Wie in Kapitel 2 beschrieben wurde, ist es möglich, mit Hilfe der relationalen Algebra Daten auszuwerten. Es gibt allerdings kein kommerzielles DBMS, welches eine solche Programmschnittstelle vorsieht. Warum findet diese Algebra in der Praxis keine direkte Verwendung? Dafür gibt es mehrere Gründe:

- Der Sprachumfang, wie er von E. F. Codd definiert wurde, ist mathematisch vollständig. Für den praktischen Gebrauch fehlen jedoch wichtige Operatoren (z. B. statistische Operatoren wie Summe, Min, Max, Mittelwert oder Gruppenfunktionen).
- Die relationale Algebra ist **prozedural**, d.h *wie* ein gewünschtes Ergebnis erreicht wird, muss der Anwender selbst formulieren. In der Praxis ist dies sehr schwierig. Der Programmierer muss sich auch über die internen Abläufe im klaren sein, um den Befehl optimal zu formulieren.

So tendierten die relationalen Sprachen von Anfang an dazu, **nicht-prozedural** zu sein. Dies sind »Programmiersprachen«, die nur noch ausdrücken, welches Ergebnis gewünscht wird, und nicht, wie der Rechner zu diesem Ergebnis kommt. Diese Art von Sprachen werden zu den Programmiersprachen der vierten Generation (4GL)[1] gezählt (im Gegensatz zu 3GL-Sprachen wie Java, C#, Pascal, COBOL usw).

Eine nicht-prozedurale Alternative zur prozeduralen relationalen Algebra ist das **relationale Kalkül** [Codd 72-1]. Eine Implementierung dieses relationalen Kalküls ist die Erweiterung der Programmiersprache PASCAL um die neue Datenstruktur RELATION in **PASCAL/R** [Schmidt 80] der Universität

1. Abkürzung: 4GL = 4th Generation Language

Hamburg. Später wurde dann Modula-2 im LIDAS-Projekt der ETH Zürich um relationale Strukturen und Anweisungen in Form des relationalen Kalküls erweitert. Die dadurch entstandene neue Sprache heißt **MODULA/R** [LIDAS 83].

Trotz der Vorteile, die eine solche Spracherweiterung bietet (z.B. Realisierung des Domain-Konzeptes des relationalen Modells durch das TYPE-Konzept dieser Sprachen), konnten sich weder PASCAL/R noch MODULA/R im kommerziellen Bereich durchsetzen.

Die derzeit verbreitetste Sprache für relationale Datenbanken ist, trotz aller Kritik, die an ihr geübt wird, unbestritten SQL. Auch in Zukunft wird diese Sprache keine Konkurrenz bekommen, was sich aus der Tatsache ableiten lässt, dass mittlerweile alle Hersteller relationaler Datenbanken SQL als Zugriffssprache anbieten.

3.1 Etwas SQL-Geschichte

Anfang der siebziger Jahre wurde in den IBM Forschungslaboratorien in San José, Kalifornien, ein Forschungsprojekt begonnen, das sich »System R« nannte. Es sollte die Praktizierbarkeit der relationalen Theorien untersuchen. Innerhalb dieses Projektes wurde eine Sprache namens SQUARE definiert, die jedoch von der Syntax zu mathematisch orientiert war.

Von den IBM-Mitarbeitern R.F. Boyce und D.D. Chamberlain wurde die Sprache SEQUEL (sprich: siequel) entwickelt, die später in SQL umbenannt wurde. Man lehnte hierbei die Syntax mehr an Begriffe der englischen Umgangssprache wie z.B. SELECT, FROM, WHERE an. Diese ersten Entwürfe von SQL waren jedoch in der Praxis unbrauchbar, da die Sprache nur **eine** Tabelle verarbeiten konnte (also keinen Join kannte). Außerdem war sie nur für Single-User-, also nicht für den Parallelbetrieb geeignet. Dieses Projekt wurde jedoch als so vielversprechend betrachtet, dass man daraus weitere Produkte entwickelte, die der System-R-Technologie entsprachen.

Mittlerweile sind diese Produkte auf dem Markt unter den Namen **DB2** und **SQL/DS** eingeführt. DB2 ist das Datenbanksystem für die IBM-Betriebssysteme MVS/370 und MVS/XA. SQL/DS ist unter den Betriebssystemen VM/CMS und DOS/VSE verfügbar. Für beide Datenbanksysteme wurde eine benutzerfreundliche Oberfläche entwickelt, die sich **QMF** (Query Management Facility) nennt.

Seit dieser Zeit wurden von fast allen DB-Herstellern SQL-Schnittstellen zu ihren relationalen und nicht-relationalen Datenbanksystemen entwickelt.

3.2 Der SQL-Markt

Die heute verfügbaren Datenbanksysteme, die eine SQL-Schnittstelle anbieten, müssen in zwei Klassen unterteilt werden. Zum einen werden sie von Hardware- oder Betriebssystemherstellern angeboten, die ein eigenes, speziell auf ihre Hardware und ihr Betriebssystem abgestimmtes Datenbanksystem vertreiben (z.B. Microsoft SQL Server), zum anderen gibt es reine Softwarehersteller, die sich auf die Entwicklung von Datenbanken spezialisiert haben und das von ihnen entwickelte Datenbanksystem auf verschiedene Hardware portieren (z.B. ORACLE). Daneben gibt es noch auf SQL basierende Datenbanksysteme, die aus nicht-kommerziellen Bestrebungen heraus entstanden sind (OpenSource), wie MySQL [Kofler] (mysql.org) oder PostgreSQL [Momjian] (www.pgsql.com). Gerade diese Datenbanksysteme erfreuen sich besonders bei Internetanwendungen zunehmender Beliebtheit.

3.3 Einführung

Seit einigen Jahren gibt es einen ANSI-Standard für SQL, der in [Date 87-1] sehr ausführlich beschrieben ist. Dieser Standard wurde 1989 durch die ANSI-Kommission definiert und wird auch als SQL-1-Standard bezeichnet. Da allerdings die bestehenden SQL-Implementierungen schon vor Jahren begonnen wurden, sind die einzelnen heute verfügbaren Produkte sehr unterschiedlich.

In den nachfolgenden Abschnitten werden die wichtigsten Befehle der Sprache SQL-1 vorgestellt. Dieses Kapitel ist nur eine Einführung in SQL und erhebt nicht den Anspruch darauf, ein SQL-Lehrbuch zu sein. Empfehlenswerte SQL-Lehrbücher sind z.B. [SQL 86], [SQL 87] oder [Ebner]. Der Sprachumfang von SQL-1 wurde in den letzten Jahren durch weitere Standards erheblich erweitert. Diese Erweiterungen, die alle auf dem SQL-1-Standard aufbauen, werden in Kapitel 4 und 5 beschrieben.

Da sich einige Begriffe des relationalen Modells von den SQL-Begriffen unterscheiden, werden in diesem Kapitel abweichend zum vorherigen Kapitel folgende Begriffe verwendet:

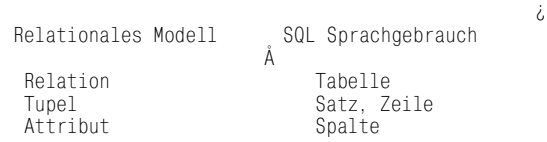

Abbildung 3.1: Gegenüberstellung der Begriffe des relationalen Modells und SQL

Ein SQL-Beispiel:

Die meisten Datenbanksysteme besitzen eine interaktive Benutzerschnittstelle.[2] Mit ihr ist es auf einfache Art möglich, SQL-Befehle zu testen.

