

Gliederung

1.	Einführung
2.	Datenbankentwurf
3.	Datenbankimplementierung
4.	Physische Datenorganisation
5.	Anfrageoptimierung
6.	Transaktionsverwaltung
7.	Datensicherheit und Wiederherstellung
8.	Business Intelligence

Gliederung

3.

Datenbankimplementierung



DBMS und SQL



SQL – Sprachelemente inkl. Datenintegrität



Sichten

DBMS und SQL

- ▶ Der Sinn einer Datenbank ist es nicht nur, Daten in **strukturierter Form** zu speichern, sondern auch Möglichkeiten zu bieten, diese **Daten zu bearbeiten, auszugeben und auswertbar** zu machen.
- ▶ Ein DBMS stellt unterschiedliche **Werkzeuge** bereit, mit welchen eine oder mehrere Datenbanken **erstellt, mit Daten gefüllt und verwaltet** werden können. So verfügt ein DBMS über mindestens eine **Benutzerschnittstelle**.
- ▶ Jede Benutzerschnittstelle bietet dem Benutzer zwei Möglichkeiten zur **Interaktion mit dem System**:
 - **Die grafische Oberfläche** bestehend aus unterschiedlichen Masken oder
 - Das direkte Entgegennehmen der Befehle (**SQL Editor**)
- ▶ Handelt es sich bei dem DBMS um ein relationales DBMS (RDBMS), das **SQL (Structured Query Language)** unterstützt, dann nimmt diese Schnittstelle **SQL-Befehle in Form von Texten / Strings entgegen**. Die SQL-Befehle werden an das DBMS gesandt und von diesem entsprechend eines Ablaufplans verarbeitet bzw. gegen die aktuelle Datenbank ausgeführt.
- ▶ SQL wurde durch das **ANSI** (American National Standard Institute) und durch das ISO (International Organization for Standardization) genormt und als Standarddatenbanksprache zur Definition, Abfrage und Manipulation von Daten in relationalen Datenbanken erklärt (aktuelle Version: SQL-3 oder SQL-99). Je nach Produkt (DBMS) gibt es die verschiedensten SQL Dialekte bzw. Erweiterungen.

DBMS und SQL

Entwicklung des ANSI Standards:

(vgl. <https://de.wikipedia.org/wiki/SQL>)

Chronologie [[Bearbeiten](#) | [Quelltext bearbeiten](#)]

- etwa 1975: *SEQUEL* = *Structured English Query Language*, der Vorläufer von *SQL*, wird für das Projekt *System R* von *IBM* entwickelt.
- 1979: *SQL* gelangt mit *Oracle V2* erstmals durch *Relational Software Inc.* auf den Markt.
- 1986: *SQL1* wird von *ANSI* als Standard verabschiedet.
- 1987: *SQL1* wird von der *Internationalen Organisation für Normung* (ISO) als Standard verabschiedet und 1989 nochmals überarbeitet.
- 1992: Der Standard *SQL2* oder *SQL-92* wird von der ISO verabschiedet.
- 1999: *SQL3* oder *SQL:1999* wird verabschiedet. Im Rahmen dieser Überarbeitung werden weitere wichtige Features (wie etwa *Trigger* oder rekursive Abfragen) hinzugefügt.
- 2003: *SQL:2003*. Als neue Features werden aufgenommen *SQL/XML*, Window functions, Sequences.
- 2006: *SQL/XML:2006*. Erweiterungen für *SQL/XML*^[2].
- 2008: *SQL:2008* bzw. ISO/IEC 9075:2008. Als neue Features werden aufgenommen *INSTEAD OF-Trigger*, *TRUNCATE-Statement* und *FETCH Klausel*.
- 2011: *SQL:2011* bzw. ISO/IEC 9075:2011. Als neue Features werden aufgenommen „Zeitbezogene Daten“ (*PERIOD FOR*). Es gibt Erweiterungen für Window functions und die *FETCH Klausel*.
- 2016: *SQL:2016* bzw. ISO/IEC 9075:2016. Als neue Features werden aufgenommen *JSON* und „row pattern matching“.
- 2019: *SQL/MDA:2019*. Erweiterungen für einen Datentyp „mehrdimensionales Feld“.

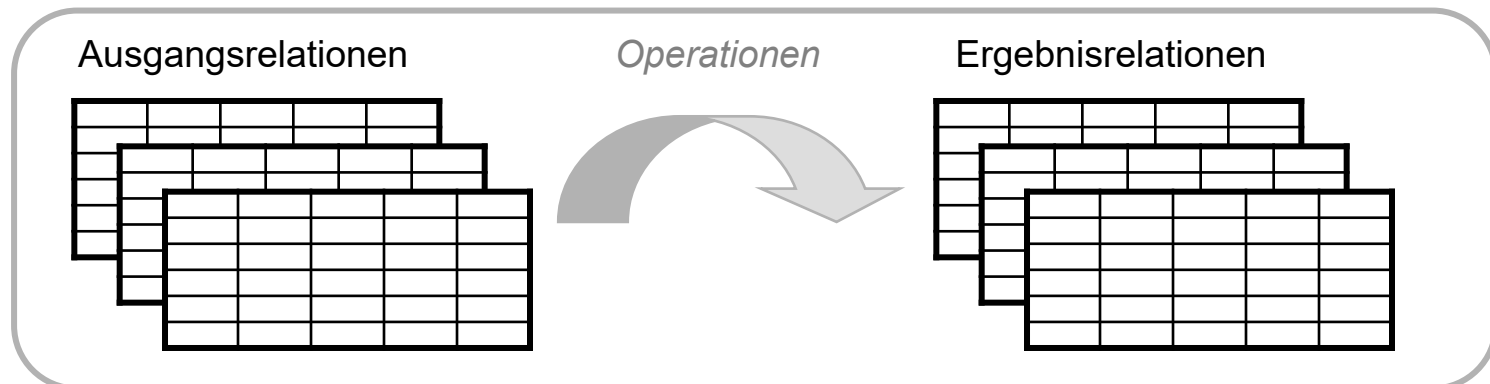
DBMS und SQL

SQL ist eine **deklarative** Anfragesprache, d. h. die Benutzer geben nur an, **welche** Daten sie interessieren, und nicht wie die Auswertung der Daten vorgenommen wird (physische Datenunabhängigkeit, Aufgabe der Anfrageoptimierer).

Die theoretischen Grundlagen für die Abfragesprache SQL bietet die **relationale Algebra**. Sie bildet den formalen Rahmen für die relationalen Datenbanksprachen indem sie einen Satz von algebraischen Operatoren definiert, die immer auf Relationen wirken.

Mathematik: Algebra ist definiert durch Mengen (Wertebereiche) sowie darauf definierten Operationen.

Mit Hilfe dieser Operatoren werden Operationen definiert, die sich auf einer Menge von Relationen (mengenorientierte Verarbeitung) anwenden lassen. Damit lassen sich beispielsweise Relationen verknüpfen, filtern oder umbenennen. Operationen lassen sich (fast) beliebig kombinieren. Die Ergebnisse aller Operationen sind ebenfalls Relationen.



Vorgehensweise bei der Implementierung einer Datenbank

1.

Datenbank anlegen



2.

Datenbank auswählen



3.

Tabellen definieren



Struktur der Tabellen festlegen (Spalten, Datentypen, Constraints)

4.

Tabellen mit Daten füllen



5.

Auswerten der Daten oder Ausführung weiterer Aktionen



Weitere mögliche Aktionen: Daten ändern oder löschen

Einteilung der SQL - Sprachelemente



DDL - Befehle

Datenbank

CREATE DATABASE:

Anlegen einer Datenbank

CONNECT DATABASE:

Verbindung zu einer Datenbank aufbauen

SP_DATABASES:

Auflistung von vorhandenen Datenbanken

SP_HELPDB:

Auflistung von Informationen zu einer / allen Datenbank/en

SP_RENAMEDB:

Ändern den Namen einer Datenbank

USE:

Auswahl einer Datenbank

DISCONNECT DATABASE:

Verbindung zu einer Datenbank abbauen

DROP DATABASE:

Löschen einer Datenbank

Tabelle

SP_HELP:

Auflistung von Informationen zu einem DB-Objekt

SP_RENAME:

Ändern den Namen eines DB-Objekts

CREATE TABLE:

Anlegen einer Tabelle

ALTER TABLE:

Ändern von Struktur einer Tabelle

DROP TABLE:

Löschen einer Tabelle

Virtuelle Tabelle

CREATE VIEW:

Anlegen einer Benutzersicht

DROP VIEW:

Löschen einer Benutzersicht

DCL - Befehle

Berechtigung

GRANT:
REVOKE:

Festlegen von Berechtigungen
Aufheben von Berechtigungen

DML - Befehle

Datensätze von Tabelle(n)

UPDATE:

Ändern von Werten in einem oder mehreren
Datensätzen

INSERT:

Aufnahme neuer Datensätze

DELETE:

Vollständiges Entfernen von vorhandenen
Datensätzen, meist mit Bedingung

DQL - Befehle

Datensätze von Tabelle(n)

SELECT:

Lesen von kompletten Datensätzen oder Teilen
davon, die meist vorgegeben Bedingungen
entsprechen müssen

Datentypen

▶ Bei der Deklaration einer Tabellenspalte muss dieser ein Datentyp zugeordnet werden.

Mit der Eingabe eines Datentyps werden

- der datentypspezifische **Wertebereich** der Werte und
- die **zulässigen Operationen** mit diesen Werten festgelegt.

▶ **SQL Server** stellt eine Reihe von (System-) Datentypen zur Verfügung, die nach folgenden **Kategorien** organisiert sind:

1. **Genaue numerische Werte**
2. **Ungefähr numerische Werte**
3. **Datum und Zeit**
4. **Zeichenfolgen**
5. **Unicode-Zeichenfolgen**
6. **Binärzeichenfolgen**
7. **Andere Datentypen**

▶ Es ist möglich auch Benutzerdefinierte Datentypen in Transact-SQL zu definieren. Aliasdatentypen basieren auf den vom System bereitgestellten Datentypen.

