2. Relationale Datenbanken (RDB) und SQL (Structured Query Language)

2.1 Das relationale Datenbankmodell

Die Segmente einer **relationalen Datenbank (RDB)** sind **Tabellen.** Tabellen sind wie Matrizen in der linearen Algebra in Form von **Spalten** und **Zeilen** aufgebaut. Alle Zeilen haben den gleichen Spaltenaufbau. Die Zeilen entsprechen den Datensätzen, die Spalten definieren die **Attribute** der Datensätze.

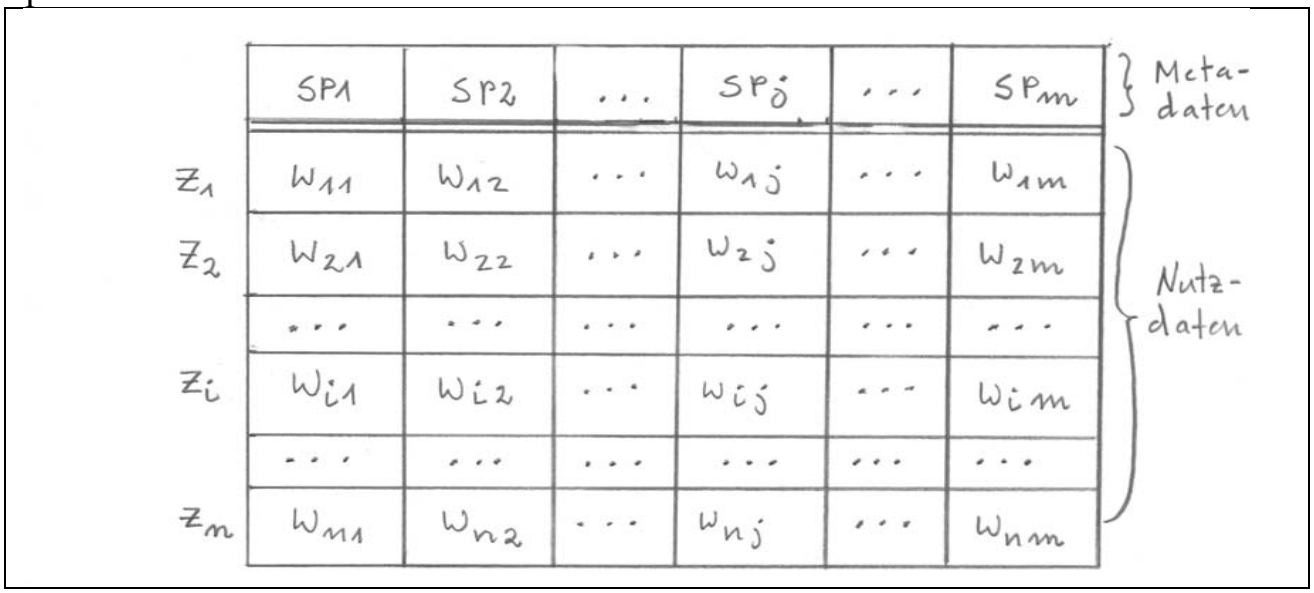

Abb.1: Aufbau einer Tabelle **T** einer RDB.

Für alle Spalten **SPj** ($1 \leq j \leq m$; $m \in \mathbb{N}$) einer Tabelle **T** werden im Schemakatalog des relationalen Datenbanksystems (RDBMS) folgende Metadaten hinterlegt:

1) Jede Spalte SPj wird durch einen **Attributnamen Aj** identifiziert.

2) Jeder Spalte Aj ist ein **Datentyp dtj** zugeordnet: dtj = dt(Aj). Im nachfolgenden Unterkapitel 2.2 wird eine Übersicht über wichtige SQL Standard-Datentypen gegeben.

3) Für jede Spalte SPj ist ein **Wertebereich Wj** festgelegt. **Wj** ist zunächst durch den Wertebereich WDT gegeben, der durch den Datentyp dtj festgelegt ist. Falls für die Spalte SPj eine **Integritätsbedingung Ij** gesetzt ist, kann dadurch der Wertebereich eingeschränkt werden. Es gilt dann: $\mathbf{Wj \subseteq WDT}$.

Def.1: Die unter 1), 2) und 3) genannten Metadaten bilden das **Relationenschema RS(T)** der Tabelle T. Man schreibt: $\mathbf{RS\ (T) = \{\ (Aj\ ,\ dtj\ ,\ Wj\ )\ |\ 1 \leq j \leq m\ \}}$. In der Formulierung des Relationenschemas kann man statt des Wertebereichs **Wj** auch die Integritätsbedingung **Ij** einsetzen: RS (T) = { (Aj , dtj , **Ij**) | $1 \leq j \leq m$ }.