Mit der Anweisung

```
CREATE TABLE ARTIKEL
       (ART_NR         SMALLINT,
        ART_BEZ         CHAR(13),
        ART_ART         CHAR(11),
        LIEF_NR         SMALLINT )
```

wird eine leere Tabelle mit folgendem Aufbau definiert:

```
ARTIKEL
                                              ¿
  ART_NR   ART_BEZ      ART_ART     LIEF_NR
Æ                                             µ
```

Mit dem INSERT-Befehl ist es möglich, Daten in eine Tabelle einzufügen:

```
INSERT INTO ARTIKEL
            VALUES (1, 'Multisync II', 'Monitor', 1)
INSERT INTO ARTIKEL
            VALUES (2, 'Multisync I', 'Monitor', 2)
```

```
ARTIKEL
                                              ¿
  ART_NR   ART_BEZ      ART_ART     LIEF_NR
Æ                                             µ
   1      Multisync II  Monitor       1
   2      Multisync I   Monitor       2
```

Mit dem SELECT-Befehl werden Daten aus einer oder mehreren Tabellen selektiert:

```
SELECT         ART_BEZ, ART_ART
  FROM         ARTIKEL
 WHERE        LIEF_NR = 2
```

Ergebnis:

```
                           ¿
  ART_BEZ      ART_ART
Æ                          µ
 Multisync I   Monitor
```

2. Zum Beispiel SPUFI in DB2, ISQL in INGRES oder SQL+ in ORACLE

3.3.1 SQL – Befehlsübersicht

Bei Datenbankbefehlen unterscheidet man sowohl bei den hierarchischen (z.B. IMS von IBM), den Netzwerk- (z.B. IDMS von Computer Associates) als auch bei den relationalen Datenbanksystemen zwei Gruppen von Datenbankanweisungen:

DDL – Data **D**efinition **L**anguage (Datendefinitionssprache)

Dazu zählen alle Datenbankanweisungen, mit denen die logische Struktur der Datenbank bzw. der Tabellen der Datenbank beschrieben bzw. verändert wird. Hierzu gehören folgende Befehle:

CREATE TABLE: neue Basistabelle erstellen

CREATE VIEW: Definieren einer logischen Tabelle

GRANT: Benutzerberechtigungen vergeben

DML – Data **M**anipulation **L**anguage (Datenmanipulationssprache)

Dazu zählen alle Anweisungen an das Datenbanksystem, die dazu dienen, die Daten zu verarbeiten. Hierzu gehören folgende Befehle:

SELECT: Daten aus der Datenbank lesen

DELETE: Zeilen einer Tabelle löschen

UPDATE: Zeilen einer Tabelle ändern

INSERT: Zeilen einer Tabelle hinzufügen.

Bei den hierarchischen und den Netzwerk-Datenbanken ist der Sprachumfang der DDL und DML relativ groß. Selbst bei der Erstellung einfacher Anwendungen ist ein relativ großer Aufwand erforderlich, bis die dazugehörige Datenbank definiert ist und die DML-Programme geschrieben und getestet sind.

Die Sprache SQL dagegen besteht aus nur wenigen einfachen Befehlen, die sowohl den DDL- als auch den DML-Teil abdecken.

3.3.2 EBNF als Hilfsmittel zur Sprachbeschreibung

Um die Sprache SQL exakt zu definieren, werden im Folgenden die Beschreibungssymbole der **E**rweiterten-**B**ackus-**N**aur-**F**orm **(EBNF)** verwendet. Die EBNF verwendet folgende Symbole:

Reservierte Wörter

werden groß geschrieben: `SELECT, FROM, UPDATE`

Alternativentrennung: `| [ + | - ] Konstante`

Optionalsymbol: `[ ]`

(keinmal, oder einmal) `SELECT [ ALL ] ...`

Wiederholungssymbol: `{ }`

(keinmal, einmal oder mehrmals) `FROM tabelle { , tabelle }`

3.4 DDL – Datendefinitionsanweisungen

Eine **Datenbank**, das ist der Bereich, in dem ein Programm operieren kann, besteht aus einer Menge von **SCHEMATA**. Eine Datenbank ist definiert durch die Zusammenstellung aller Daten aller SCHEMATA. Ein SCHEMA besteht aus einer Menge von Tabellendefinitionen, VIEWs und Berechtigungen. Ein SCHEMA wird durch die **CREATE SCHEMA**-Anweisung definiert.

Abbildung 3.2: Syntax der CREATE SCHEMA-Anweisung

```
CREATE SCHEMA                                                          ¿

  create schema  ::=  CREATE SCHEMA AUTHORIZATION user
                      [ schema element { , schema element } ]

  schema element ::=    create table
                      | grant anweisung
                      | view definition
```

3.4.1 CREATE TABLE – Tabellen erstellen

Mit dem **CREATE TABLE**-Befehl wird eine neue (leere) Tabelle definiert und angelegt. In SQL werden zwei Arten von Tabellen unterschieden:

- Die **Basistabellen** ist eine Tabelle, die physisch, in der Regel auf einer Magnetplatte, Daten aufnimmt.
- Die **VIEW** ist eine logische Tabelle (siehe 3.4.3), die nur eine logische Sicht auf eine oder mehrere physische Tabellen darstellt.

Mit dem Befehl `CREATE TABLE` wird eine neue Basistabellen definiert, mit `CREATE VIEW` (Kapitel 3.4.3) wird eine VIEW definiert.

Abbildung 3.3: Syntax der CREATE TABLE-Anweisung

```
CREATE TABLE                                                           ¿

  create table ::=  CREATE TABLE basistabellen name
                       ( spalten def { , spalten def } )

  spalten def ::=  spalten name  daten typ [ NOT NULL ]

  daten typ   ::=    CHARACTER [ ( lΣnge ) ]
                   | NUMERIC   [ ( precision [,scale ] ) ]
                   | DECIMAL   [ ( precision [,scale ] ) ]
                   | INTEGER
                   | SMALLINT
                   | FLOAT  [ ( precision ) ]
                   | REAL
                   | DOUBLE PRECISION
```

Precision ist eine ganze Zahl größer 0, die anzeigt, wie viele Ziffern die Zahl beinhalten soll.

Scale ist eine ganze Zahl größer 0 und kleiner **precision**, die die Anzahl der Nachkommastellen angibt.

```
CREATE TABLE ARTIKEL
       (ART_NR          SMALLINT NOT NULL,
        ART_BEZ         CHARACTER(13),
        ART_ART         CHARACTER(11),
        LIEF_NR         SMALLINT )
```

SQL-Datentypen

Wie in allen anderen Programmiersprachen gibt es auch in SQL verschiedene Datentypen. Dassdamals das ANSI-Standard-Komitee nur diese wenigen Datentypen (die in der Praxis meist nicht ausreichen) in die Definition aufgenommen hat, ist damit begründet, dass alle diese Datentypen mit den damals verbreitetsten Programmiersprachen FORTRAN, COBOL und PL/1 kompatibel waren. Heute stellen die meisten Datenbanksysteme allerdings weitere nützliche Datentypen zur Verfügung, indem sie zumindest teilweise den SQL-2-Standard oder Objektdefinitionen aus dem SQL-3-Standard unterstützen.