Datentypkategorien von SQL Server

Datentyp	Beschreibung / Bedeutung	Speichergröße
1. Exakte Zahlendatentypen für ganzzahlige Daten		
bigint	Ganze Zahl zwischen -9,223,372,036,854,775,808 und 9,223,372,036,854,775,807	8 Byte
int	Ganze Zahl zwischen - 2.147.483.648 und 2.147.483.648	4 Byte
smallint	Ganze Zahl zwischen -32.768 und 32.767	2 Byte
tinyint	Ganze Zahl zwischen 2 und 255	1 Byte
bit	0 oder 1 (true or false)	1 – 2 Byte
Numerische Datentypen mit fester Genauigkeit und fester Anzahl an Dezimalstellen		
decimal [(p [, s])]	Gepackte Dezimalzahl	depends on precision (5-17 Byte)
numeric [(p [, s])]	Synonym zu decimal	depends on precision (5-17 Byte)
Datentypen zur Darstellung von Währungswerten		
money	Float mit 4 Dezimalstellen	8 Byte
smallmoney	Float mit 2 Dezimalstellen	4 Byte
2. float [(n)]		
	Fließkommazahl mit 15 Stellen Genauigkeit	depends on precision (4-8 Byte)
real [(n)]	Fließkommazahl mit 7 Stellen Genauigkeit	4 Byte

Datentypkategorien von SQL Server

Datentyp	Beschreibung / Bedeutung	Speichergröße
3. date	Datum YYYY-MM-DD (0001-01-01 bis 9999-12-31), diverse Datenformate	3 Byte
time [(fsp)]	Zeit hh:mm:ss[.nnnnnnn], (00:00:00.0000000 bis 23:59:59.9999999)	5 Byte
datetime [(fsp)]	Datum und Zeit YYYY-MM-DD hh:mm:ss [.nnn] (1.1.1753 bis 31.12.9999 und 00:00:00.000 bis 23:59:59.997)	8 Byte
datetime2 [(fsp)]	Datum und Zeit YYYY-MM-DD hh:mm:ss [.nnnnnnn] (0001-01-01 bis 9999-12-31 und 00:00:00.0000000 bis 23:59:59.9999999)	G < 3: 6 Byte; G 4 – 5: 7 Byte; G 5 >: 8 Byte
smalldatetime [(fsp)]	Datum und Zeit YYYY-MM-DD hh:mm:ss [.nnn] (1.1.1753 bis 31.12.9999 und 00:00:00.000 bis 23:59:59.997)	8 Byte
datetimeoffset	Datum und Zeit (und Zeitzone) YYYY-MM-DD hh:mm:ss [.nnnnnnn] [{+ -}hh:mm] (0001-01-01 bis 9999-12-31 und 00:00:00.0000000 bis 23:59:59.9999999 und -14:00 bis +14:00)	10 Byte

fsp: fractional second precision = Sekundenbruchteile (n) mit Parameterangabe (1-7)

Datentypkategorien von SQL Server

Datentyp	Beschreibung / Bedeutung	Speichergröße
4. char [(n)]	Text fester Länge 1 bis 8.000 Bytes	n Byte
varchar [(n max)]	Text variabler Länge 1 bis 8.000 Bytes	n + 2 Byte
text	Zeichendaten bis 2 hoch 31 - 1 Zeichen	n Byte
5. nchar [(n)]	Unicode-Text fester Länge 1 bis 4.000 Bytes	n * 2 Byte
nvarchar [(n max)]	Unicode-Text variabler Länge 1 bis 8.000 Bytes	n * 2 + 2 Byte
ntext	Unicode-Zeichendaten bis 2 hoch 30 - 1 Zeichen	n * 2 Byte
6. binary [(n)]	Binäre Daten fester Länge 1 bis 8.000 Bytes	n Byte
varbinary [(n max)]	Binäre Daten variabler Länge 1 bis 8.000 Bytes	n Byte + 2
image	Binärdaten variabler Länge 0 bis 2.147.483.647 Bytes	
7. cursor	Zum Speichern eines Verweises auf einen Cursor enthalten	
sql_variant	Zum Speichern der Werte verschiedener Datentypen (int, binary, char)	
table	Zum Speichern eines Resultsets in Form einer Tabelle für die spätere Verarbeitung	
timestamp	Eindeutiger Zeitstempel in Datenbank	
uniqueidentifizier	Global eindeutiger Bezeichner	
hierarchyid	Zur hierarchischen Darstellung von Beziehungen zwischen den Zeilen	
xml	Zum Speichern von XML-Daten	

Datentypkonvertierungen

Zu:																										
Von:	binary	varbinary	char	varchar	nchar	nvarchar	datetime	smalldatetime	decimal	numeric	float	real	bigint	int(INT4)	smallint(INT2)	tinyint(INT1)	money	smallmoney	bit	timestamp	uniqueidentifier	image	ntext	text	sql_variant	xml
binary	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
varbinary	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
char	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
varchar	●	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
nchar	●	●	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
nvarchar	●	●	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
datetime	●	●	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
smalldatetime	●	●	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
decimal	○	○	○	○	○	○	○	○	★	★	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
numeric	○	○	○	○	○	○	○	○	○	★	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
float	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
real	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
bigint	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
int(INT4)	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
smallint(INT2)	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
tinyint(INT1)	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
money	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
smallmoney	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
bit	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
timestamp	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
uniqueidentifier	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
image	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
ntext	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
text	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	
sql_variant	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	
xml	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	

● Explizite Konvertierung

○ Implizite Konvertierung

○ Konvertierung nicht zulässig

★ Erfordert einen expliziten CAST-Operator, um den Verlust von Genauigkeit oder Dezimalstellen zu vermeiden, der bei einer impliziten Konvertierung auftreten kann.

● Implizite Konvertierungen zwischen XML-Datentypen werden nur unterstützt, wenn Quelle oder Ziel untypisierter XML-Code ist. Andernfalls muss explizit konvertiert werden.

Die Abbildung führt alle expliziten und impliziten Datentypkonvertierungen in T-SQL auf.

Die Funktionen CAST und CONVERT konvertieren einen Ausdruck explizit von einem Datentyp in einen anderen.

CAST (expression AS data_type [(length)])

CONVERT (data_type [(length)] , expression [, style])

Implizite Konvertierungen sind Konvertierungen, die ohne Angabe der CAST- oder CONVERT-Funktion durchgeführt werden.

```
SELECT 'The price is ' + CAST(APreis AS VARCHAR(20))
FROM Artikel
```

```
SELECT 'The price is ' + CONVERT(VARCHAR(20), APreis)
FROM Artikel
```

Tipps

Groß- / Kleinschreibung ist beim Schreiben von SQL – Anweisungen irrelevant. Wenn man sie verwendet, dann nur um eine bessere Lesbarkeit zu erreichen.

Ebenso verhält es sich mit **Zeilenumbrüchen und Einrückungen**, die Sie so einsetzen können, wie Sie möchten, um die Übersicht zu bewahren.

Nutzen Sie auch die Möglichkeit, **Kommentare** zwischen den Anweisungen einzugeben:

Einzeilige Kommentare:

Mehrzeilige Kommentare:

Einleitung mit zwei Bindestrichen (- -)

Kommentarbeginn mit / *

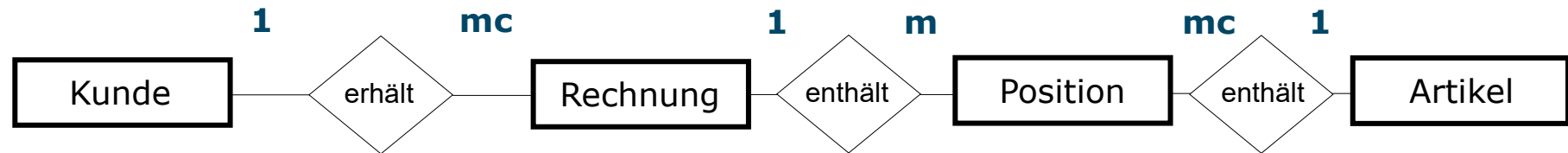
Kommentarende mit */

Kommentare werden im Abfrage-Editor in grüner Schrift dargestellt.

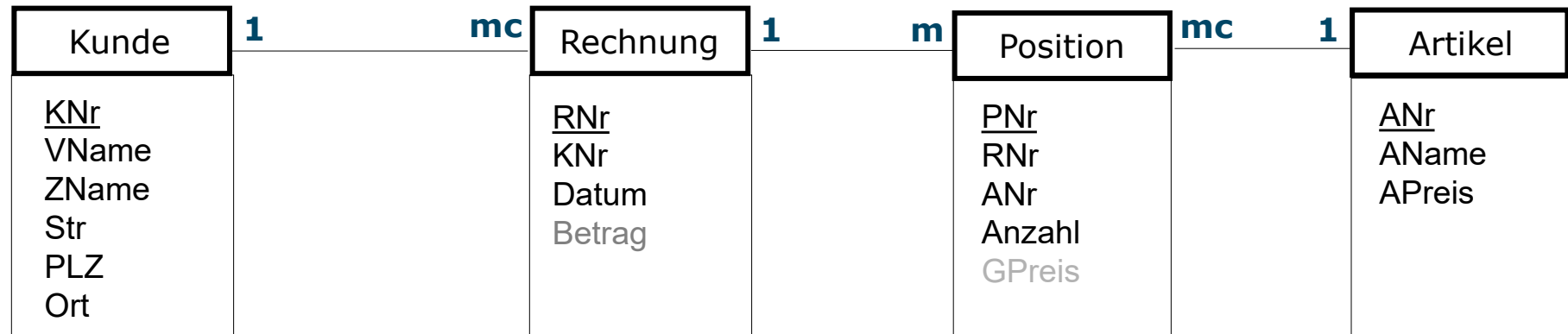


Rechnungsbeispiel

Darstellung als Entity-Relationship-Diagramm (ERM – Diagramm)



Darstellung als Relationales Datenmodell (RM - Diagramm)



Rechnungsbeispiel

Die eigentlichen Tabellen(inhalte) zu implementieren...