Jede relationale Datenbank RDB kann als **Vereinigungsmenge** von Tabellen **Tk** angesehen werden ($1 \leq j \leq L$; $L \in \mathbb{N}$):

$$RDB = \bigcup_{k=1}^{L} Tk$$

Daher gilt für das Relationenschema der gesamten Datenbank RS(RDB):

$$RS(RDB) = \bigcup_{k=1}^{L} RS(Tk)$$

Anm.1: Jede Zeile **zi** einer Tabelle **T** ($1 \leq i \leq n$; $n \in \mathbb{N}$) kann als **m**-Tupel dargestellt werden (m = Anzahl der Spalten von T): **zi = (wi1 , wi2 , ..., wij , ... , wim).** Hierbei ist jedes **wij** ist der Wert eines Attributs **Aj** bzw. einer Spalte **SPj**. Da wir für jedes Attribut **Aj** den Wertebereich **Wj** kennen, ist jede Zeile **zi** als m-Tupel Element des **kartesischen Produkts** der Wertebereiche **W1 , W2 ,...,Wm** :

zi ∈W1 x W2 x ... x Wk

Die Tabelle T ist somit **Teilmenge des kartesischen Produkts** der Wertebereiche **Wj**. Als Formel dargestellt: **T ⊆ W1 x W2 x ... x Wk**. Damit ist T auch eine mathematische Relation. Dieses hat zur Folge, dass in der Theorie relationaler Datenbanken sowohl der Aufbau der relationalen Datenbank als auch die SQL-Befehle zum Lesen und Schreiben auf Tabellen vollständig mit Mitteln der relationalen Algebra

beschrieben werden können. Die SQL-Befehle werden dabei als relationale Operatoren spezifiziert.

2.2 SQL-Datentypen

Die Datentypen von SQL sind normiert. Jüngere Normen sind z.B. SQL:1999 und SQL:2003. Leider implementieren alle DBMS-Produkte diese Normen nicht 1:1. Wichtige Abweichungen bei Oracle™ sind nachfolgender Tabelle auch dokumentiert.

Nr.	**SQL-Datentyp**	**Zweck**	**Beispielwerte**	**Oracle™ Spezifika**
(1)	char(n)	Zeichenkette mit fester Länge n ($n \in \mathbb{N}$)	‘Haus‘	
(2)	varchar(n)	Zeichenkette mit variabler Länge, mit maximal n Zeichen	‘Karl der Große‘	varchar2(n)
(3)	clob	character large object		
(4)	integer	ganze Zahlen z ($z \in \mathbb{Z}$) mit $-2^{31} \leq z \leq 2^{31}-1$	-1035 bzw. 2345	number
(5)	float	für Gleitpunktzahlen (entspricht IEEE-8-Byte Gleitpunktzahl), d.h. für	-3.78e-2 bzw. 2561.61	

		rationale Zahlen $q \in \mathbb{Q}$		
(6)	decimal(p,q)	für rationale Festpunkt-zahlen mit insgesamt p Dezimalstellen, davon q Nachkommastellen	3170924.56 mit (p,q)=(9,2) bzw. 1.2798 mit (p,q)=(5,4)	number(p,q)
(7)	date	für Datumsangaben	‘2013-10-16‘	
(8)	time	für Zeitangaben	‘11:30:49‘	
(9)	timestamp	Kombination aus Datums- und Zeitangabe		date

2.3 Der Sprachumfang von SQL

Die Structured Query Language (SQL) ist eine genormte Anfrage-, Definition- und Transaktionskontrollsprache für relationale Datenbanken. SQL ist eine **deklarative** Programmiersprache im Unterschied zu Java oder C, die **imperative** Programmiersprachen sind.

Bei einer **imperativen** Programmiersprache wird durch eine Sequenz von Anweisungen genau beschrieben, **wie** ein bestimmtes Resultat maschinell ermittelt werden soll.

Bei einer **deklarativen** Programmiersprache wird in der Hauptsache nur spezifiziert, **welches** Resultat ermittelt werden soll. D.h. hier wird die Problembeschreibung formuliert. Wie das entsprechende Resultat ermittelt werden soll, bestimmt bei einer interpretierten deklarativen Programmiersprache wie SQL der zugehörige Kommandointerpreter. Ein Beispiel einer anderen deklarativen Programmiersprache ist PROLOG, eine Sprache zur Programmierung prädikatenlogischer Formeln.

Der **Sprachumfang** von SQL besteht im Wesentlichen aus drei Teilmengen:
a) Data Definition Language (DDL): Die DDL enthält Befehle, die auf dem Schemakatalog des DBS agieren und somit der Verwaltung von Metadaten dienen. Wichtige Befehle der DDL sind:

Befehl	**Zweck**
CREATE TABLE	Anlegen einer Tabelle. Hierbei werden alle Metadaten einer Tabelle festgelegt.
ALTER TABLE	Änderungen von Metadaten einer Tabelle.
DROP TABLE	Löschen aller Metadaten **und** aller Nutzdaten einer Tabelle.

b) Data Manipulation Language (DML): Zur DML gehören die Befehle, die zur Verwaltung der Nutzdaten einer Tabelle dienen. Hierzu gehören schreibende Befehle als auch der lesende Befehl SELECT:

Befehl	**Zweck**

INSERT	Einfügen einer neuen Zeile in eine Tabelle.
UPDATE	Ändern einzelner Spaltenwerte in einer oder mehreren Zeilen einer Tabelle.
DELETE	Löschen einer oder mehrerer Zeilen einer Tabelle
SELECT	Lesen von einer oder mehreren Zeilen einer oder mehrerer Tabellen, die logisch miteinander verknüpft sind.

c) Data Control Language (DCL): Zur DCL gehören die Befehle zur Verwaltung von Transaktionen und zur Verwaltung von Zugriffsrechten.