Beim erstmaligen Installieren eines Datenbankmanagementsystems werden automatisch eine Reihe von Tabellen vom DBMS aufgebaut: die so genannten **Systemtabellen**. Die Menge der Systemtabellen nennt man **Systemkatalog**. Welche Systemtabellen es gibt und was in ihnen festgehalten wird, ist in jedem Datenbanksystem unterschiedlich.[3] Von einem relationalen Datenbanksystem wird allerdings gefordert, dass es möglich sein muss, die Systemtabellen mit der eigenen Datenbanksprache (z.B. SQL) zu verarbeiten. Der CREATE TABLE-Befehl hält die Struktur der neuen Tabelle in diesen Systemtabellen fest.

3.4.2 GRANT – Benutzer berechtigen

Mit Hilfe des GRANT-Befehls werden die Daten vor unberechtigtem Zugriff geschützt.

Dem Datenbanksystem sind eine Menge von Benutzern bekannt. Ein Benutzer identifiziert sich beim Anmelden (LOGON) an den Rechner mit einem Namen und einem Passwort. Manche DBMSe fordern ein zweites Anmelden beim Benutzen des DBMS (ORACLE). Somit weiß das DBMS – unabhängig davon, ob ein solches zweites LOGON durchgeführt wurde – zu jedem Zeitpunkt, wer einen bestimmten DB-Befehl abgesetzt hat, und kann prüfen, ob dieser Anwender zu diesem Befehl berechtigt ist.

3. In SQL-2 ist der Systemkatalog normiert.

Gerade, was das Konzept des Datenschutzes betrifft, gibt es bei den einzelnen DB-Systemen gravierende Unterschiede. Allerdings richtet sich kein DBMS bezüglich der Benutzerberechtigungen nach dem SQL-1 Standard.

Im SQL-1-Standard wird keine Benutzererlaubnis benötigt, um eine Basistabelle zu kreieren. Beim Kreieren einer VIEW wird für alle in der VIEW benutzten Basistabellen eine SELECT-Berechtigung benötigt.

Der GRANT-Befehl hat folgenden Aufbau:

Abbildung 3.4: Syntax der GRANT-Anweisung

```
GRANT:
                                                                    ¿
  grant anweisung  ::=
           GRANT privileg ON tabelle TO
                 benutzername { , benutzername }
               | PUBLIC
               [ WITH GRANT OPTION ]

  privileg  ::=   ALL [ PRIVILEGES ]
                | operation { , operation }

  operation ::= SELECT | INSERT | DELETE |
                UPDATE [ ( spaltenname {, spaltenname } ) ]
```

Wurde anstelle von Benutzernamen **PUBLIC** aufgeführt, dann gelten die Privilegien für alle dem System bekannten Benutzer.

WITH GRANT OPTION räumt dem Benutzer das Recht ein, seine Privilegien auf einer Tabelle an andere Benutzer weiterzugeben.

Beispiel 1

Mit diesem GRANT dürfen alle dem DBMS bekannten Benutzer die Tabelle ARTIKEL lesen und jegliche Art von Veränderungen in ihr vornehmen:

```
GRANT    ALL
ON        ARTIKEL
TO            PUBLIC
```

Beispiel 2

Dieser GRANT berechtigt den Benutzer HERMANN, die ganze Tabelle ARTIKEL zu lesen und die Spalten ART_BEZ, ART_ART und LIEF_NR zu ändern:

```
GRANT    UPDATE (ART_BEZ, ART_ART, LIEF_NR), SELECT
ON        ARTIKEL
TO            HERMANN
```

3.4.3 CREATE VIEW – Definieren einer logischen Tabelle

Unter einer VIEW (Sicht) versteht man eine logische Sicht auf eine oder mehrere physische Basistabellen, die mit CREATE TABLE angelegt wurden.

Über folgende Basistabelle

```
KUNDEN
                                                    ¿
 KNR KNAME     KSTRAáE     KHSNR KPLZ  KORT
Æ                                                   µ
  1  FAMA      Goethestr     19  60700 Langen
  2  GSA       Hoenbergstr   2a  63700 Oberursel
  3  Kl┌ckner   Paradiesweg   7   20800 Pinneberg
  4  RADOVAN   Im Sand       45  10000 Berlin
  5  G┌hler     Schmalweg     11  69000 Heidelberg
```

soll eine VIEW definiert werden, die alle Kundennamen und Kunden-Postleitzahlen im PLZ-Gebiet 6 enthält. Mit

```
CREATE    VIEW KUNDEN6 (KNAME, KPLZ) AS
SELECT    KNAME, KPLZ
  FROM    KUNDEN
 WHERE    KPLZ LIKE '6%'
```

wird folgende logische Tabelle definiert:

```
KUNDEN
    É          ∩                É     ∩          ¿    KUNDEN6
 KNRºKNAME    ºKSTRAáE     KHSNRºKPLZ ºKORT                      ¿
Æ                                                   µ       KNAME     KPLZ
  1 ºFAMA     ºGoethestr     19 º60700ºLangen          Æ                  µ
  2 ºGSA      ºHoenbergstr   2a º63700ºOberursel  ===>  FAMA      60700
              ¼            Å          ¼                   GSA       63700
  3  Kl┌ckner  Paradiesweg   7   20800 Pinneberg         G┌hler    69000
  4  RADOVAN   Im Sand       45  10000 Berlin
    É          ∩            Å     É     ∩
  5 ºG┌hler   ºSchmalweg     11 º69000ºHeidelberg
              ¼                        ¼
```

Abbildung 3.5: VIEW über die Tabelle KUNDEN

Bei der Definition einer VIEW werden

- nur VIEW-Definitionsparameter im Systemkatalog abgelegt,
- keine Datenbankoperationen ausgeführt,
- keine physischen Tabellen erzeugt.

Erst beim Zugriff auf eine VIEW (durch einen SELECT-, DELETE- oder UPDATE-Befehl) wird der jeweilige SQL-Befehl um die VIEW-Definition erweitert (VIEW-Auflösung):

```
                                                    ¿
  Urspr┌nglicher                 Vom DBMS erzeugter
    SELECT                          SELECT
                              Å
  SELECT  KNAME                 SELECT  KNAME
    FROM  KUNDEN6                 FROM  KUNDEN
   WHERE  KNAME LIKE  G%         WHERE  KNAME LIKE  G%
                                   AND  KPLZ  LIKE  6%
```

Abbildung 3.6: Auflösung einer VIEW bei der Ausführung eines SELECT-Befehls

Hier die vollständige Syntax des CREATE VIEW-Befehls:

Abbildung 3.7: Syntax der CREATE VIEW-Anweisung

```
CREATE VIEW
                                                                  ¿
 CREATE VIEW  viewname [ ( spaltenname {, spaltenname } ) ]
         AS   select spezifikation
              [ WITH CHECK OPTION ]
```

Mit WITH CHECK OPTION wird bei allen datenverändernden Befehlen, die diese VIEW benutzen, geprüft, ob die Daten noch über die VIEW erreichbar sind.

So wird z. B.

```
UPDATE    KUNDEN6
   SET    KPLZ = '50000'
```

abgewiesen, da dieser Befehl Datensätze derart verändert, dass sie nicht mehr zur VIEW gehören. VIEW-Update-Operationen sind ein sehr komplexes Thema und werden in [DATE 86-1], [CODD 87] und [CODD 90] behandelt.