Kunde

<u>KNr</u>	VName	ZName	Str	PLZ	Ort
K001	Hugo	Müller	Gartenstr. 4a	69123	Heidelberg
K002	Georg	Mayer	Neckarstr. 1	69123	Heidelberg

Artikel

<u>ANr</u>	AName	APreis
A001	Computer	5.000
A002	Drucker	1.000
A003	Kabel	500

Rechnung

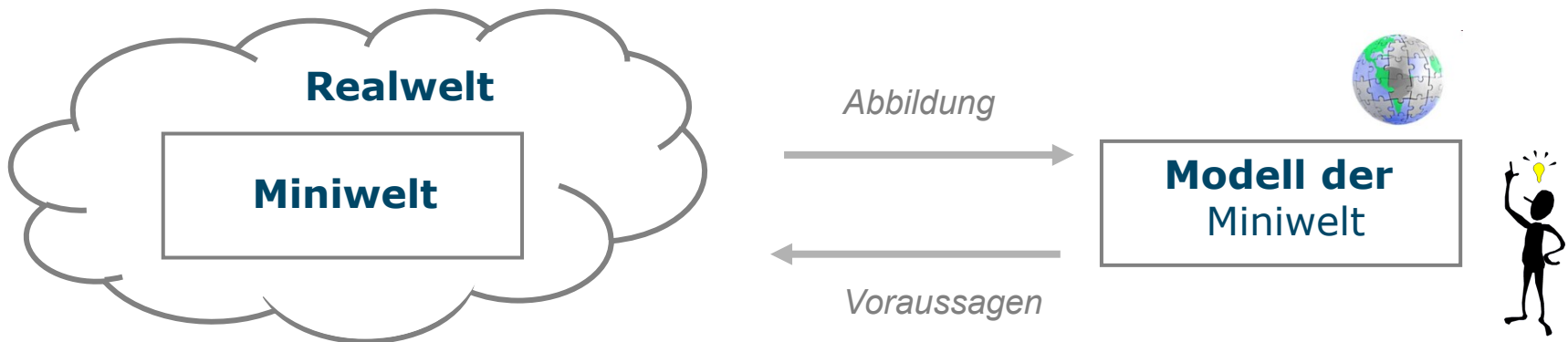
<u>RNr</u>	KNr	Datum	Betrag
R001	K001	04.04.2004	13.000
R002	K001	05.04.2004	2.000
R003	K002	05.04.2004	5.000

Position

<u>PNr</u>	RNr	ANr	Anzahl	GPreis
P001	R001	A001	2	10.000
P002	R001	A002	3	3.000
P003	R002	A002	1	1.000
P004	R002	A003	2	1.000
P005	R003	A001	1	5.000

Definition Datenintegrität

- ▶ Mit fehlender Datenintegrität bezeichnet man den Sachverhalt, dass aus einem schlecht modellierten Informationssystem nicht zu jedem Zeitpunkt **zuverlässigen Aussagen** über die abgebildete Miniwelt gewonnen werden können. Schuld hieran ist die mit der Miniwelt nicht übereinstimmende Informations-**Struktur** des fehlerhaft modellierten Informationssystems.



Implementierung von Datenintegrität ...

- ▶ Beim Entwurf und bei der Implementierung einer Datenbankanwendung ist die Erarbeitung von **Relationenschemata** (Datenmodells) eine zentrale Aufgabe. Hier werden die Strukturen der zu speichernden Daten festgelegt.
- ▶ Der Benutzer profitiert von einer guten Datenstruktur, indem er Daten ohne Redundanzen erfassen kann und auf inkonsistente Dateneingaben vom Datenbanksystem automatisch hingewiesen wird.

...beim Anlegen von Tabellen

Implementierung von Datenintegrität beim Anlegen von Tabellen

Die Integrität der Daten wird **auf Spaltenebene** beim Deklarieren von Tabellen festgelegt und kann in drei verschiedene Arten unterteilt werden:



Entitätenintegrität

*Jede Zeile einer Tabelle einer Datenbank entspricht einer Entität. Die **Zeile** einer Tabelle weist eine Entitätenintegrität auf, wenn sie vollständig **mit der Entität übereinstimmt**, die sie abbildet.*



Domänenintegrität

*Als Datenbankentwickler kann man nicht garantieren, dass ein bestimmtes Datenelement in einer Datenbank korrekt ist, aber man kann dafür sorgen, dass es einen **gültigen Wert** hat.*



Referentielle Integrität

*Die Beziehungen zwischen den Tabellen sind nicht bidirektional. In der Regel hängt eine Tabelle von einer anderen ab. Beispiel: Tabelle AUFTRAG hängt von der Tabelle KUNDE ab. Diese Art von Beziehung wird häufig als Eltern/Kind-Beziehung bezeichnet, wobei KUNDE die Eltern-Tabelle und AUFTRAG die Kind-Tabelle ist. Die Kind-Tabelle ist von der Eltern-Tabelle abhängig. Für die Abbildung der Beziehung erscheint der Primärschlüssel der Eltern-Tabelle als Spalte (oder Gruppe von Spalten) in der Kind-Tabelle und bildet dort einen **Fremdschlüssel**. Die Angabe von Fremdschlüsseln allein genügt für die Abbildung einer tatsächlich brauchbaren Beziehung zwischen Tabellen jedoch nicht, diese entsteht erst durch die Einbindung von **referentiellen Integritätsregeln**.*

Implementierung von Datenintegrität beim Anlegen von Tabellen

- SQL Server stellt folgende Mechanismen zum Festlegen bzw. Erzwingen der Datenintegrität in einer Spalte bereit:

Einschränkungen

- ✓ *Constraints (engl.)*
- ✓ *Prüfvorschriften*
- ✓ *Prüfregeln*
- ✓ *Gültigkeitsregeln*

PRIMARY KEY

UNIQUE

NOT NULL

CHECK

DEFAULT

FOREIGN KEY

*Entitäten-
Integrität*

*Domänen-
Integrität*

**REFERENTIELLE
INTEGRITÄTS-
REGELN**

*Referentielle
Integrität*

PRIMARY KEY \leftrightarrow NOT NULL + UNIQUE

- Wurden die Bedingungen einmal deklariert, überprüft das DBMS bei jedem Einfügings- und Aktualisierungsversuch, ob die gesetzten Bedingungen durch den neu einzufügenden oder zu ändernden Datensatz missachtet werden und verhindert die auslösende Operation gegebenenfalls.

Implementierung von Datenintegrität beim Anlegen von Tabellen

► Viele der Prüfregeln für Spaltenwerte lassen sich auf zwei Arten im Quellcode von T-SQL implementieren:

Angabe implizit

```
CREATE TABLE      Position
(
    PNr            INT NOT NULL IDENTITY(100,1) PRIMARY KEY,
    RNr            INT NOT NULL FOREIGN KEY REFERENCES Rechnung (RNr) ON DELETE CASCADE,
    ANr            INT NOT NULL FOREIGN KEY REFERENCES Artikel (ANr),
    Anzahl         INT NULL CHECK (Anzahl BETWEEN 0 AND 1000),
    GPreis         SMALLMONEY NULL
    Währungsart    VARCHAR(20) NULL DEFAULT 'EURO')
```

Angabe explizit

[CONSTRAINT <Regelname>]

```
CREATE TABLE      Position
(
    PNr            INT NOT NULL IDENTITY(100,1),
    RNr            INT NOT NULL,
    ANr            INT NOT NULL,
    Anzahl         INT NULL,
    GPreis         SMALLMONEY NULL,
    Währungsart    VARCHAR(20) NULL DEFAULT 'EURO'
    CONSTRAINT PK_Position PRIMARY KEY (PNr),
    CONSTRAINT FK_Position_1 FOREIGN KEY (RNr) REFERENCES Rechnung (RNr) ON DELETE CASCADE,
    CONSTRAINT FK_Position_2 FOREIGN KEY (ANr) REFERENCES Artikel (ANr),
    CONSTRAINT Anzahl_Werte CHECK (Anzahl BETWEEN 0 AND 1000),
    CONSTRAINT Unique_Spalten UNIQUE(PNr))
```

Anlegen einer Tabelle

```
CREATE TABLE tabelle1 (merkmal1 typ1 [,merkmal2 typ2, ...]);
```

```
CREATE TABLE Kunde (
  KNr      INT UNIQUE NOT NULL IDENTITY(100,1),
  VName    VARCHAR(50) NULL,
  ZName    VARCHAR(50) NULL,
  Geschlecht CHAR(1) NOT NULL,
  Strasse  VARCHAR(50) NULL,
  PLZ      VARCHAR(20) NULL,
  Ort      VARCHAR(50) NULL,
  CONSTRAINT PK_Kunde PRIMARY KEY (KNr),
  CONSTRAINT CK_Geschlecht CHECK(GESCHLECHT IN('w','m')) )
```

NOT NULL

Ein solches Feld darf nicht leer sein, es muss auf jeden Fall einen Wert erhalten.

UNIQUE

Jeder Wert in dieser Spalte muss innerhalb der Spalte eindeutig sein.

PRIMARY KEY

Ist eine Spalte als Schlüsselattribut deklariert, so sind implizit beide Sondereigenschaften gesetzt: Ein Primärschlüssel darf nicht leer und er muss eindeutig sein! Denn der Wert einer Primärschlüssel-Zelle ist für diese Zeile eindeutig, damit sind mehrere, leere oder doppelte Werte ausgeschlossen.