Befehl	**Zweck**
BEGIN	Definition einer Transaktion.
COMMIT	Änderung einer DB nach korrekter Durchführung der Transaktion.
ROLLBACK	Zurücksetzung der DB in den Zustand vor Beginn der Transaktion.
GRANT	Zuteilung von Zugriffsrechten an Nutzer.
REVOKE	Widerrufen von Zugriffsrechten.

2.4 DDL Befehle

a) Eine Syntaxbeschreibung des Befehls CREATE TABLE zum Anlegen einer Tabelle:

```
CREATE TABLE tabname
(sp1 dt1 [sp1_IntBed],
 sp2 dt2 [sp2_IntBed],
  …  …  …,
 spN dtN [spN_IntBed]
 [, TabIntBed])
```

Erläuterung:
sp1, ..., spN:Attribute (Spaltennamen);
dt1, ..., dtN: Datentyp der Spalten 1 bis N;
spi_IntBed: Integritätsbedingung der Spalte i ($1 \leq i \leq N$)
Tab_IntBed: Integritätsbedingung, die für die ganze Tabelle wirksam ist.

BSP.1: Anlegen einer Tabelle **Kunde** mit den Attributen **knr** (Kundennummer, PRIK), **knam** (Kundenname), **plz** (Postleitzahl mit der sachlichen Integritätsbedingung $1000 \leq$ **plz** ≤ 99999) und **ort** (Lieferort des Kunden, MUSS-Attribut):

```
CREATE TABLE Kunde
(knr integer PRIMARY KEY,
 knam char (30) NOT NULL,
```

```
plz integer CHECK(1000<=plz AND plz<=99999),
ort varchar(50) NOT NULL)
```

b) Eine Syntaxbeschreibung des Befehls ALTER TABLE zur Änderung einer Tabelle:

```
ALTER TABLE tabname modus SP_IntBed1 [, …, SP_IntBedK]
```

Erläuterung:
modus: Art der Änderung: ADD: hinzufügen, MODIFY: überschreiben, DROP: löschen (insbesondere: DROP COLUMN: löschen einer Spalte).
SP_IntBedi: hinzugefügte, überschriebene oder geänderte Spalte und / oder Integritätsbedingung ($1 \leq i \leq K$).

BSP.2: Hinzufügen der Spalte **strasse** (Lieferanschrift mit Straße und Hausnummer) mit Integritätsbedingung NOT NULL (MUSS-Attribut) in die Tabelle **Kunde:**

```
ALTER TABLE Kunde ADD strasse varchar(30) NOT NULL
```

BSP.3: Das Attribut **plz** der Tabelle **Kunde** mit der zusätzlichen Integritätsbedingung NOT NULL ausstatten:

```
ALTER TABLE Kunde MODIFY plz integer NOT NULL
```

BSP.4: Löschen eines Attributs **kredit** in der Tabelle **Kunde:**

```
ALTER TABLE Kunde DROP COLUMN kredit
```

c) Eine Syntaxbeschreibung des Befehls DROP TABLE zum Löschen einer Tabelle:

```
DROP TABLE tabname
```

Hinweis: Der Befehl DROP TABLE löscht eine Tabelle vollständig: Alle Nutz- und Metadaten werden gelöscht. Man sollte mit diesem Befehl besonders vorsichtig umgehen.

BSP.5: Löschen der Tabelle **Gartikel:**

```
DROP TABLE Gartikel
```

2.5 Die schreibenden DML Befehle

a) Eine Syntaxbeschreibung des Befehls INSERT zum Einfügen einer neuen Zeile in eine Tabelle:

```
INSERT INTO tabname(sp1, sp2, …, spN)
        VALUES(w1, w2, …, wN)
```

Erläuterung: Der INSERT Befehl hat zwei Klauseln: (1) Die Spaltenklausel (sp1, sp2, …, spN), die angibt, welche Spaltenpositionen der Zeile mit Werten zu füllen ist;

(2) Die Werteklausel (w1, w2, …, wN), die die Werte enthält, die gemäß Spaltenklausel einzufüllen sind.

Die Spaltenklausel ist optional. Die Werteklausel ist notwendig. Falls keine Spaltenklausel angegeben ist, müssen mit der Werteklausel alle gemäß ihrer Integritätsbedingung notwendigen Spaltenpositionen gefüllt werden. Dabei ist die Reihenfolge der Spalten gemäß der aktuellen Spaltenfolge im Tabellenschema einzuhalten. Da Abweichungen zu Fehler führen, ist die Nutzung der Spaltenklausel zu empfehlen, weil dann die Wertefolge in der Werteklausel nach der Folge in der Spaltenklausel richten muss. Die Werte müssen der Syntax für Konstanten in SQL folgen:

SQL-Datentyp	**Allgemeine Konstante**	**Anmerkung**
integer	NNN…N	N: eine Dezimalziffer
float, decimal(p,q)	aaa…a**.**bb..b	a, b: Dezimalziffern
char(n), varchar(m)	‘XXX … XXX‘	X: ein ASCII-Zeichen
date	**`TO_Date('21.10.2013','DD.MM.YYYY')`**	Oracle-spazifisch

date	Date'2013-10-23'	Allgemein

BSP.6: Einfügen einer Zeile in die Tabelle **Kunde:**

```
INSERT INTO Kunde(knr,knam,plz,ort,strasse,kredit,aedat)
VALUES(107,'Meyer GmbH',53999,'Bonn','Feuerbach 777',
1000.00, TO_Date('21.10.2013','DD.MM.YYYY'))
```

b) Eine Syntaxbeschreibung des Befehls UPDATE zur Änderung einer oder mehreren Zeilen einer Tabelle:

```
UPDATE tabname SET sp1=WA1, sp2=WA2, …, spN=WAn
        WHERE LOG_BED
```

Erläuterung: Der UPDATE Befehl hat zwei Klauseln: (1) Die SET-Klausel, die beschreibt, welche Spaltenposition **spK** ($1 \leq K \leq n$, **spK** ist der Spaltenname) mit dem Inhalt eines Wertausdrucks **WAK** zu überschreiben ist. **WAK** ist im einfachsten Falle eine Konstante. **WAK** kann aber auch gemäß des Datentyps der Spalte aus einem Spaltennamen, zulässigen Operatoren und ggfls. eines SQL-Ausdrucks zusammengesetzt sein. (2) Die WHERE-Klausel, die mit einer logischen Bedingung die Zeilenmenge einschränkt, auf die die Änderungsoperation ausgeführt wird.

BSP.7: Ändern der gesamten Anschrift eines Kunden (knr=107):

```
UPDATE Kunde SET plz=50789, ort='Koeln', strasse='Amselweg
17' WHERE KNR=107
```

BSP.7: Erhöhen der Preise aller Artikel um 5 Prozent:

```
UPDATE Artikel SET preis=1.05*preis
```

c) Eine Syntaxbeschreibung des Befehls DELETE zum Löschen einer oder mehrerer Zeilen einer Tabelle:

```
DELETE FROM tabname WHERE LOG_BED
```

Erläuterung: Mit der WHERE-Klausel des DELETE Befehls wird die Menge der Zeilen bestimmt, die zu löschen sind. Hinweis: Ist keine WHERE-Klausel angegeben, werden **alle** Zeilen der Tabelle gelöscht! Also ist Vorsicht geboten.

BSP.8: Löschen des Kunden mit **knr**=108:

```
DELETE FROM Kunde WHERE KNR=108
```

2.6 Das SELECT

Das SELECT ist der Befehl zum lesenden Zugriff auf Tabellen einer relationalen Datenbank. Das SELECT ist ein mächtiger Befehl mit einer Reihe von Klauseln.

Beschreibung des allgemeinen Aufbaus eines SELECT Befehls:

```
SELECT [selectArt] spaltenauswahl FROM tabListe
    WHERE LOG_BED
    [GROUP BY spaltenliste]
    [HAVING LOG_BED]
    [ORDER BY spaltenfolge]
```

Jede ausgeführte SELECT-Anfrage produziert eine Ergebnisliste. Die Ergebnisliste hat den Aufbau einer temporären Tabelle. Der Knotenaufbau der Ergebnisliste ist durch die **`spaltenauswahl`** bestimmt. Die Knoten sind die Zeilen der Ergebnistabelle. Man kann sagen, jede SELECT-Anfrage transformiert eine zu lesende Tabelle in eine Ergebnistabelle.

Anm.1: Eine minimale Form der SELECT-Anfrage besteht aus einer einfachen Liste von ausgewählten Spalten **sp1, sp2, …, spN** einer Tabelle **T**:

```
SELECT [selectArt] spaltenauswahl FROM T
```

BSP.9: Auf die Tabelle KUNDE, wie sie in BSP.6 und BSP.7 mit Daten gefüllt wurde, wird folgende SELECT-Anfrage gestellt:

```
SELECT KNR, KNAM, ORT, AEDAT FROM Kunde
```

Die Ergebnistabelle (Ergebnisliste) hat dann beispielsweise folgenden Aufbau:

```
107  Meyer GmbH                  Koeln         21.10.13
109  Schulz KG                   Bonn          23.10.13
110  Schmitz, Josef              Koeln         28.10.13
115  Schmitz, Josef              Leverkusen    28.10.13
```

In folgender Abbildung ist die Ergebnistabelle zum SELECT aus BSP.9 als Liste dargestellt:
< ABB.X: ERGEBNISLISTE>

BSP.10: Würde man sich nur auf die Spalten KNAM und AEDAT beschränken wollen, würde folgende SELECT-Anfrage gestellt:

```
SELECT KNAM, AEDAT FROM Kunde
```

Die Ergebnistabelle (Ergebnisliste) hat dann folgenden Aufbau:

```
Meyer GmbH                  21.10.13
Schulz KG                   23.10.13
Schmitz, Josef              28.10.13
Schmitz, Josef              28.10.13
```

Im folgenden gehen wir nun die verschiedenen Klauseln des SELECT-Befehls durch:

A) Die Klausel **`selectArt`** steuert, ob doppelte Zeilen unterdrückt werden.
Folgende Einträge sind möglich: **`selectArt := ALL|DISTINCT|UNIQUE`**
Hierbei gilt: **ALL :** Alle Zeilen werden ausgegeben. Dieses ist der Default.
DISTINCT: Doppelte Zeilen werden unterdrückt. **UNIQUE:** Synonym zu DISTINCT.