Einschränkungen bei der Definition von VIEWs

Der SELECT-Befehl in der VIEW-Definition unterliegt im SQL-1-Standard einigen Restriktionen. So darf in ihm kein UNION benutzt werden – eine nicht sofort einsichtige Einschränkung.

Eine weitere Einschränkung gibt es bezüglich der Verwendung von GROUP BY und HAVING in der VIEW-Definition. Diese ist allerdings nicht allgemeingültig definiert und somit aus den Unterlagen des jeweiligen DBMS-Handbuchs zu entnehmen.

Einschränkungen bezüglich Updates auf VIEWs

Betrachten wir folgende VIEW-Definition:

```
CREATE    VIEW SUM (S3) AS
SELECT    S1 + S2
  FROM    TAB1
```

```
  TAB1                        SUM
        ¿                           ¿
 S1   S2                      S3
Æ       µ                    Æ   µ
  2   3                        5
  4   7                       11
  3  11                       14
  0   2                        2
```

und folgende SQL-Befehle:

```
UPDATE  SUM  INSERT INTO SUM(S3)
   SET  S3 = 10  VALUES     (10)
```

Bei beiden Operationen ist es nicht eindeutig, welche Daten in die unter der VIEW liegende Basistabelle (und in welche Spalte) geschrieben werden sollen.

SQL-1 lässt jedoch nicht unbedingt bei jeder theoretisch veränderbaren VIEW[4] ein UPDATE zu.

In SQL-1 sind Daten in einer VIEW veränderbar, wenn die SELECT-Spezifikation in der VIEW-Definition

1. nicht das Schlüsselwort DISTINCT enthält,
2. der Spaltenausdruck nur einfache Spaltennamen, also keine arithmetischen Ausdrücke wie S1 + 4 oder eine Funktion wie AVG(S1) enthält,
3. in der FROM-Komponente nur einen UPDATE-baren Tabellennamen beinhaltet,
4. die WHERE-Bedingung keine Unterabfrage besitzt,
5. und keine GROUP-BY- und keine HAVING-Komponente enthält.

VIEW-Beispiel

Abschließend ein Beispiel für eine etwas umfangreichere VIEW-Definition:

Für die beiden folgenden Basistabellen soll eine VIEW erstellt werden, die den Kundennamen, die bestellte Artikelnummer und die Mengen beinhaltet.

Basistabellen:

```
BESTELLUNGEN                    KUNDEN
                ¿                                                          ¿
 KNR ART_NR MENGE                KNR KNAME     KSTRAáE     KHSNR KPLZ  KORT
Æ                µ              Æ                                          µ
  1    3      3                    1  FAMA      Goethestr     19  60700 Langen
  4    2      2                    3  Kl┌ckner  Paradiesweg   7   20800 Pinneberg
  3    8      1                    4  RADOVAN   Im Sand       45  10000 Berlin
  3    3      1                    5  G┌hler    Schmalweg     11  69000 Heidelberg
```

Mit

```
CREATE        VIEW BESTELLISTE ( KNAME, ART_NR, MENGE) AS
SELECT        KNAME, ART_NR, MENGE
                  FROM BESTELLUNGEN B, KUNDEN K
                  WHERE B.KNR = K.KNR
```

4. Wird ausführlich in [DATE86-1], [CODD 87] und [CODD 90] behandelt.

wird folgende VIEW definiert:

```
BESTELLISTE
                              ¿
 KNAME     ART_NR  MENGE
Æ                             µ
 FAMA         3       3
 Kl┌ckner     8       1
 Kl┌ckner     3       1
 RADOVAN      2       2
```

Anwendungsmöglichkeiten von VIEWs

VIEWs werden benutzt

1. wie ein Unterprogramm: Man kann eine VIEW benutzen, um komplexe SQL-Anweisungen schrittweise zu vereinfachen und zu testen.
2. um eine Benutzersicht bereitzustellen: Meist werden die Tabellen so aufgesplittet, dass es für einen Anwender, der nicht genügend Erfahrung mit SQL hat, zu schwierig ist, für eine spezielle Abfrage die Tabellen wieder zusammenzufügen. Hier findet die VIEW eine sinnvolle Anwendung, indem man für diesen Anwender eine für ihn sinnvolle Benutzersicht auf mehrere Tabellen definiert, die für ihn dann wie eine einzige einfache Tabelle aussieht.
3. aus Datenschutzgründen: Mit dem GRANT-Befehl ist es nur möglich, für einen bestimmten Benutzer die Zugriffsrechte spaltenweise einzuschränken. Manchmal ist es allerdings notwendig, dass unterschiedliche Benutzer nur auf bestimmte Zeilen einer Tabelle zugreifen dürfen. Dies wird mit Hilfe einer VIEW ermöglicht, indem in einer Spalte der Tabelle die Benutzernamen mit abgelegt werden. Dazu folgendes Beispiel:

Basistabelle:

```
KUNDEN
                                                              ¿
 KNR KNAME     KSTRAáE     KHSNR KPLZ  KORT       SACHB
Æ                                                             µ
  1  FAMA      Goethestr     19  60700 Langen     M┐ller
  3  Kl┌ckner  Paradiesweg   7   20800 Pinneberg  Schmitt
  4  RADOVAN   Im Sand       45  10000 Berlin     Schmitt
  5  G┌hler    Schmalweg     11  69000 Heidelberg M┐ller
```

Über die folgende VIEW dürfen die zuständigen Sachbearbeiter nur auf ihre Teile der Tabelle zugreifen:

```
CREATE    VIEW DEINEKUNDEN AS
 SELECT  *
  FROM    KUNDEN
 WHERE    SACHB = USER
```

Mit USER ist der aktuelle Benutzer gemeint, welcher auf die VIEW zugreifen möchte.

3.5 DML – Datenmanipulations-anweisungen

3.5.1 Die SELECT-Anweisung (Daten aus der Datenbank lesen)

Zu den DML-Befehlen gehört der wohl wichtigste und komplexeste SQL-Befehl – SELECT –, obwohl er keine Daten verändert. Mit ihm werden Daten aus einer oder mehreren Tabellen gelesen. Das Ergebnis des SELECT sind Daten, die wiederum wie eine Tabelle strukturiert sind.

Wie wir schon gesehen haben, wird der SELECT-Befehl, oder Teile davon, in anderen SQL-Befehlen mitverwendet, z. B. in der VIEW-Definition.

So einfach diese Anweisung im ersten Moment auch aussieht, es bedarf sehr viel praktischer Übung, bis man in der Lage ist, auch komplexe Befehle zu formulieren. Wie gut man SQL beherrscht, hängt direkt davon ab, wie gut man diesen Befehl kennt, da er die am häufigsten verwendete Anweisung ist und auch in anderen SQL-Befehlen Verwendung findet.

Eine SELECT-Anweisung besteht vereinfacht ausgedrückt aus SELECT-Spezifikationen, die mit UNION verbunden werden können.

SELECT-Spezifikation

Wir beginnen bei der Definition der SELECT-Anweisung zunächst mit der

```
select spezifikation:                                          ¿

  select spezifikation ::=
         SELECT [ ALL | DISTINCT ] spaltenausdruck
                                   tabellenausdruck
```

Abbildung 3.8: Syntax der SELECT-Spezifikation

DISTINCT unterdrückt doppelte Zeilen im Ergebnis. ALL (default) bewirkt, dass alle, also auch gleiche Zeilen, im Ergebnis erscheinen.