Anlegen einer Tabelle

```
CREATE TABLE Kunde (
  KNr      INT UNIQUE NOT NULL IDENTITY(100,1),
  VName    VARCHAR(50) NULL,
  ZName    VARCHAR(50) NULL,
  Geschlecht CHAR(1) NOT NULL,
  Strasse  VARCHAR(50) NULL,
  PLZ      VARCHAR(20) NULL,
  Ort      VARCHAR(50) NULL,
  CONSTRAINT PK_Kunde PRIMARY KEY (KNr),
  CONSTRAINT CK_Geschlecht CHECK(GESCHLECHT IN('w','m')) )
```

IDENTITY [(seed , increment)]

▶ Erstellt eine Identitätsspalte in einer Tabelle. Diese Eigenschaft wird in den Transact-SQL-Anweisungen CREATE TABLE und ALTER TABLE verwendet. Argumente:

▶ **seed** Der Wert, der für die erste in die Tabelle geladene Zeile verwendet wird.

increment Der inkrementelle Wert, der zum Identitätswert der zuvor geladenen Zeile addiert wird. Angabe entweder sowohl vom Ausgangswert als auch vom inkrementellen Wert oder von keinem der beiden (Standardwert: (1, 1)).

CHECK (<Ausdruck> <Prädikat>)

▶ CHECK-Einschränkungen erzwingen die Domänenintegrität, indem sie die Werte begrenzen, die in einer Spalte zulässig sind.

Anlegen einer Tabelle

```
CREATE TABLE      Position
(
    PNr            INT UNIQUE NOT NULL IDENTITY(100,1),
    RNr            INT NOT NULL,
    ANr            INT NOT NULL,
    Anzahl         INT NULL,
    CONSTRAINT PK_Position PRIMARY KEY (PNr),
    CONSTRAINT FK_Position_2 FOREIGN KEY (ANr) REFERENCES Artikel (ANr),
    CONSTRAINT FK_Position_1 FOREIGN KEY (RNr) REFERENCES Rechnung (RNr)
    ON DELETE CASCADE,
)
```

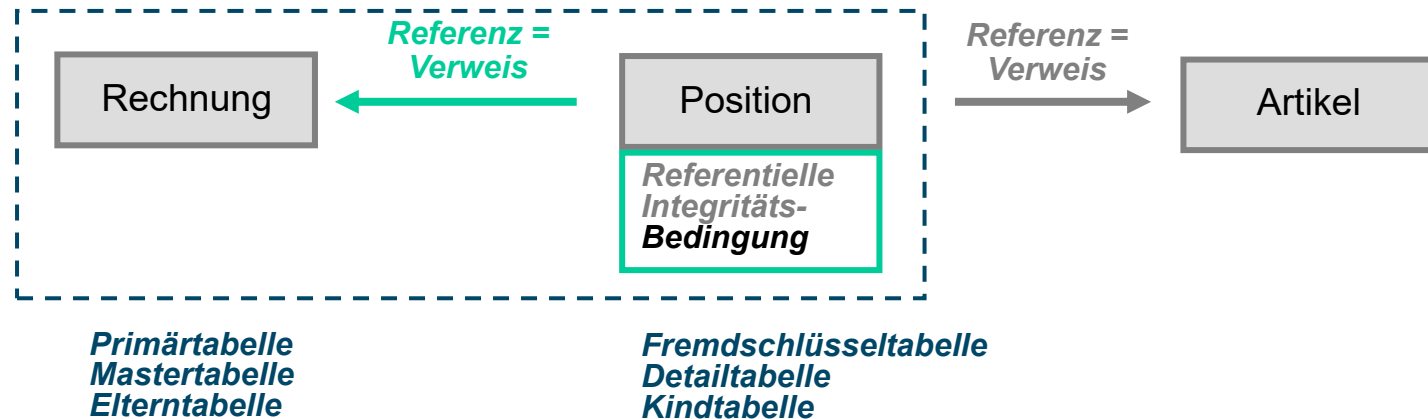
FOREIGN KEY

- Über die Definition von Fremdschlüsseln lassen sich die 1: n – Beziehungen eines normalisierten Relationenmodells in Tabellen einer Datenbank hinterlegen. Referentielle Integritätsbedingungen werden hierbei über entsprechende FOREIGN KEY Definitionen festgelegt.
- Durch die Angabe von referentiellen Integritätsbedingungen wird es für das DBMS möglich, bei Manipulationen / Operationen auf eine Tabelle zu prüfen, ob diese Operation zulässig ist bzw. ob sich daraus Folgeoperationen ergeben.
- Die referentielle Integrität macht eine Beziehung zwischen Tabellen erst zu dem, was sie ist!

Beim Löschen einer Rechnung ist es wünschenswert, auch sofort alle Rechnungspositionen zu löschen. Umgekehrt ist das Anlegen einer Position nur möglich, wenn es auch die entsprechende Rechnung gibt.

Anlegen einer Tabelle

```
CREATE TABLE      Position
(
    PNr            INT UNIQUE NOT NULL IDENTITY(100,1),
    RNr            INT NOT NULL,
    ANr            INT NOT NULL,
    Anzahl         INT NULL,
    CONSTRAINT PK_Position PRIMARY KEY (PNr),
    CONSTRAINT FK_Position_2 FOREIGN KEY (ANr) REFERENCES Artikel (ANr),
    CONSTRAINT FK_Position_1 FOREIGN KEY (RNr) REFERENCES Rechnung (RNr)
    ON DELETE CASCADE,
)
```



Referentielle Integritätsbedingungen beim SQL Server

Über die Angabe von ON DELETE bzw. ON UPDATE lassen sich unterschiedliche Reaktionen auf das Löschen bzw. Ändern von Datensätzen in der **Primärtabelle** definieren.

	ON DELETE	ON UPDATE
CASCADE	Ein DELETE in der Primärtabelle führt auch zu einem Löschen der entsprechenden Datensätze in der Fremdschlüsseltabelle.	Ein UPDATE in der Primärtabelle führt auch zu einer Änderung der Fremdschlüsselwerte in den entsprechenden Datensätzen in der Fremdschlüsseltabelle.
NO ACTION	Ein DELETE in der Primärtabelle kann nur ausgeführt werden, wenn in keiner Fremdschlüsseltabelle mehr ein Satz mit dem entsprechenden Wert existiert. Dies ist das Defaultverhalten .	Ein UPDATE in der Primärtabelle kann nur ausgeführt werden, wenn in keiner Fremdschlüsseltabelle mehr ein Satz mit dem entsprechenden Wert existiert. Dies ist das Defaultverhalten .
SET NULL	Wird ein Eintrag in der Primärtabelle (Mastertabelle) gelöscht, werden die entsprechenden Datensätze in der Fremdschlüsseltabelle (Detaildatensätze) zwar nicht gelöscht, aber der Inhalt der Fremdschlüsselspalte geleert. Dies ist allerdings nur möglich, wenn diese NULL-Werte zulässt.	Wird ein Eintrag in der Primärtabelle (Mastertabelle) geändert, werden die entsprechenden Datensätze in der Fremdschlüsseltabelle (Detaildatensätze) zwar nicht gelöscht, aber der Inhalt der Fremdschlüsselspalte geleert. Dies ist allerdings nur möglich, wenn diese NULL-Werte zulässt.
SET DEFAULT	Wird ein Eintrag in der Primärtabelle (Mastertabelle) gelöscht, werden die entsprechenden Datensätze in der Fremdschlüsseltabelle (Detaildatenätze) zwar nicht gelöscht, aber der Inhalt der Fremdschlüsselspalte auf den definierten Stanadardwert zurückgesetzt wird. Dies ist allerdings nur möglich, wenn es einen Standardwert für diese Spalte gibt und dieser referenziert werden kann.	Wird ein Eintrag in der Primärtabelle (Mastertabelle) geändert, werden die entsprechenden Datensätze in der Fremdschlüsseltabelle (Detaildatensätze) zwar nicht gelöscht, aber der Inhalt der Fremdschlüsselspalte auf den definierten Stanadardwert zurückgesetzt wird. Dies ist allerdings nur möglich, wenn es einen Standardwert für diese Spalte gibt und dieser referenziert werden kann.

Ändern von Struktur einer Tabelle (Schema-Modifikation)

ALTER TABLE tabelle1 **ADD | DROP** merkm1 typ1 [,merkm2 typ2, ...];

▶ Ändert die Definition / Struktur einer Tabelle durch Hinzufügen, Modifizieren oder Löschen von Spalten und / oder Einschränkungen.

```
CREATE TABLE      Artikel
(
    ANr             INT UNIQUE NOT NULL IDENTITY(100,1),
    AName           VARCHAR(50) NULL DEFAULT 'Neuer Artikel',
    CONSTRAINT PK_Artikel PRIMARY KEY (ANr)
)
```

DEFAULT

▶ Gibt den Wert an, der für die Spalte bereitgestellt wird, wenn bei einer INSERT-Aktion kein Wert explizit angegeben wird. **DEFAULT** – Definitionen können auf jede Spalte angewendet werden. Davon ausgeschlossen sind lediglich Spalten, die durch die **IDENTITY**-Eigenschaft definiert sind.

```
ALTER TABLE      Artikel
ADD              APreis      SMALLMONEY
```

```
ALTER TABLE      Artikel
ALTER COLUMN      APreis      MONEY
```

Ändern von Struktur einer Tabelle (Schema-Modifikation)

```
CREATE TABLE Rechnung
(
    RNr      INT UNIQUE NOT NULL IDENTITY(100,1),
    KNr      INT NOT NULL,
    Datum    SMALLDATETIME NULL,
    CONSTRAINT PK_Rechnung PRIMARY KEY (RNr)
)
```

```
ALTER TABLE Rechnung
ADD CONSTRAINT FK_Rechnung FOREIGN KEY (KNr) REFERENCES Kunde (KNr)
```

Anlegen einer Tabelle – Definition des SQL-Statements in T-SQL

CREATE TABLE table_name

```
(
    {
        < column_definition > | < table_constraint >
    } [ ,...n ]
)
```

< column_definition > ::=

```
{ column_name data_type }
[ { DEFAULT constant_expression | [ IDENTITY [ ( seed , increment ) ] ] } ]
[ ROWGUIDCOL ]
[ < column_constraint > [ ...n ] ]
```