BSP.11: Im Unterschied zum BSP.10 wird mit dem folgenden Aufruf
`SELECT UNIQUE KNAM, AEDAT FROM Kunde`
folgendes Ergebnis erzielt, an man erkennen kann, dass die doppelte Ergebniszeile zum Eintrag KNAM=‘Schmitz, Josef‘ unterdrückt wurde:

```
Schulz KG                    23.10.13
Meyer GmbH                   21.10.13
Schmitz, Josef               28.10.13
```

B) Die **`spaltenauswahl`**-Klausel bestimmt, wie der Spaltenaufbau der Ergebnistabelle inhaltlich strukturiert sein soll. Die Klausel kann aus komplexen Ausdrücken aufgebaut sein. In der Beschreibung gehen wir hier den Weg von einfachen zu komplexen Ausdrücken.

B1) **`spaltenauswahl := *`** : Alle Spalten der zulesenden Tabelle **T** sind für die Ergebnistabelle ausgewählt.

BSP.12: `SELECT * FROM Kunde`
Im Ergebnis treten alle Spalten der Tabelle **Kunde** auf:

```
107 Meyer GmbH      50789 Koeln      Amselweg 17  1000 21.10.13 ET
109 Schulz KG       53999 Bonn       Windmühle 99 3000 23.10.13 HA
110 Schmitz, Josef  50111 Koeln      Hohes Tor 7  1500 28.10.13 HA
115 Schmitz, Josef  51373 Leverkusen Pulvermühle  4000 28.10.13 CH
```

B2) **`spaltenauswahl := sp1, sp2, …, spN`** : Nur die angegebenen Spalten **sp1, sp2, …, spN** der zulesenden Tabelle **T** sind für die Ergebnistabelle ausgewählt. Diese Form der Spaltenauswahl wurde bereits oben in der Anm.1, BSP.9 und BSP.10 diskutiert.

B3) **`spaltenauswahl := spAus[{, spAus}]`**: Hier besteht die Spaltenauswahl aus SQL-Ausdrücken **`spAus`** , die aus

- Spaltennamen,
- Konstanten,
- Operatoren,
- Funktionen,
- SQL-Anfragen

zusammengesetzt sein können. Weiterhin kann jeder Ausdruck über die Klausel **AS xAlias** mit einem Aliasnamen **xAlias** verbunden sein.

B3.1) Operatoren und Funktionen für numerische Datentypen:
a) Numerische Operatoren: NOP := + | - | * | /

b) Numerischer Elementarausdruck: spw1 NOP spw2 (spw1, spw2 sind Spaltennamen oder Konstanten.

BSP.13: Berechnung des Kreditlimits der Kunden in US-Dollar (1 EUR = 1.3086 USD, Stand 30.6.2013):

```
SELECT KNAM, KREDIT*1.3086 AS USKRED, ‘Stand:30.6.13’ FROM
Kunde
```

Diese Anfrage erzielt folgendes Ergebnis als CSV-Ergebnisliste. Die Spaltennamen der Ergebnistabelle sind hier mitaufgeführt:

```
"KNAM";"USKRED";"'STAND:30.6.13'"
"Meyer GmbH                    ";1308,6;"Stand: 30.6.13"
"Schulz KG                     ";3925,8;"Stand: 30.6.13"
"Schmitz, Josef                ";1962,9;"Stand: 30.6.13"
"Schmitz, Josef                ";5234,4;"Stand: 30.6.13"
```

BSP:14: Umrechnung des Kreditlimits in US-Dollar, wobei der Umrechnungsfaktor nicht als Konstante sondern durch eine eingebaute SELECT-Anfrage als Spaltenausdruck ermittelt wird.

```
SELECT knr, knam, kredit, 'EUR', kredit * (SELECT
waehfaktor From Land where waehkenn='USD') AS USkred,
'USD' FRoM Kunde
```

Diese Anfrage erzielt folgendes Ergebnis als CSV-Ergebnisliste. Die Spaltennamen der Ergebnistabelle sind hier mitaufgeführt:

```
"KNR";"KNAM";"KREDIT";"'EUR'";"USKRED";"'USD'"
117;"Steigenberger Hustensaft      ";5000;"EUR";6892;"USD"
107;"Meyer GmbH                    ";1000;"EUR";1378,4;"USD"
109;"Schulz KG                     ";3000;"EUR";4135,2;"USD"
110;"Schmitz, Josef                ";1500;"EUR";2067,6;"USD"
115;"Schmitz, Josef                ";4000;"EUR";5513,6;"USD"
```