- Der **Spaltenausdruck** bestimmt, welche Spalten in die Ergebnisrelation aufgenommen werden.
- Der **Tabellenausdruck** wählt Zeilen der Eingangstabelle(n) aus.

```
          SELECT    Spaltenausdruck
                                                       ¿
FROM                      Å Å      Å       Å
 Tabellenausdruck

                           ß
```

Abbildung 3.9: Mit dem Spaltenausdruck werden Spalten, mit dem Tabellenausdruck Zeilen selektiert.

Hier ein Beispiel für eine SELECT-Spezifikation:

Zeige alle Lieferanten und die Anzahl der verschiedenen Teile, welche sie liefern!

```
SELECT DISTINCTl.lief_name,' Anzahl gel. Artikel: ',
                count(a.art_nr)
  FROM lieferanten l, artikel a
 WHERE l.lief_nr = a.lief_nr
 GROUP BY l.lief_name

                                                    ¿
 lief_name
Æ                                                   µ
 Thomson         Anzahl gel. Artikel:   1
 Audio Master    Anzahl gel. Artikel:   2
 NEC             Anzahl gel. Artikel:   2
 Easyprint       Anzahl gel. Artikel:   3
```

Im SQL-1-Standard ist es nicht möglich, die Überschriften der Ergebnisspalten zu beinflussen (`count(a.art_nr) AS anzahl`). Mit Hilfe dieser SELECT-Spezifikation kann nun die Spitze der SELECT-Syntax definiert werden:

Abbildung 3.10: Syntax der SELECT-Anweisung

```
SELECT:
                                                                    ¿
 select ausdruck ::=   select term
                     | select ausdruck UNION [ ALL ] select term

  select term  ::=  select spezifikation | ( select ausdruck )
```

Mit **UNION** können mehrere SELECT-Spezifikationen gemäß dem relationalen UNION-Operator verknüpft werden.[5] Da es nach den relationalen Regeln keine doppelten Zeilen in einer Tabelle gibt, werden bei der UNION-Verknüpfung der SELECT-Spezifikationen die doppelten Zeilen eliminiert. Mit der Angabe von ALL nach UNION kann dies unterdrückt werden.

Laut dieser Syntax dürfen mit UNION nur zwei SELECT verknüpft werden. Bei der Verknüpfung von drei (oder mehreren) SELECT-Spezifikationen a,b und c müssen diese eingeklammert werden:

`(a UNION b) UNION c` oder `a UNION (b UNION c)`

Die Klammerung ist notwendig (siehe auch Kapitel 2), da

```
(a UNION ALL b) UNION c
```

nicht das gleiche ist wie

```
a UNION ALL (b UNION c).
```

Spaltenausdruck

Im Spaltenausdruck wird festgelegt, welche Spalten (Daten) die Ergebnisrelation besitzen soll.

```
SELECT l.lief_name,'Anzahl gel. Artikel:',count(a.art_nr)
...
...
```

5. siehe Kapitel 2.4.4.

Dieser Spaltenausdruck gibt an, dass folgende Daten in die Ergebnisrelation geschrieben werden sollen:

`l.lief_name`	die Lieferantennamen aus der Relation Lieferanten,
`'Anzahl gel. Artikel:'`	ein konstanter Text,
`count(a.art_nr)`	die Anzahl der Ergebniszeilen.

Der Spaltenausdruck ist folgendermaßen definiert:

Abbildung 3.11: Syntax des Spaltenausdrucks

```
spalten ausdruck:                                                    ¿

  spalten ausdruck  ::=   scalar ausdruck { , scalar ausdruck }
                        | *

  scalar ausdruck   ::=   [   |   ] term {   |    term }

  term              ::=   factor { * | /  factor }

  faktor            ::=   atom
                        | spalten referenz
                        | funktions referenz
                        | ( scalar ausdruck )

  atom              ::= konstante | USER

  funktions referenz ::=   COUNT(*)
                         | distinct funktions ref
                         | all funktions ref

  distinct funktions ref  ::=
    AVG | MAX | MIN | SUM | COUNT   (DISTINCT spalten referenz )

  all funktions ref  ::=
    AVG | MAX | MIN | SUM | COUNT   ( [ ALL ] scalar ausdruck )

  spalten referenz   ::= [ tabelle | range variable . ] spaltenname
```

Wie man an dieser Syntax erkennen kann, basiert der Spaltenausdruck in erster Linie auf durch Komma getrennten, arithmetischen Ausdrücken. In diesen können Konstanten, Funktionen oder Tabellenspalten beliebig mit den arithmetischen Operatoren +, –, *, / verknüpft werden.

Funktionen

Standard-SQL erlaubt die Benutzung von fünf Funktionen:

Funktion	Bedeutung
COUNT	Anzahl der Werte in der Spalte
SUM	Summe der Werte in der Spalte
AVG	Mittelwert der Spalte
MAX	größter Wert in der Spalte
MIN	kleinster Wert in der Spalte

Die Funktionen `COUNT(spaltenname)`, `SUM(spaltenname)`, `AVG(spaltenname)`, `MAX(spaltenname)` und `MIN(spaltenname)` werten nur die unterschiedlichen Werte der Spalte aus, wenn das Wort `DISTINCT` folgt (Default ist `ALL`). Wenn `DISTINCT` angegeben wurde, darf das Argument nur ein Spaltenname sein wie z. B. `SUM( DISTINCT umsatz)`. Ansonsten darf ein skalarer Ausdruck als Argument folgen, z. B. `SUM(umsatz*1.16)`.

`COUNT(*)` bezieht sich nicht auf eine spezielle Spalte und darf nicht in Verbindung mit der Funktionserweiterung `DISTINCT` verwendet werden.

`SELECT COUNT( DISTINCT *)` ist nicht erlaubt, aber

`SELECT DISTINCT COUNT(*)` ist ein gültiger SQL-Ausdruck.

Tabellenausdruck

Bevor wir mit der formalen Syntax des Tabellenausdrucks fortfahren, hier zunächst eine Übersicht seiner Einzelteile:

Abbildung 3.12: Die Einzelteile des Tabellenausdrucks

```
¿                                                                    
  FROM           definiert die Eingangstabellen
                 Å
  WHERE          selektiert aufgrund einer Bedingung die
                 Zeilen der Eingangstabellen
                 Å
  GROUP BY       gruppiert Zeilen auf der Basis gleicher
                 Spaltenwerte
                 Å
  HAVING         selektiert nur Gruppen im GROUP BY Teil
                 laut einer Bedingung
```

Abbildung 3.13: Syntax des Tabellenausdrucks

```
tabellenausdruck:
                                                                     ¿
  tabellenausdruck ::=    from klausel
                        [ where klausel ]
                        [ group by klausel ]
                        [ having klausel ]

  from klausel     ::= FROM  tabelle [ range variable ]
                         { , tabelle [ range variable ] }

  where klausel    ::= WHERE such bedingung

  group by klausel ::= GROUP BY   spaltenreferenz
                              { , spalten referenz }

  having klausel   ::= HAVING suchbedingung
```

- `Tabelle` kann eine **Basistabelle** oder eine **VIEW** sein. Eine Tabelle kann qualifiziert, also in der Form `besitzer.tabellenname`, oder unqualifiziert angegeben werden.
- `Spalten-Referenz` ist eine durch Kommata getrennte Liste von qualifizierten (in der Form `tabellenname.spaltenname` oder `range-variable.spaltenname`) oder unqualifizierten (nur `spaltenname`) Spaltennamen (siehe Abb. 3.11).