< column_constraint > ::= [CONSTRAINT constraint_name]

```
{
    [ NULL | NOT NULL ] |
    [ PRIMARY KEY | UNIQUE ] | REFERENCES ref_table [ ( ref_column ) ]
    [ ON DELETE { CASCADE | NO ACTION } ] [ ON UPDATE { CASCADE | NO ACTION } ]
}
```

< table_constraint > ::= [CONSTRAINT constraint_name]

```
{
    [ { PRIMARY KEY | UNIQUE } { ( column [ ,...n ] ) } ] |
    FOREIGN KEY ( column [ ,...n ] ) REFERENCES ref_table [ ( ref_column [ ,...n ] ) ]
    [ ON DELETE { CASCADE | NO ACTION } ] [ ON UPDATE { CASCADE | NO ACTION } ]
}
```

SQL-Skript zum Rechnungsbeispiel (Anlegen von Tabellen)

```

CREATE TABLE      Kunde
(
    KNr      INT UNIQUE NOT NULL IDENTITY(100,1),
    VName    VARCHAR(50) NULL,
    ZName    VARCHAR(50) NULL,
    Strasse  VARCHAR(50) NULL,
    PLZ      VARCHAR(20) NULL,
    Ort      VARCHAR(50) NULL,
    CONSTRAINT PK_Kunde PRIMARY KEY (KNr))

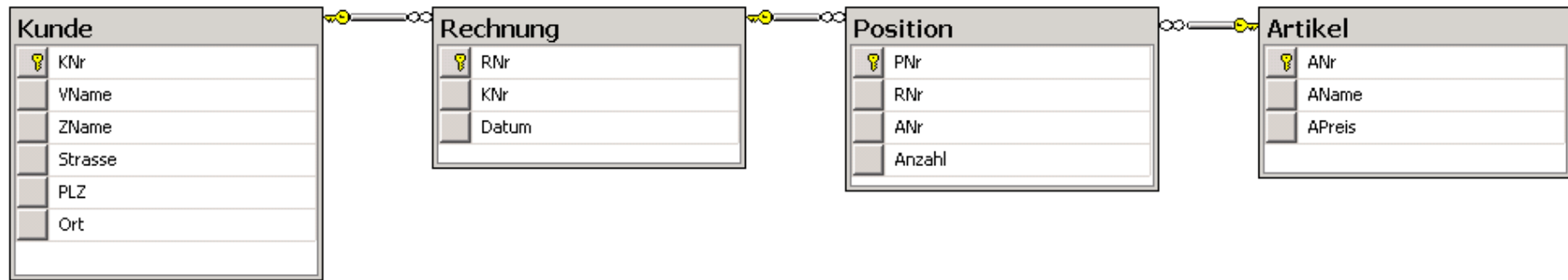
CREATE TABLE      Artikel
(
    ANr      INT UNIQUE NOT NULL IDENTITY(100,1),
    AName    VARCHAR(50) NULL DEFAULT 'Neuer Artikel',
    APreis   SMALLMONEY NULL,
    CONSTRAINT PK_Artikel PRIMARY KEY (ANr))

CREATE TABLE      Rechnung
(
    RNr      INT UNIQUE NOT NULL IDENTITY(100,1),
    KNr      INT NOT NULL,
    Datum    SMALLDATETIME NULL,
    CONSTRAINT PK_Rechnung PRIMARY KEY (RNr),
    CONSTRAINT FK_Kunde FOREIGN KEY (KNr) REFERENCES Kunde (KNr))

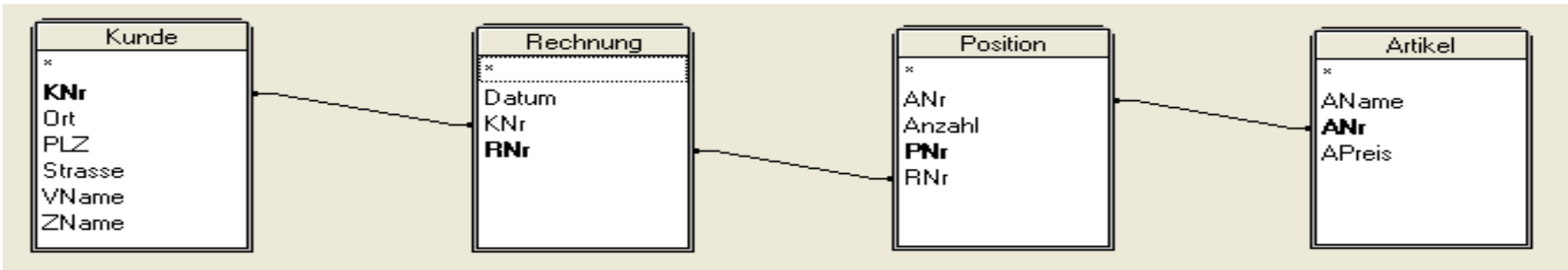
CREATE TABLE      Position
(
    PNr      INT UNIQUE NOT NULL IDENTITY(100,1),
    RNr      INT NOT NULL,
    ANr      INT NOT NULL,
    Anzahl   INT NULL,
    CONSTRAINT PK_Position PRIMARY KEY (PNr),
    CONSTRAINT FK_Position_1 FOREIGN KEY (RNr) REFERENCES Rechnung (RNr)
    ON DELETE CASCADE,
    CONSTRAINT FK_Position_2 FOREIGN KEY (ANr) REFERENCES Artikel (ANr))
    
```

Implementiertes Datenbankdiagramm zum Rechnungsbeispiel

SQL Server Ansicht



Excel Ansicht



Vom System generierter Code zum Anlegen einer Tabelle

```
USE [DBV]
GO
/***** Objekt: Table [dbo].[Position] Skriptdatum: 06/10/2009 07:45:45 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Position]
(
    [PNr] [int] IDENTITY(100,1) NOT NULL,
    [RNr] [int] NOT NULL,
    [ANr] [int] NOT NULL,
    [Anzahl] [int] NULL,
    CONSTRAINT [PK_Position] PRIMARY KEY CLUSTERED ([PNr] ASC)WITH (IGNORE_DUP_KEY = OFF)
ON [PRIMARY]
) ON [PRIMARY]
GO
USE [DBV]
GO
ALTER TABLE [dbo].[Position] WITH CHECK ADD CONSTRAINT [FK_Position_1] FOREIGN KEY([RNr])
REFERENCES [dbo].[Rechnung] ([RNr]) ON DELETE CASCADE
GO
ALTER TABLE [dbo].[Position] WITH CHECK ADD CONSTRAINT [FK_Position_2] FOREIGN KEY([ANr])
REFERENCES [dbo].[Artikel] ([ANr])
```


Komplexe Integritätsbedingungen

Theorie!

Gemäß dem SQL-Standard sind auch komplexe Integritätsbedingungen, die sich auf mehrere Relationen beziehen, möglich. In gewisser Weise stellen natürlich die FOREIGN KEY – Klauseln schon solche Integritätsbedingungen dar, da sie sich immer auf zwei Relationen beziehen.

```
CREATE TABLE      prüfen
(
  MatrNr  INT NOT NULL FOREIGN KEY REFERENCES Studenten ON DELETE CASCADE,
  VorlNr  INT NOT NULL FOREIGN KEY REFERENCES Vorlesungen,
  PersNr  INT NOT NULL FOREIGN KEY REFERENCES Professoren ON DELETE SET NULL,
  Note    NUMERIC(2,1) CHECK(Note BETWEEN 0.7 AND 5.0),
  PRIMARY KEY (MatrNr, VorlNr),
  CONSTRAINT VorherHören
      CHECK (EXISTS (SELECT *
                      FROM hören h
                      WHERE h.VorlNr = prüfen.VorlNr AND
                           h.MatrNr = prüfen.MatrNr))
)
```

Leider werden diese ausdruckskräftigen Integritätsklauseln von den kommerziellen Datenbanksystemen derzeit kaum unterstützt. Das mag auch daran liegen, dass die Überprüfung u. a. sehr aufwendig ist. Deshalb muss man sich mit **Triggern** „behelfen“.

MS SQL Server liefert folgende Fehlermeldung:

Unterabfragen sind in diesem Kontext nicht zulässig. Es sind nur Skalarausdrücke zulässig.



Erweiterung der Datenbanksprache um Programmierkonstrukte - Kontrollstrukturen in T-SQL

- ▶ In jeder höheren Programmiersprache existieren Kontrollstrukturen, um komplexe Programmabläufe abzubilden, bei denen nicht sequentiell ein Befehl nach dem anderen abgearbeitet werden soll, sondern der Programmablauf durch Bedingungen und Wiederholungen gekennzeichnet wird.
- ▶ Der Name „Kontrollstrukturen“ kommt daher, dass man durch ihren Einsatz zusätzliche Kontrollmöglichkeiten über den Programmablauf erhält. Kontrollstrukturen werden in zwei Kategorien unterteilt:



Auswahl- und Entscheidungsstrukturen

IF-ELSE

CASE

Auswahlstrukturen kommen immer dann zum Einsatz, wenn ein Teil des Programmcodes nur unter bestimmten Voraussetzungen abgearbeitet wird, oder in Abhängigkeit von bestimmten Bedingungen einer Variablen unterschiedliche Werte zugewiesen werden sollen.

Kurz: **Immer dann, wenn irgendwelche Bedingungen geprüft werden müssen.**



Schleifen- oder Wiederholungsstrukturen

WHILE

Wiederholungsstrukturen dienen dazu, eine oder mehrere **Anweisungen mehrmals hintereinander auszuführen**. Die Anzahl der Wiederholungen kann dabei auf unterschiedliche Weise festgelegt werden. Wiederholungen gibt es

- mit einer fixen Anzahl Wiederholungen, die bereits zu Beginn der Schleife feststehen,
- Wiederholungen, die so lange durchgeführt werden, bis eine Abbruchbedingung eintritt. Das Eintreten der Abbruchbedingung kann sowohl am Anfang als auch am Ende der Schleife überprüft werden.