Anm.2: Damit eine solche implizite SELECT-Anfrage funktioniert, müssen drei Bedingungen passen: a) Es darf als Operand nur **eine** Spalte in der Spaltenauswahl des eingebauten SELECT ausgewählt sein. b) Der Datentyp der ausgewählten Spalte muss zum Operator, der dem eingebauten SELECT übergeordnet ist, passen. c) Die Ergebnismenge des eingebauten SELECT darf nur aus einer Ergebniszeile bestehen.

c) Numerische Skalarfunktionen:

- Potenzfunktion: POW(a,b)
- Modulofunktion: MOD(n,m) (≡ n(mod m))
- Runden von x auf n Nachkommastellen: ROUND(x,n)
- Weitere Funktionen: EXP(x), LN(X), FLOOR(X), …

BSP:15: Runden der Preise der Tabelle Artikel auf eine Nachkommastelle:

```
SELECT artnr, artbez, round(preis,1) from Artikel
```

Diese Anfrage erzielt folgendes Ergebnis als CSV-Ergebnisliste:

```
"ARTNR";"ARTBEZ";"ROUND(PREIS,1)"
1;"Seife";1,1
```

```
2;"Brot";2,4
3;"Hustensaft";4,5
4;"";1,2
5;"Kuchen";3,5
```

B3.2) Operatoren und Funktionen für Zeichenketten:

- Substring-Funktion: substr(x,v,b) (≡ Substring der Zeichenkette in Spalte x von Position v bis Position b)
- Verkettung von Spaltenwerten: spa1||spa2

BSP:16: Anfrage auf Zeichenkettenspalten mit Verkettung und Substringbildung:

```
SELECT knr, substr(knam,1,5) as k1, ort||','||strasse as k2
from kunde
```

Diese Anfrage erzielt folgendes Ergebnis als CSV-Ergebnisliste:

```
"KNR";"K1";"K2"
117;"Steig";"Rütli,Tell Str.1"
107;"Meyer";"Koeln,Amselweg 17"
109;"Schul";"Bonn,Windmühle 99"
110;"Schmi";"Koeln,Hohes Tor 7"
115;"Schmi";"Leverkusen,Pulvermühle 1"
```

B3.3) Datumsfunktionen:

- **CURRENT** : liefert das aktuelle Datum
- **TO_DATE(‘datumsangabe‘, ‘Formatstring‘)**

B3.4) Aggregatfunktionen:

- **COUNT(*)** : Anzahl der Ergebniszeilen
- **SUM(spn)** : Summe aller Werte der Spalte **spn** (gilt nur für Spalten mit numerischem Datentyp)
- **MAX(spn)** : Maximum aller Werte der Spalte **spn**
- **MIN(spn)** : Minimum aller Werte der Spalte **spn**.

BSP:17: Anfrage auf die Artikeltabelle mit Aggregatfunktionen:

```
SELECT count(artnr), sum(preis), sum(preis)/count(artnr)
from artikel
```

Diese Anfrage erzielt folgendes Ergebnis als CSV-Ergebnisliste:

```
"COUNT(ARTNR)";"SUM(PREIS)";"SUM(PREIS)/COUNT(ARTNR)"
5;12,68;2,536
```

C) Die **WHERE**-Klausel

Die WHERE-Klausel schränkt mit einer logischen Bedingung **LOG_BED** die Menge der Ergebniszeilen einer SELECT-Anfrage ein. Eine logische Bedingung ist aus **logischen Operatoren** und **Vergleichsausdrücken** aufgebaut. Vergleichsausdrücke haben die allgemeine Form: **SPX VOP VWERT**. Hierbei ist **SPX** ein Spaltenname, **VOP** ein Vergleichsoperator und **VWERT** ein Vergleichswert. Vergleichswerte können Konstanten oder SQL-Ausdrücke sein (vgl. B3)).

C.1) Die logischen Operatoren sind: **AND**, **OR**, **NOT** .
C.2) Numerische Vergleichsoperatoren: =, !=, >=, <=, >, < .
C.3) Vergleichsoperatoren für Zeichenketten: =, !=, LIKE
C.4) Vergleichsoperatoren für NULL-Werte: IS NULL, IS NOT NULL
C.5) Mengenwertiger Vergleichsoperator: IN (mathematisch: x Element einer Menge M).

Anm.3: Der LIKE-Operator vergleicht den Inhalt einer Spalte **spx** arbeitet mit einer Zeichenkette **vgl**, die als Vergleichsmuster dient: **spx LIKE vgl**
Innerhalb des Vergleichsmusters können folgende Ersatzzeichen (Jokerzeichen, bzw. Wildcards) verwendet werden: **%** : für einen beliebigen String, **?**: für ein beliebiges Zeichen.

Anm.4: Der IN-Operator prüft, ob der Inhalt einer Spalte **spx** in einer Vergleichsmenge **M** vorkommt: **spx IN M**
Die Menge M kann dabei (a) als Menge von Konstanten oder (b) als Ergebnismenge eines eingebetteten SELECT angegeben werden:
zu (a): `select * from kunde where branche IN ('HA', 'CH')`
zu (b): s. BSP.20.