- `Range-Variablen` werden benötigt, wenn in einem SELECT die gleiche Tabelle mehrmals gebraucht wird. Mit Einführung einer solchen Range-Variablen wird einer Tabelle für diesen SELECT ein anderer Name gegeben, auf den man sich dann individuell beziehen kann. Dies wird z.B. beim **Auto-Join** (Verknüpfung einer Tabelle mit sich selbst) oder bei manchen **korrelierten Subselects** benötigt. Oft wird die Range-Variable aber auch nur benutzt, um einen langen Tabellennamen in einem SELECT zu vereinfachen.

Hier einige SELECT-Beispiele:

Beispiel 1:

Zeige alle Kunden auf, von welchen eine Bestellung vorliegt. **Aufgabe**

Lösung

```
SELECT  k.kname
FROM    kunden k, bestellungen b
WHERE   k.knr = b.knr
```

Beispiel 2:

Zeige alle Monitore aus der Artikel-Tabelle auf. **Aufgabe**

Lösung

```
SELECT art_bez, art_art
FROM   artikel
WHERE  art_art = 'Monitor'
```

Beispiel 3:

Hier ein Beispiel, das die Notwendigkeit einer Range-Variablen verdeutlicht.

Zeige alle Mitarbeiter auf, die mehr als ihr Chef verdienen. **Aufgabe**

Für dieses Beispiel wird eine weitere Tabelle definiert. **Lösung**

```
Mitarbeiter
                                    ¿
 name           chef           gehalt
Æ                                   µ
 Mºller                          4500
 Gºnther        Mºller           3000
 Weber          Mºller           4000
 Frisch         Mºller           4600
 Heck           Gºnther          3100
 Hinz           Gºnther          2700
 Hilt           Gºnther          2900
 Meier          Frisch           2100
 Korn           Frisch           3200
```

Um diese Frage beantworten zu können, muss die MITARBEITER-Tabelle mit sich selbst verknüpft werden (Auto-Join). Dies geschieht, indem die betreffende Tabelle in der FROM-Klausel gleich zweimal jeweils mit verschiedenen Range-Variablen aufgeführt wird. Durch die Join-Bedingung `m1.chef = m2.name` wird jedem Mitarbeiter das Gehalt des Chefs zugeordnet.

```
SELECT        m1.name, m1.chef, m1.gehalt, m2.gehalt
FROM            mitarbeiter m1, mitarbeiter m2
WHERE        m1.gehalt > m2.gehalt
  AND        m1.chef  =  m2.name

MITARBEITER
                                              ¿
 name           chef           gehalt gehalt
Æ                                             µ
 Frisch         Mºller           4600   4500
 Heck           Gºnther          3100   3000
```

Suchbedingung

Suchbedingungen werden in der WHERE- und HAVING-Klausel benötigt, um bestimmte Spalten zu selektieren bzw. zu gruppieren (siehe auch Abb. 3.13). Das Ergebnis einer solchen Suchbedingung ist für jede Zeile entweder WAHR, FALSCH oder UNBEKANNT (bei NULL-Werten). Nur die Zeilen, in denen die Suchbedingung WAHR ist, werden in die Ergebnisrelation übernommen.

Abbildung 3.14: Syntax der Suchbedingung

```
suchbedingung:                                                    ¿
  suchbedingung    ::= logischer term { OR logischer term }
  logischer term   ::= [ NOT ] logischer factor { AND [ NOT ] logischer factor }
  logischer faktor ::= prÐdikat | ( suchbedingung )
```

Eine solche Suchbedingung besteht meist aus einer Reihung von Prädikaten, die mit AND oder OR verbunden werden. Durch Klammerung wird eine bestimmte Abarbeitungsreihenfolge erzwungen.

Man unterscheidet sieben Arten von Prädikaten:

Vergleichsprädikat:	`A.LIEF_NR = L.LIEF_NR`
Intervallprädikat (BETWEEN):	`KPLZ BETWEEN '60000'  AND '69999'`
Ähnlichkeitsprädikat (LIKE):	`KPLZ LIKE '6%'`[6]
Test auf NULL:	`LIEF_NR IS NULL`
IN-Prädikat:	`ART_ART IN ('Monitor','Drucker')`
ALL- oder-ANY-Prädikat:	`LIEF_NR >ALL (SELECT LIEF_NR FROM ARTIKEL)`
EXISTS-Prädikat:	`EXISTS (SELECT * FROM ARTIKEL WHERE ...)`

6. Das Zeichen % zeigt an, dass an dieser Stelle in der zu überprüfenden Spalte beliebige Zeichen stehen dürfen.

```
prΣdikat:                                                          ¿

  prΣdikat ::=  vergleichsprΣdikat
              | between prΣdikat
              | like prΣdikat
              | test auf null
              | in prΣdikat
              | all or any prΣdikat
              | exists prΣdikat

  vergleichsprΣdikat    ::= scalar ausdruck  vergleichsoperator
                            scalar ausdruck | subquery

  vergleichsoperator    ::=   =  |  <>  |  <  |  >  |  <=  |  >=

  between prΣdikat      ::= scalar ausdruck  [NOT] BETWEEN
                            scalar ausdruck AND scalar ausdruck

  like prΣdikat         ::= spaltenreferenz [NOT] LIKE
                                    konstante  [ESCAPE atom]

  test auf null         ::= spaltenreferenz IS [ NOT ] NULL

  in prΣdikat           ::= scalar ausdruck [ NOT ] IN
                              subquery | konstante { , konstante }

  all oder any prΣdikat ::= scalar ausdruck vergleichsoperator
                              [ ALL | ANY | SOME ] subquery

  exists prΣdikat       ::= EXISTS subquery

  subquery              ::= ( select spezifikation )
```

Abbildung 3.15: Syntax des SQL-1-Prädikatausdrucks (spaltenreferenz und scalar-ausdruck siehe Abb. 3.11)

Während die Wirkung der meisten Prädikatarten selbsterklärend ist, bedürfen das ALL-oder-ANY-Prädikat und das EXISTS-Prädikat einer zusätzlichen Erklärung:

Das ALL-oder-ANY-Prädikat

Das **ALL-oder-ANY-Prädikat** lässt sich am besten an einem Beispiel verdeutlichen:

Wer verdient mehr als die Mitarbeiter des Herrn Frisch?

```
SELECT *
FROM   mitarbeiter
WHERE  gehalt >ALL
  (SELECT gehalt
   FROM   mitarbeiter
   WHERE  chef = 'Frisch')
```

```
                                    ¿
 name          chef          gehalt
Æ                                   µ
 Frisch        Mºller          4600
 Mºller                        4500
 Weber         Mºller          4000
```

Zunächst wird die Ergebnismenge des Subselects

```
SELECT    gehalt
FROM        mitarbeiter
WHERE    chef = 'Frisch'

       ¿
 gehalt
Æ      µ
   2100
   3200
```

errechnet. Danach werden im äußeren SELECT die Gehälter der Mitarbeiter selektiert, deren Gehalt größer ist als 2100 und 3200.

Die allgemeine Form des ALL-oder-ANY-Prädikats lautet:

```
scalar-ausdruck   quantifizierter-operator   subquery,
```

wobei `quantifizierter-operator` ein normaler Vergleichsoperator wie =, <>, <, >, <=, >= ist, gefolgt von `ANY`, `ALL` oder `SOME`. `SOME` ist nur ein anderes Wort für `ANY` mit der gleichen Wirkung. Der Vergleich wird als *WAHR* erkannt, wenn der Vergleichsoperator ohne `ALL` (bzw. `ANY`) *wahr* ist für ALLE Werte der Subquery (bzw. EINEN bei `ANY`) .