IF-ELSE – Anweisung

Eine Anweisung
pro Bedingungsblock

```
IF Bedingung
    Anweisung
```

```
IF Bedingung
    Anweisung
ELSE
    Anweisung
```

Mehrere Anweisungen
pro Bedingungsblock

```
IF Bedingung
BEGINN
    Anweisung1
    ...
    AnweisungN
END
```

```
IF Bedingung
    Anweisung
ELSE
BEGINN
    Anweisung1
    ...
    AnweisungN
END
```

```
IF Bedingung
BEGINN
    Anweisung1
    ...
    AnweisungN
END
ELSE
BEGINN
    Anweisung1
    ...
    AnweisungN
END
```

Prüfung mehrerer Bedingungen

```
IF Bedingung1
    Anweisung
ELSE
    IF Bedingung2
        Anweisungen
    ELSE
        IF Bedingung3
            Anweisungen
        ELSE
            Anweisungen
```

```
IF Bedingung1 AND Bedingung2 OR Bedingung3
BEGIN
    Anweisungen
END
```

```
IF MONTH(SYSDATETIME( )) IN(1,2,3,4,5)
    PRINT 'Vorsaison'
ELSE
    IF MONTH(SYSDATETIME( )) IN(6,7,8,9)
        PRINT 'Hauptsaison'
        PRINT 'Hier ist alles sehr teuer!'
    ELSE
        PRINT 'Nachsaison'
```

CASE – Anweisung

Ein Ausdruck wird mit verschiedenen
Ergebniswerten verglichen

CASE Bedingung

WHEN Wert1 **THEN** Ergebnis1

WHEN Wert2 **THEN** Ergebnis2

...

WHEN WertN **THEN** ErgebnisN

ELSE ErgebnisX **END**

Prüfung unterschiedlicher Bedingungen

CASE

WHEN Bedingung1 **THEN** Ergebnis1

WHEN Bedingung2 **THEN** Ergebnis2

...

WHEN BedingungN **THEN** ErgebnisN

ELSE ErgebnisX **END**

```
SELECT DISTINCT CASE DATEPART(MONTH, Datum)
```

```
  WHEN '1' THEN 'Januar'
```

```
  WHEN '2' THEN 'Februar'
```

```
  WHEN '3' THEN 'März'
```

```
  WHEN '4' THEN 'April'
```

```
  WHEN '5' THEN 'Mai,
```

```
  WHEN '6' THEN 'Juni'
```

```
  WHEN '7' THEN 'Juli'
```

```
  WHEN '8' THEN 'August'
```

```
  WHEN '9' THEN 'September'
```

```
  WHEN '10' THEN 'Oktober'
```

```
  WHEN '11' THEN 'November'
```

```
  WHEN '12' THEN 'Dezember'
```

```
  ELSE 'Monat unbekannt' END
```

```
FROM Rechnung
```

```
SELECT MatrNr,
```

```
( CASE
```

```
  WHEN Note < 1.5 THEN 'sehr gut'
```

```
  WHEN Note < 2.5 THEN 'gut'
```

```
  WHEN Note < 3.5 THEN 'befriedigend'
```

```
  WHEN Note < 4.0 THEN 'ausreichend'
```

```
  ELSE 'nicht bestanden' END )
```

```
FROM prüfen;
```

WHILE - Schleifen

WHILE Bedingung
BEGIN

 Anweisungen
END

Im folgenden Beispiel verdoppelt die WHILE-Schleife die Preise, solange der durchschnittliche Preis aller Artikel unter 7.000 liegt. Anschließend erfolgt die Ausgabe des Höchstpreises. Die Schleife wird durchlaufen solange der Höchstpreis unter 20.000 liegt.

*Tabelle Artikel
davor*

ANr	AName	APreis
100	Computer	5000,0000
101	Drucker	1000,0000
102	Kabel	500,0000
103	Neuer Artikel	NULL

```

WHILE (SELECT AVG(APreis) FROM Artikel) < 7000
BEGIN
    UPDATE Artikel
    SET APreis = APreis * 2

    SELECT MAX(APreis) AS Höchstpreis FROM Artikel
    IF (SELECT MAX(APreis) FROM Artikel) >= 20000
        BREAK
    ELSE
        CONTINUE
END
    
```

*Tabelle Artikel
danach*

ANr	AName	APreis
100	Computer	20000,0000
101	Drucker	4000,0000
102	Kabel	2000,0000
103	Neuer Artikel	NULL

Höchstpreis
10000,00

Höchstpreis
20000,00

Trigger als Ergänzung von Constraints

- ▶ Trigger sind benutzerdefinierte in Transact-SQL geschriebene Programme, die mit Ereignisprozeduren in anderen Programmiersprachen vergleichbar sind. Ereignisprozeduren werden **automatisch** gestartet, wenn das zugrunde liegende **Ereignis** eintritt. Ein Trigger ist fest mit einer bestimmten Tabelle verknüpft und reagiert auf **Datenmodifikation** (Ereignisse: INSERT, UPDATE, DELETE) dieser Tabelle. Man könnte Trigger demnach als Ereignisprozeduren für Tabellen bezeichnen.
- ▶ Der SQL-Server enthält zwei Arten von Triggern: DML-Trigger und DDL-Trigger. DML-Trigger sind aus folgenden Gründen nützlich:
 - Sie können als **Konsistenzsicherungsmechanismen** zur Ergänzung von Gültigkeitsprüfungen (Constraints) eingesetzt werden. Im Gegensatz zu CHECK-Einschränkungen können DML-Trigger auf Daten in anderen Tabellen zugreifen. Mit Trigger lassen sich demnach komplexere Geschäftsregeln als mit Constraints realisieren. Trigger werden erst ausgeführt, wenn alle anderen Integritätsprüfungen (Constraints) abgeschlossen sind.
 - Sie können **Änderungen** über verknüpfte Tabellen in der Datenbank kaskadierend **weitergeben**. So kann ein Trigger nicht nur Überprüfungs-, sondern auch Berechnungsfunktionen übernehmen. Denkbar sind z. B. Trigger, die Statistiken aktuell halten oder die Werte abgeleiteter Spalten berechnen.
 - Sie können den **Status** einer Tabelle vor und nach einer Datenänderung **auswerten** und, basierend auf den festgestellten Unterschieden, bestimmte **Aktionen ausführen**.
 - Mehrere DML-Trigger desselben Typs (INSERT, UPDATE oder DELETE) für eine Tabelle ermöglichen es, dass als Reaktion auf dieselbe Änderungsanweisung unterschiedliche Aktionen durchgeführt werden.

Trigger als Ergänzung von Constraints

```
CREATE TRIGGER
{ON | AFTER | INSTEAD OF}
[WITH ENCRYPTION]
[NOT FOR REPLICATION]
{INSERT, UPDATE, DELETE}
AS
BEGINN
```

```
END
```

triggername
tabellenname

SQL-Anweisungen

In der Artikeltable soll es die Spalte ArtMengeBestellt geben, die automatisch über Trigger gewartet wird. Wenn ein Artikel bestellt wird, muss die Menge automatisch erhöht werden; wenn eine Lieferung an den Kunden geht, muss sie automatisch reduziert werden.

UPDATE - Trigger

```
CREATE TRIGGER tblPosition_UPDATE
ON Position
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON
    IF UPDATE(ANr) OR UPDATE(Anzahl)
    BEGIN
        UPDATE Artikel
        SET Artikel.ArtMengeBestellt = Artikel.ArtMengeBestellt - Deleted.Anzahl
        FROM Deleted INNER JOIN Artikel ON Deleted.ANr = Artikel.ANr
        UPDATE Artikel
        SET Artikel.ArtMengeBestellt = Artikel.ArtMengeBestellt + Inserted.Anzahl
        FROM Inserted INNER JOIN Artikel ON Inserted.ANr = Artikel.ANr
    END
END
```

INSERT - Trigger

```
CREATE TRIGGER tblPosition_INSERT
ON Position
AFTER INSERT
AS
BEGIN
    SET NOCOUNT ON
    DECLARE @artikel INT
    DECLARE @menge INT
    SET @artikel = (SELECT ANr FROM inserted)
    SET @menge = (SELECT Anzahl FROM inserted)
    UPDATE Artikel
    SET ArtMengeBestellt = ArtMengeBestellt + @menge
    WHERE ANr = @artikel
END
```

DELETE - Trigger

```
CREATE TRIGGER tblPosition_DELETE
ON Position
AFTER DELETE
AS
BEGIN
    SET NOCOUNT ON
    DECLARE @artikel INT
    DECLARE @menge INT
    SET @artikel = (SELECT ANr FROM deleted)
    SET @menge = (SELECT Anzahl FROM deleted)
    UPDATE Artikel
    SET ArtMengeBestellt = ArtMengeBestellt - @menge
    WHERE ANr = @artikel
END
```

Überprüfung der Datenbankintegrität mit **DBCC CHECKDB**

- ▶ Die Anweisung **DBCC CHECKDB** (T-SQL) überprüft die Zuordnungsintegrität sowie die strukturelle und logische Integrität aller Objekte in der angegebenen Datenbank. Auf diese Weise wird die Zuordnung und strukturelle Integrität von Benutzer- und Systemtabellen sowie Indizes in der Datenbank überprüft.
- ▶ Durch die Ausführung von **DBCC** wird sichergestellt, dass etwaige Integritätsprobleme in der Datenbank gemeldet werden und später vom Systemadministrator oder Datenbankbesitzer behoben werden können.
- ▶ Die Ausführung von **DBCC CHECKDB** schließt Folgendes ein:
 - ☑ Die Ausführung von **DBCC CHECKALLOC** für die Datenbank
 - ☑ Die Ausführung von **DBCC CHECKTABLE** für jede Tabelle und Sicht in der Datenbank
 - ☑ Die Überprüfung der Service Broker-Daten in der Datenbank
 - ☑ Die Ausführung von **DBCC CHECKCATALOG** für die Datenbank
 - ☑ Die Überprüfung des Inhalts jeder indizierten Sicht in der Datenbank
- ▶ **DBCC CHECKDB** verwendet einen internen Datenbanksnapshot, um die Transaktionskonsistenz bereitzustellen, die notwendig ist, um diese Überprüfungen auszuführen.