BSP:18: Gesucht werden die Kunden des PLZ-Bereichs 5, deren Name mit „Sch“ beginnt:

```
SELECT * from kunde where 50000<=plz and plz<60000 and
knam LIKE 'Sch%'
```

Diese Anfrage erzielt folgendes Ergebnis als CSV-Ergebnisliste:

```
"KNR";"KNAM";"PLZ";"ORT";"STRASSE";"KREDIT";"AEDAT";"BRANCHE"
109;"Schulz KG      ";53999;"Bonn";"Windmühle 99";3000;23.10.13;"HA"
110;"Schmitz, Josef ";50111;"Koeln";"Hohes Tor 7";1500;28.10.13;"HA"
115;"Schmitz, Josef ";51373;"Leverkusen";"Pulvermühle
1";4000;28.10.13;"CH"
```

BSP:19: Anfrage auf NULL-Werte in der Artikeltabelle:

```
SELECT * from artikel where artbez is null
```

Diese Anfrage erzielt folgendes Ergebnis als CSV-Ergebnisliste:

```
"ARTNR";"ARTBEZ";"PREIS"
4;"";1,23
```

BSP.20: Lesen aller Kunden, die aus einem EU-Land kommen. Hierbei wird die Vergleichsmenge für die Elementbeziehung durch ein SELECT auf die Tabelle **Land** erzeugt, wobei nur die Länder ausgewählt werden, die zur Ländergruppe ‘EU‘ gehören:

```
SELECT * FROM Kunde WHERE kland IN (SELECT LKURZ FROM Land
WHERE LGR = ‘EU’)
```

D) Die **GROUP BY**-Klausel

Eine **Gruppe** in SQL ist eine Teilmenge **U** von Zeilen einer Tabelle **T**, die bezüglich einer Spalte **spx**, dem sog. Gruppenbegriff, alle den gleichen Wert haben. Hat **T n** Zeilen und hat **spx k** verschiedene Werte mit $k \leq n$, so gilt: $\mathbf{U1} \cup \mathbf{U2} \cup ... \cup \mathbf{Uk} = \mathbf{T}$. Im nachfolgenden Beispiel hat die Kundentabelle n=5 Zeilen und k=3 verschiedene Werte in der Spalte branche. Dadurch entstehen drei Gruppen, die mit Aggregatfunktionen ausgewertet werden können.

BSP:21: Anfrage der Kundentabelle, die nach der Spalte **branche** gruppiert wird:

```
SELECT branche, count(knr), sum(kredit) from kunde Group by branche
```

Diese Anfrage erzielt folgendes Ergebnis als CSV-Ergebnisliste:

```
"BRANCHE";"COUNT(KNR)";"SUM(KREDIT)"
"CH";2;9000
"HA";2;4500
"ET";1;1000
```

E) Die **HAVING**-Klausel

Mit der WHERE-Klausel konnte die Anfrage auf der Ebene von Zeilen einer Tabelle eingeschränkt werden. Mit der HAVING-Klausel können logische Bedingungen für Gruppen aufgestellt werden. In der Ergebnisliste sind dann nur die Gruppen vertreten, die die logische Bedingung **log_Bed** erfüllen. Allg. Syntax: **HAVING log_Bed**

BSP:22: Anfrage auf die Gruppen der Kundentabelle, die mindestens 2 Elemente haben:

```
SELECT branche, count(knr), sum(kredit) from kunde Group
by branche HAVING COUNT(knr)>=2
```

Diese Anfrage erzielt folgendes Ergebnis als CSV-Ergebnisliste:

```
"BRANCHE";"COUNT(KNR)";"SUM(KREDIT)"
"CH";2;9000
"HA";2;4500
```

F) Die **ORDER BY**-Klausel

Die ORDER BY Klausel dient der Sortierung der Ergebnisliste nach Spalten. Es kann jeweils aufsteigend (ASC (engl.: ascending)) bzw. absteigend (DESC (engl.: descending)) sortiert werden. ASC ist der Default. Die Reihenfolge in der Spaltenliste bestimmt die Sortierproirität. Allg. Syntax: **ORDER BY Spaltenliste**

BSP:23: Die Ergebnisliste wird aufsteigend nach Kundenname sortiert. Gibt es mehrere Kunden gleichen Namens wird absteigend nach PLZ sortiert:

```
SELECT knr, knam, plz, kredit from kunde order by knam,
plz DESC
```

Diese Anfrage erzielt folgendes Ergebnis als CSV-Ergebnisliste:

```
"KNR";"KNAM";"PLZ";"KREDIT"
107;"Meyer GmbH                    ";50789;1000
115;"Schmitz, Josef                ";51373;4000
110;"Schmitz, Josef                ";50111;1500
109;"Schulz KG                     ";53999;3000
117;"Steigenberger Hustensaft      ";4411;5000
```

H) Die **Mehrtabellen-Verarbeitung (JOIN)**

Bei einem **JOIN** wird mit SELECT auf **mehrere Tabellen** zugegriffen. Insbesondere hat man damit die Möglichkeit, Datensätze gemäß der logischen Verknüpfung zwischen Tabellen, die durch Fremdschlüsselbeziehungen gegeben sind, aufzubereiten. Das JOIN erzeugt aus den Tabellen, die in der Tabellenliste gegeben sind, ein **kartesisches Produkt**. Beim **natürlichen JOIN** wird dieses kartesische Produkt durch eine WHERE-Klausel, in der die FKEY-Bedingungen gesetzt sind, eingeschränkt.