Das EXISTS-Prädikat

Die allgemeine Form des EXISTS-Prädikates lautet:

```
     EXISTS  subselect
```

Das EXISTS-Prädikat wird als WAHR erkannt, wenn der Subselect mindestens eine Zeile selektiert.

Selektiere alle Artikel, für die mindestens eine Bestellung vorliegt:

```
SELECT DISTINCT art_nr,art_bez
FROM   artikel a
WHERE  EXISTS  (SELECT *
                FROM   bestellungen b
                WHERE  a.art_nr = b.art_nr)
ORDER  BY art_nr

                      ¿
 art_nr art_bez
Æ                     µ
      1 Multisync II
      2 Multisync I
      3 Herkules
      8 Laser Printer
```

Da es bei der Verwendung des EXISTS-Prädikats unerheblich ist, welche Spalten der dem EXISTS folgende Subselect enthält, wird als Spaltenausdruck meist nur * angegeben.

3.5.2 Die DELETE-Anweisung (Zeilen löschen)

Mit der DELETE-Anweisung werden in einer Tabelle gelöscht. Folgende Syntax definiert den DELETE Befehl:

```
DELETE                                                                  ¿
  delete anweisung ::= DELETE FROM tabelle [ where klausel ]
```

Abbildung 3.16: Syntax der DELETE-Anweisung

Alle Zeilen, für die die `where-klausel` WAHR ist, werden aus der Tabelle gelöscht.

Lösche alle Monitore aus der Tabelle ARTIKEL:

```
DELETE
FROM   ARTIKEL
WHERE  ART_ART = 'Monitor'
```

Für die `where-klausel` gilt beim DELETE-Befehl (wie auch beim UPDATE-Befehl) die Einschränkung, dass sich der Subselect nicht auf Spalten der zu löschenden Tabelle beziehen darf.

So ist folgender Befehl **nicht** erlaubt:

```
DELETE
FROM        ARTIKEL
WHERE    ART_ART =
            (SELECT ART_ART
             FROM   ARTIKEL
             WHERE  ART_ART = 'Monitor')
```

3.5.3 Die UPDATE-Anweisung (Zeilen ändern)

Mit der UPDATE-Anweisung werden Zeilen in einer Tabelle geändert. Folgende Syntax definiert den UPDATE-Befehl:

```
UPDATE (SQL):                                                           ¿
  update anweisung ::= UPDATE tabelle
                       SET  wertzuweisung { , wertzuweisung }
                       [ where klausel ]

  wertzuweisung    ::= spaltenreferenz = scalar ausdruck | NULL
```

Abbildung 3.17: Syntax der UPDATE-Anweisung

Ersetze in der Tabelle ARTIKEL den Namen 'Monitor' durch 'Bildschirm'!

```
UPDATE    ARTIKEL
SET      ART_ART = 'Bildschirm'
WHERE    ART_ART = 'Monitor'
```

Für die `where-klausel` gilt beim UPDATE-Befehl (wie auch beim DELETE-Befehl) die Einschränkung, dass sich ein Subselect nicht auf Spalten der zu ändernden Tabelle beziehen darf. So ist folgender Befehl **nicht** erlaubt:

```
UPDATE    ARTIKEL
SET        ART_ART = 'Bildschirm'
WHERE    ART_ART =
           (SELECT ART_ART
            FROM   ARTIKEL
            WHERE  ART_ART = 'Monitor')
```

3.5.4 Die INSERT-Anweisung (Zeilen einfügen)

Mit der INSERT-Anweisung werden in einer Tabelle Zeilen hinzugefügt. Man unterscheidet zwei Arten von INSERT-Anweisungen:

1. Einfügen von konstanten Werten:

   ```
   INSERT INTO LIEFERANTEN
   VALUES (22, 'IBM')
   ```

2. Einfügen von Daten, die aus einer Tabelle stammen:

   ```
   INSERT       INTO    NEUE_LIEFERANTEN
                SELECT  *
                FROM    ALTE_LIEFERANTEN
   ```

Folgende Syntax definiert den INSERT-Befehl:

Abbildung 3.18: Syntax der INSERT-Anweisung

```
INSERT:                                                              ¿

  insert anweisung ::=
        INSERT INTO tabelle
            [ ( spaltenreferenz { , spaltenreferenz } ) ]
          VALUES ( konstante | NULL { , konstante | NULL } )
         | select spezifikation
```

`tabelle` spezifiziert hierbei die Zieltabelle. Wenn eine Spaltenreferenz angegeben wurde, werden alle Spalten, die nicht aufgeführt wurden, mit NULL belegt.

Beim INSERT-Befehl gilt für die Select-Spezifikation die Einschränkung, dass sie sich nicht auf die Zieltabelle beziehen darf. So ist folgender Befehl **nicht** erlaubt:

```
INSERT    INTO ARTIKEL
          SELECT *
          FROM ARTIKEL
```

3.6 Einbettung von SQL in 3GL-Sprachen

Oftmals sind in Anwendungsprogrammen Probleme zu lösen, für die die Sprache SQL aufgrund fehlender **Berechnungsuniversalität** nicht geeignet ist. Meist können solche Probleme nur durch mehr oder weniger komplexe Algorithmen (z. B. **bedingte Verzweigungen**, **Iterationen** usw.) gelöst werden.

An diesem Beispiel soll die fehlende Berechnungsuniversalität von SQL verdeutlicht werden [Neumann 92]. Angenommen, es sollen Polygone mit einer unbekannten Anzahl von Eckpunkten durch folgende Tabelle dargestellt werden:

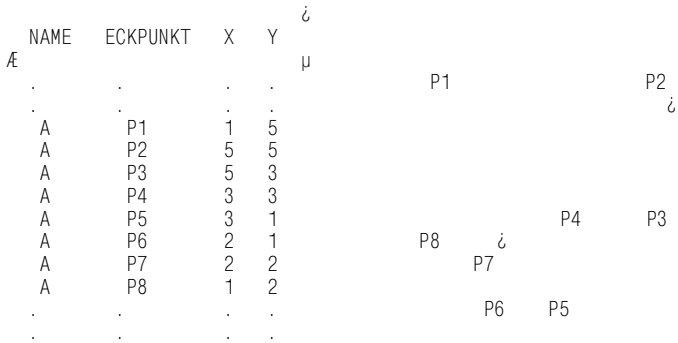

Abbildung 3.19: Darstellung eines Polygons in einer Tabelle

Dann ist die naheliegende Frage »Welchen Flächeninhalt hat Polygon A?« durch SQL nicht zu lösen, da SQL die für diese Aufgabe nötigen Schleifenkonstrukte oder Rekursionsmöglichkeiten fehlten.