Überprüfung der Datenbankintegrität mit **DBCC CHECKDB**

```
DBCC CHECKDB [ ( 'database_name' | database_id | 0
                [, NOINDEX |
                    { REPAIR_ALLOW_DATA_LOSS |
                      REPAIR_FAST |
                      REPAIR_REBUILD } ] ) ]
                [ WITH {
                    [ ALL_ERRORMSG ]
                    [, [ NO_INFOMSGS ] ]
                    [, [ TABLOCK ] ]
                    [, [ ESTIMATEONLY ] ]
                    [, [ PHYSICAL_ONLY ] ]
                    [, [ DATA_PURITY ] ] } ] ]
```

1.

2.

3.

1. Gibt an, dass keine intensive Überprüfung nicht gruppierter Indizes für Benutzertabellen ausgeführt werden soll. Auf diese Weise wird die Gesamtausführungszeit reduziert. NOINDEX wirkt sich nicht auf Systemtabellen aus, da Integritätsprüfungen für Systemtabellenindizes immer ausgeführt werden.
2. Gibt an, dass DBCC CHECKDB gefundene Fehler behebt. Die angegebene Datenbank muss sich im Einzelbenutzermodus befinden, damit eine der folgenden Reparaturoptionen verwendet werden kann.
3. Ermöglicht die Angabe von Optionen.

-- Check the current database.

DBCC CHECKDB;

-- Check the DBV database without nonclustered indexes.

DBCC CHECKDB ('DBV', NOINDEX);

-- Überprüfen der aktuellen Datenbank, wobei Informationsmeldungen unterdrückt werden

DBCC CHECKDB WITH NO_INFOMSGS;

Anzeigen einer Tabelle

SP_HELP tabelle

SP_HELP

SP_HELP Artikel

- ▶ Anzeige der Tabelle mit Datenfelddefinition
(Ohne Angabe von Tabellennamen: Anzeige der vorhandenen Tabellen in der ausgewählten DB)

Umbenennen einer Tabelle

SP_RENAME 'alter_tabellenname', 'neuer_tabellenname'

EXEC SP_RENAME 'Artikel', 'Material'

EXEC SP_RENAME 'Material.AName', 'MBezeichnung'

- ▶ Ändert den Namen eines vom Benutzer erstellten Objekts in der aktuellen Datenbank.
Bei diesem Objekt kann es sich um eine Tabelle, einen Index, eine Spalte oder einen Aliasdatentyp handeln.

Löschen (von vorhandenen Datensätzen in) einer Tabelle

```
DELETE FROM tabelle1
[WHERE suchbedingung1 [AND/OR suchbedingung2...]];
```

► Gelöscht wird der **Inhalt einer Tabelle**, nicht die Tabelle selbst.

Die Angabe von Bedingung(en) in der WHERE-Klausel ist optional.

Wird keine Bedingung angegeben, so wird der gesamte Tabelleninhalt gelöscht.

```
DROP TABLE tabelle1;
```

► Die angegebene **Tabelle** wird mit allen enthaltenen Daten inkl. ihrer Definition aus dem System **entfernt!**

Wenn andere Datenobjekte oder Transaktionen die zu löschende Tabelle verwenden oder referenzieren, ist das Löschen nicht möglich!

DELETE FROM	Artikel
WHERE	APreis > 500
DELETE FROM	Artikel
DROP TABLE	Artikel

Aufnahme von (neuen) Datensätzen in eine Tabelle

```
INSERT INTO tabelle1 (merkmal1, [,merkmal2,...])
VALUES (wert1 [,wert2,...]);
```

```
CREATE TABLE Artikel
(
    ANr      INT NOT NULL IDENTITY(100,1),
    AName    VARCHAR(50) NULL DEFAULT 'Neuer Artikel',
    APreis   SMALLMONEY NULL,
    CONSTRAINT PK_Artikel PRIMARY KEY (ANr)
)
```

```
INSERT INTO Artikel
VALUES ('Computer', 5000)

INSERT INTO Artikel (APreis, AName)
VALUES (1000, 'Drucker')

INSERT INTO Artikel (APreis, AName)
VALUES (500, 'Kabel')

INSERT INTO Artikel (APreis, AName)
VALUES (NULL, DEFAULT)
```

ANr	AName	APreis
100	Computer	5000
101	Drucker	1000
102	Kabel	500
103	Neuer Artikel	NULL

Einfügen der Daten, die nicht in der Reihenfolge der Spalten vorliegen.

Ändern von Werten in Datensätzen einer Tabelle

```
UPDATE tabelle1 SET merkmal1 = wert1 [,merkmal2 = wert2, ...]
[WHERE suchbedingung1 [AND/OR suchbedingung2...]];
```

```
UPDATE Artikel
SET APreis = 100
WHERE AName = 'Kabel'
```



ANr	AName	APreis
100	Computer	5000
101	Drucker	1000
102	Kabel	100

```
UPDATE Artikel
SET APreis = ROUND(APreis * 0.95, 2)
WHERE AName = 'DRUCKER'
```



ANr	AName	APreis
100	Computer	5000
101	Drucker	950
102	Kabel	100

Einbindung von **Funktionen** in den Quellcode:

Der Preis der Artikel mit der Bezeichnung ‚Drucker‘ wird um 5 % gesenkt, dabei wird der neue Preis auf zwei Nachkommastellen gerundet.

Kleine Auswahl nützlicher T-SQL - Funktionen

Datumsfunktionen

Syntax	Beschreibung	Beispiele
DAY(<datetime-Ausdruck>)	Ermittelt den Tag eines Datumswerts, Rückgabewert Ganzzahl	DAY ('2008-10-25') erzeugt 25
MONTH (<datetime-Ausdruck>)	Ermittelt den Monat eines Datumswerts, Rückgabewert Ganzzahl	MONTH ('2008-10-25') erzeugt 10
YEAR (<datetime-Ausdruck>)	Ermittelt das Jahr eines Datumswerts, Rückgabewert Ganzzahl	YEAR ('2008-10-25') erzeugt 2008
GetDate()	Liefert das aktuelle Systemdatum und Uhrzeit, Rückgabewert datetime	CREATE TABLE... Datum SMALLDATETIME NOT NULL DEFAULT (GetDate())....
DateDiff(<Datumseinheit>,<Start-Datum>,<End-Datum>)	Berechnet die Differenz zwischen Start- und Enddatum	DateDiff (day, '2000-1-1', '2001-1-1') erzeugt 366

Lesen von Datensätzen aus Tabelle(n)

Bei der Speicherung und der Selektion von Daten in Tabellen...

- ▶ Die Reihenfolge der Zeilen spielt keine Rolle.
Zeilen werden über ihre Schlüsselwerte identifiziert.
- ▶ Die Reihenfolge der Spalten spielt keine Rolle.
Spalten werden über ihre Spaltenüberschriften identifiziert.

Beziehungen zwischen den Daten / Datensätzen hängen von den einzelnen Dateninhalten (Werten) ab, dabei spielen die Spaltenüberschriften eine wichtige Rolle. Entsprechend den obigen Prinzipien / Regeln lautet der SELECT – Befehlssatz (zunächst stark vereinfacht) wie folgt:

SELECT [DISTINCT | ALL] Attribut(e) {oder} *

Ausgabe

obligatorisch

FROM Relation(en)

Eingabe

obligatorisch

WHERE Bedingung(en)

optional

ORDER BY Attribut(e) [ASC | DESC]

optional

Lesen von Datensätzen aus Tabelle(n)

Schlüsselwort	Erläuterung
SELECT: * : ALL: DISTINCT:	Auswahl bestimmter Attribute Welche Spalten / Daten sollen pro Datensatz ausgegeben werden? Auswahl aller Attribute Auswahl mit Duplikaten (default) Auswahl ohne Duplikate Die Angabe DISTINCT bezieht sich immer auf alle Tupel, nicht nur auf einzelne Attribute.
FROM:	Legt fest, aus welchen Relationen ausgewählt wird
WHERE:	Auswahlbedingungen für bestimmte Datensätze Attributwerte der einzelnen Datensätze sollen bestimmte Bedingungen erfüllen, damit der jeweiliger Datensatz in die Ergebnisrelation aufgenommen werden kann.
ORDER BY: ASC: DESC:	Festlegung der Attribute, nach welchen das Ergebnis sortiert werden soll Aufsteigende Sortierung (default) Absteigende Sortierung

Mengenorientierte Operationen in SELECT – Statements

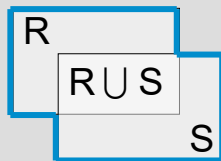
- ▶ Um die Mengenoperationen Union, Differenz und Durchschnitt auf den Relationen R und S durchführen zu können, müssen beide Relationen miteinander **kompatibel** sein.
- ▶ Die Typkompatibilität (Vereinigungsverträglichkeit o. Schemagleichheit) zweier Relationen ist gegeben, wenn
 - ✓ R und S den gleichen Grad (Attribut- bzw. Spaltenanzahl) haben
 - ✓ der Wertebereich der Attribute von R und S identisch ist
- ▶ Ist die Typkompatibilität nicht gegeben, so kann diese künstlich erzeugt werden, indem fehlende Spalten der entsprechenden Relation mit NULL aufgefüllt werden.