BSP:24: Gegeben sind die Tabellen KUNDE und ARTIKEL. Wenn man das Kaufverhalten eines Kunden in der Form „**1** Kunde kauft **n** Artikel mit einer jeweiligen Menge **mn**“ benötigt man eine Referenztabelle KUKAUFTA, die für jeden Kaufvorgang eines Artikels eine Zeile anlegt und pro Zeile über eine FKEY-Beziehung auf **knr** und eine FKEY-Beziehung auf **artnr** die logische Verknüpfung zur Kunden- und zur Artikeltabelle verwaltet. Das Relationenschema dieser Tabelle wird nachfolgend in Form eines CREATE TABLE Befehls gegeben:

```
CREATE TABLE KUKAUFTA
   (KKANR integer NOT NULL,
    KNR integer NOT NULL,
    ARTNR integer NOT NULL,
    MENGE integer NOT NULL,
    CONSTRAINT KUKAUFTA_PK PRIMARY KEY(KKANR)
```

```
    CONSTRAINT KUKAUFTA_FK1 FOREIGN KEY ("ARTNR")
      REFERENCES ARTIKEL(ARTNR),
     CONSTRAINT KUKAUFTA_FK2 FOREIGN KEY ("KNR")
      REFERENCES KUNDE(KNR)
  )
```

Das **JOIN**, mit dem nun die Verknüpfung der drei Tabellen KUNDE, ARTIKEL und KUKAUFTA hergestellt wird, erzeugt das kartesische Produkt:

```
KUNDE x ARTIKEL X KUKAUFTA
```

Hier wird ein **natürlicher JOIN** aufgebaut, in dessen WHERE-Klausel die FKEY-Bedingungen auf **artnr** und **knr** enthalten sind:

```
SELECT K.knr, substr(knam,1,12), A.artnr, artbez, menge,
       preis*menge as wert
FROM kunde K, Artikel A, KUKAUFTA B
WHERE k.knr=b.knr AND a.artnr=b.artnr
```

Diese Anfrage erzielt folgendes Ergebnis als CSV-Ergebnisliste:

```
"KNR";"SUBSTR(KNAM,1,12)";"ARTNR";"ARTBEZ";"MENGE";"WERT"
109;"Schulz KG   ";1;"Seife";20;22
109;"Schulz KG   ";5;"Kuchen";30;105
109;"Schulz KG   ";3;"Hustensaft";1;4,5
107;"Meyer GmbH  ";2;"Brot";11;25,85
```

```
110;"Schmitz, Jos";1;"Seife";5;5,5
110;"Schmitz, Jos";2;"Brot";5;11,75
110;"Schmitz, Jos";5;"Kuchen";5;17,5
```

Anm.5: Ein JOIN ist in der Regel sehr RAM beanspruchend, da im Hintergrund immer ein kartesisches Produkt aufgebaut wird. Im obigen Beispiel hat die Ergebnismenge des natürlichen JOIN 7 Ergebniszeilen. Das Mengengerüst war dabei: 5 Kunden, 5 Artikel und 7 KUKAUFTA-Zeilen. Im Hintergrund hatte das kartesische Produkt daher 175 Zeilen. Für DBS mit großen Tabellen sollten nach Möglichkeit JOINs nur eingeschränkt benutzt werden.

Anm.6: Neben der klassischen Syntax, die wie in BSP.24 die Beziehung des kartesischen Produkts modelliert, gibt es in SQL auch die JOIN- ON-Syntax, in der die Verknüpfung mit der Referenztabelle in einer ON-Klausel hergestellt wird. Die nachfolgende INNER-JOIN-ON-SELECT-Anfrage erzielt das gleiche Ergebnis, wie die SELECT-Anfrage aus BSP.24:

```
Select kunde.knr, substr(knam,1,12), A.artnr, artbez,
    menge, preis*menge as wert
from kunde , Artikel A
Inner JOIN KUKAUFTA B ON a.artnr=B.artnr
where kunde.knr=B.knr
```

Lernziele zu Kap.2: Relationale Datenbanken und SQL

1. **Den allgemeinen Aufbau einer Tabelle und die mathematischen Begriffe des kartesischen Produkts und der Relation erklären können.**
2. **Die Metadaten einer Tabelle als Relationenschema beschreiben können.**
3. **Wichtige SQL-Datentypen kennen.**
4. **Den Sprachumfang von SQL erläutern können.**
5. **Die Aufgaben und den Aufbau der DDL-Befehle CREATE TABLE, ALTER TABLE und DROP TABLE bennenen können.**
6. **Die Syntax der DML-Befehle INSERT, UPDATE und DELETE kennen.**
7. **Das SELECT-Kommando mit seinen Klauseln kennen.**
8. **Erörtern können, welche Formen einer Mehrtabellenverarbeitung es in einem SELECT geben kann.**