SQL wurde auch nie für diese Art von Fragestellungen konzipiert. Im Gegensatz zu 3GL-Sprachen wie Java, COBOL oder C++, deren Hauptvorteil in ihrer Fähigkeit besteht, Algorithmen zu beschreiben, ist eine **nichtprozedurale Sprache** wie SQL nur sehr schwer um solche **operationalen Funktionen** zu erweitern. Um jedoch auch mit einer 3GL-Sprache mit Hilfe von SQL auf eine relationale Datenbank zugreifen zu können, müssen Möglichkeiten geschaffen werden, um SQL-Anweisungen in 3GL-Programme »einbetten« zu können. In der Regel wird vom RDBMS-Hersteller eine Schnittstelle von 3GL-Programmen zu einer relationalen Datenbank zur Verfügung gestellt. Des Weiteren existieren Standards wie ODBC (Open Database Connectivity) bzw. JDBC (Java Database Connectivity), die diese Schnittstelle standardisieren. ODBC, ursprünglich von Microsoft entwickelt (http://www.microsoft.com/data/odbc/default.htm), steht mittlerweile für verschiedene Betriebs-- und Datenbanksysteme zur Verfügung. JDBC ist die Datenbankschnittstelle für die Programmiersprache JAVA und wurde von SUN eingeführt (http://java.sun.com/products/jdbc).

3.6.1 Embedded SQL

Diese Schnittstelle von einer 3GL-Sprache zu SQL wird »**embedded SQL**« (**eingebettetes SQL**) genannt. Das folgende Programmgerüst liest aus der Programmiersprache C Daten aus der Tabelle Polygone heraus :

```
10  exec sql include sqlca;
15
20  main ()
25
```

```
30  exec sql connect to dbname;
35
40  /* deklariere einen lese-cursor */
50  exec sql declare selpoly cursor for read only;
60        select x,y
70        from  polygone
80        where name = 'A'
90        order by eckpunkt;
100
105
110 /* öffne Cursor und lese ersten Eckpunkt */
120 exec sql open selpoly;
130 exec sql fetch selpoly into x, y;
140 if (sqlca.sqlcode < 0)
150 {
160     error("Fehler beim ersten Zugriff");
170 }
180
190 /* solange weitere Eckpunkte vorhanden: lese sie und hänge sie in
Linked-List an */
200
210 while (sqlca.sqlcode != NO_MORE_ROWS)
220 {
230     appendEckpunkt(ll, x, y);
240     exec sql fetch selpoly into x, y;
250 }
260 /* Fläche des Polygons berechnen und ausgeben */
270 printf (Fläche des Polygons A: %ld\n", Fläche(ll));
280
290 exec sqlclose selpoly;
300             disconnect all;
310 exit
```

Die durch `exec sql` eingeleiteten erweiterten C-Anweisungen werden durch einen **Precompiler** (**Vorübersetzer**) in Standard-C-Unterprogrammaufrufe übersetzt. Die eigentliche Kommunikation zwischen dem C-Programm und dem Datenbanksystem findet zum einen über Parameter dieser RDBMS-Unterprogramme und zum anderen über die **Sqlca** (SQL-Communication Area) statt. Dieser SQLCA wird in dem Include-File sqlca (Zeile 10) definiert und ist programmtechnisch gesehen ein Satz globaler Statusvariablen, die vom RDBMS nach jeder SQL-Anweisung gesetzt und vom C-Programm ausgewertet werden können. Eine solche globale Variable ist **sqlcode**. Über diese Variable meldet das RDBMS nach jeder SQL-Anweisung eventuell aufgetretene Probleme, wie z.B. LOCK-Probleme, an das Programm zurück. Das Programm muss diese Variable auswerten und eine entsprechende Fehlerbehandlung einleiten (Zeile 140).

3.6.2 Das Mengenproblem

Ein besonderes Problem, das bei der Verbindung von 3GL- und mengenorientierten Sprachen zu lösen ist, entsteht dadurch, dass bei SQL das Ergebnis einer SELECT-Anweisung eine vorher nicht definierbare Menge von Tupeln liefert. Diese Menge muss einer der 3GL-Sprache bekannten Datenstruktur zugewiesen werden. Eine naheliegende Datenstruktur, die in allen verbreiteten 3GL-Sprachen zu finden ist und in der Lage ist, eine Menge von gleichstrukturierten Daten aufzunehmen, wäre das ARRAY. So wäre es möglich, durch

```
polygon_struct poplygon_arr[100];

polygon_arr = select  *
                from    polygone
                where   name = 'A'
                order   by eckpunkte
```

eine zum Programmerstellungszeitpunkt nicht bekannte Tupelmenge einem Array zuzuweisen. Ein ARRAY hat jedoch in allen Sprachen den Nachteil, dass es statisch deklariert werden muss, was bedeutet, dass die Größe des ARRAYs schon zur Übersetzungszeit des Programms bekannt sein muss. Somit müsste das ARRAY übermäßig groß definiert werdenn um einen Überlauf zu vermeiden.

In Embedded-SQL wird, um **SQL-Ergebnismengen** verarbeiten zu können, ein anderer Weg eingeschlagen. Zunächst muss ein sogenannter **CURSOR** durch die Anweisung `declare cursor` (Zeile 50) definiert werden. Dieser Cursor wird dann durch die Anweisung `open` (Zeile 120) ausgeführt. Das RDBMS übergibt dann bei jeder `fetch`**-Anweisung** (Zeile 240) ein **Ergebnistupel** in die bei FETCH aufgeführten C-Variablen. Solche C-Variablen werden **Host-Variable** genannt. Durch die Programmschleife um diesen FETCH werden somit Tupel für Tupel vom RDBMS entgegengenommen und (in diesem Beispiel) in eine Liste eingetragen.

3.6.3 Dynamisches SQL

Nicht immer ist es möglich, schon zur Zeit der Programmerstellung eine SELECT-Anweisung statisch zu programmieren. Wenn beispielsweise erst der Anwender eines Programms zur Laufzeit die WHERE-Bedingung eines SELECTs interaktiv am Bildschirm eingeben möchte, so ist ein Programm in der Form

```
READ :CONDITION;

EXEC SQL DECLARE Y FOR
      SELECT * FROM KUNDEN
      WHERE :CONDITION
            .
            .
            .
```

unabdingbar. Wenn ein SQL-Befehl erst zur Laufzeit bekannt ist, so wird diese Art des Programmierens **Dynamic-SQL** genannt. Dynamic SQL wurde erst mit dem SQL-2-Standard (siehe Kapitel 4) normiert.

3.7 Kritik an SQL

Obwohl die Sprache SQL zunächst umfassend und einfach erscheint, bereitet ihre Anwendung in der Praxis auch Probleme. So begründen anerkannte Datenbankexperten die nur sehr zögerliche Durchsetzung der relationalen Datenbanken unter anderem mit der Unvollkommenheit dieser Sprache [Date 84], [Codd 90].

So ist es z. B. mit SQL nicht möglich, die Gehälter in der Mitarbeitertabelle zu erhöhen, wenn der Umfang der Gehaltserhöhung in einer anderen Tabelle steht. In SQL können Daten einer Tabelle nicht mit Werten einer anderen Tabelle geändert werden.

Ein anderes Beispiel:

Aufgabe Ermittle die Namen aller Mitarbeiter, die älter als der Durchschnitt sind.

Lösung So ist der folgende SQL-Befehl korrekt:

```
SELECT    name
FROM        mitarbeiter
WHERE    alter > (SELECT AVG(alter) FROM mitarbeiter)
```

Dagegen ist

```
SELECT    name
FROM        mitarbeiter
WHERE    (SELECT AVG(alter) FROM mitarbeiter) < alter
```

syntaktisch falsch, da der Subselect rechts neben dem Operator stehen muss. Solche uneinsichtigen und logisch nicht erklärbaren Regeln verkomplizieren das Lernen einer Sprache.

Erst mit dem SQL-2-Standard wurden einige dieser Beschränkungen von SQL-1 behoben.