Schemagleichheit



Union

zweier Tabellen R und S



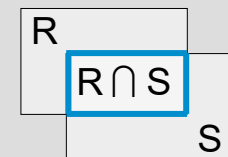
Differenz

zweier Tabellen R und S



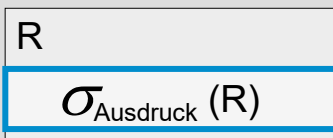
Durchschnitt

zweier Tabellen R und S



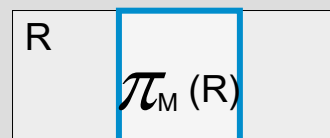
Selektion

aus der Tabelle R mit Hilfe eines Ausdrucks



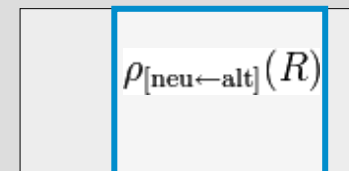
Projektion

der Tabelle R auf eine Menge M von Merkmalen

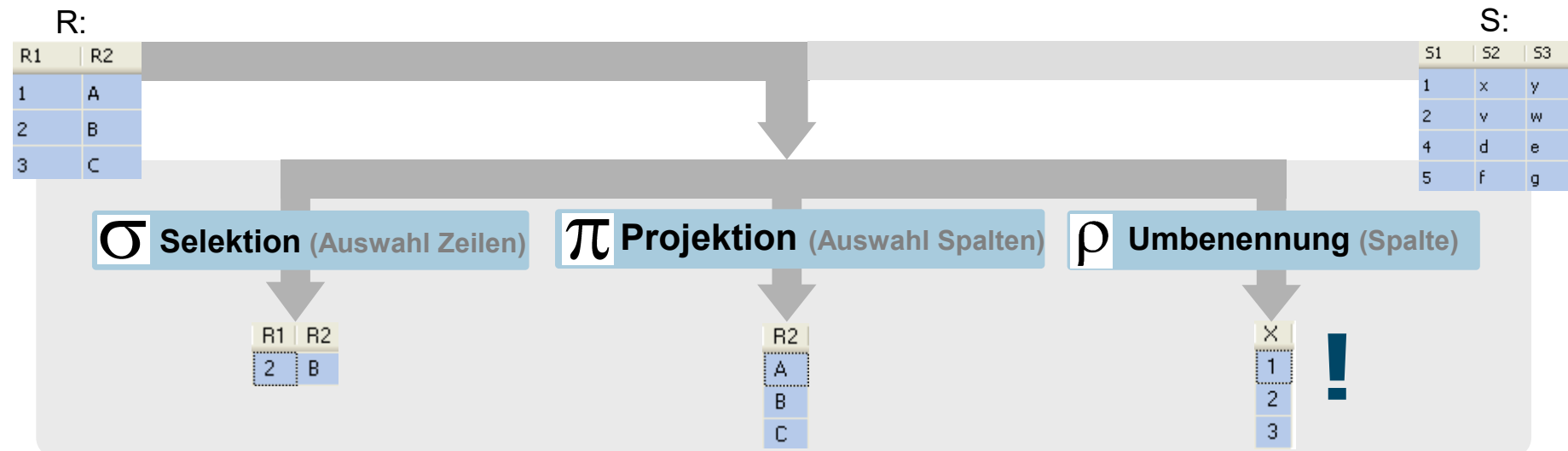
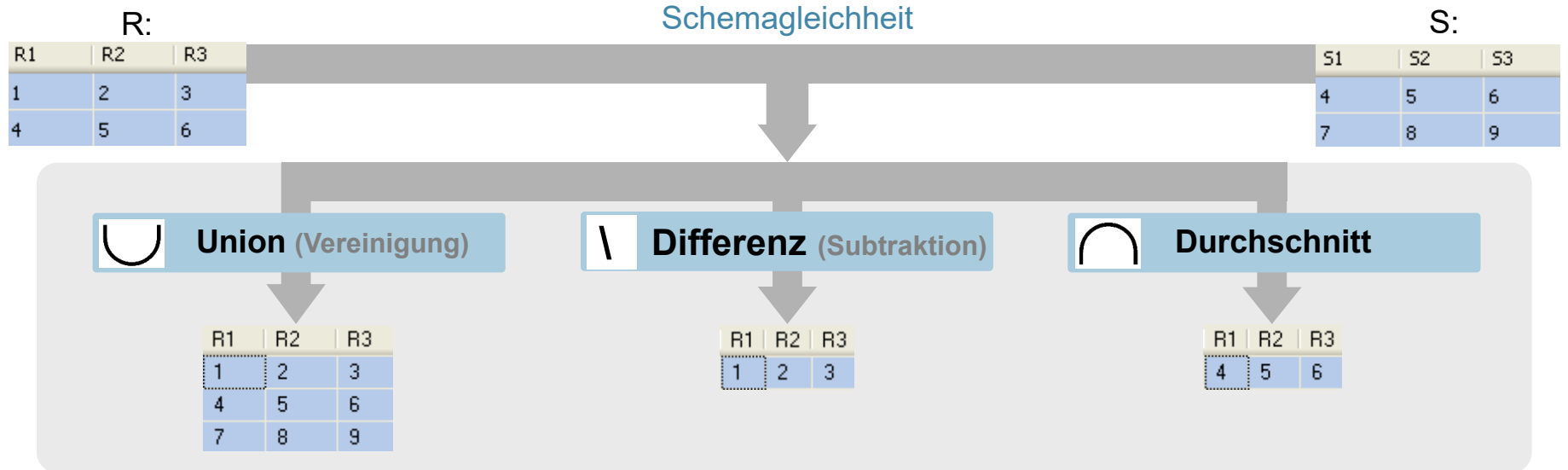


Umbenennung

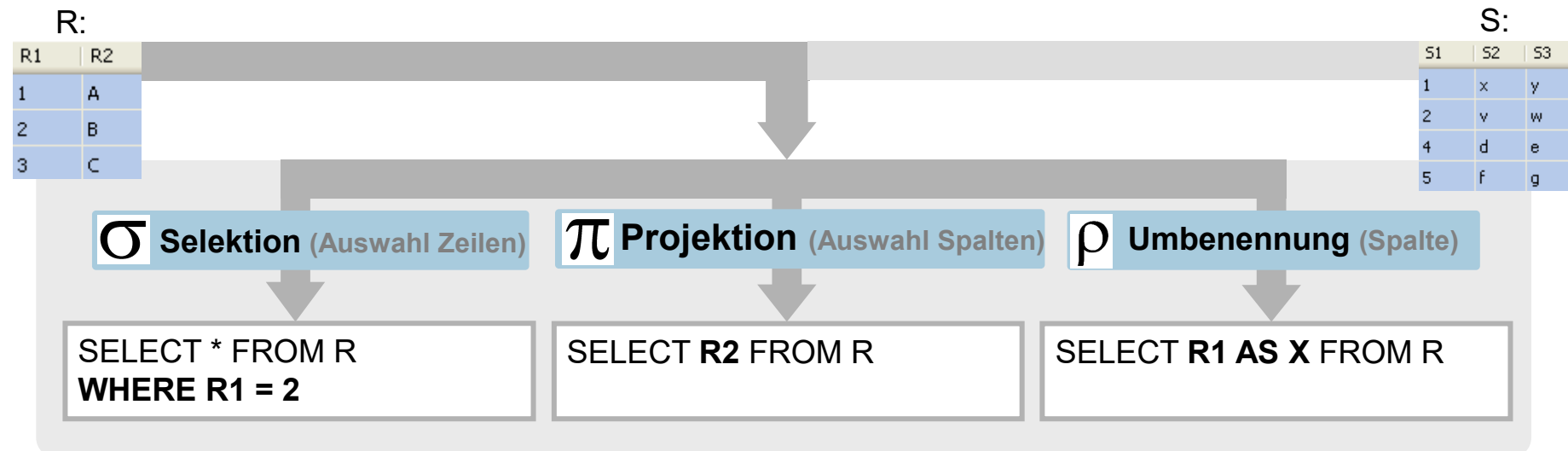
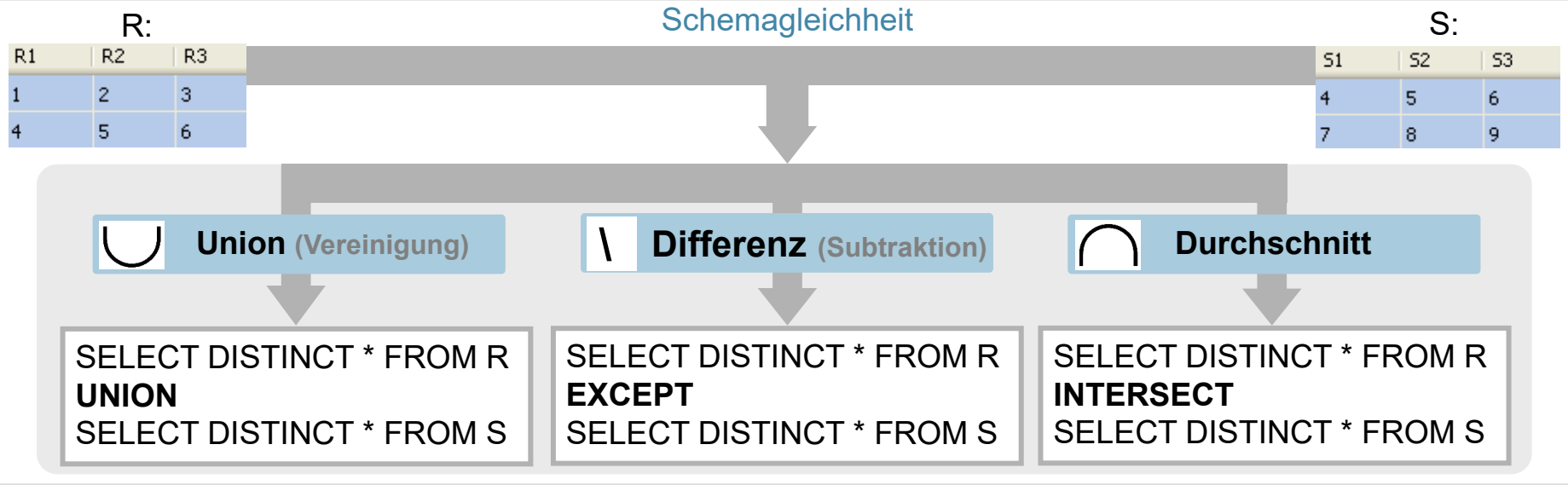
einer Tabelle oder einer Spalte



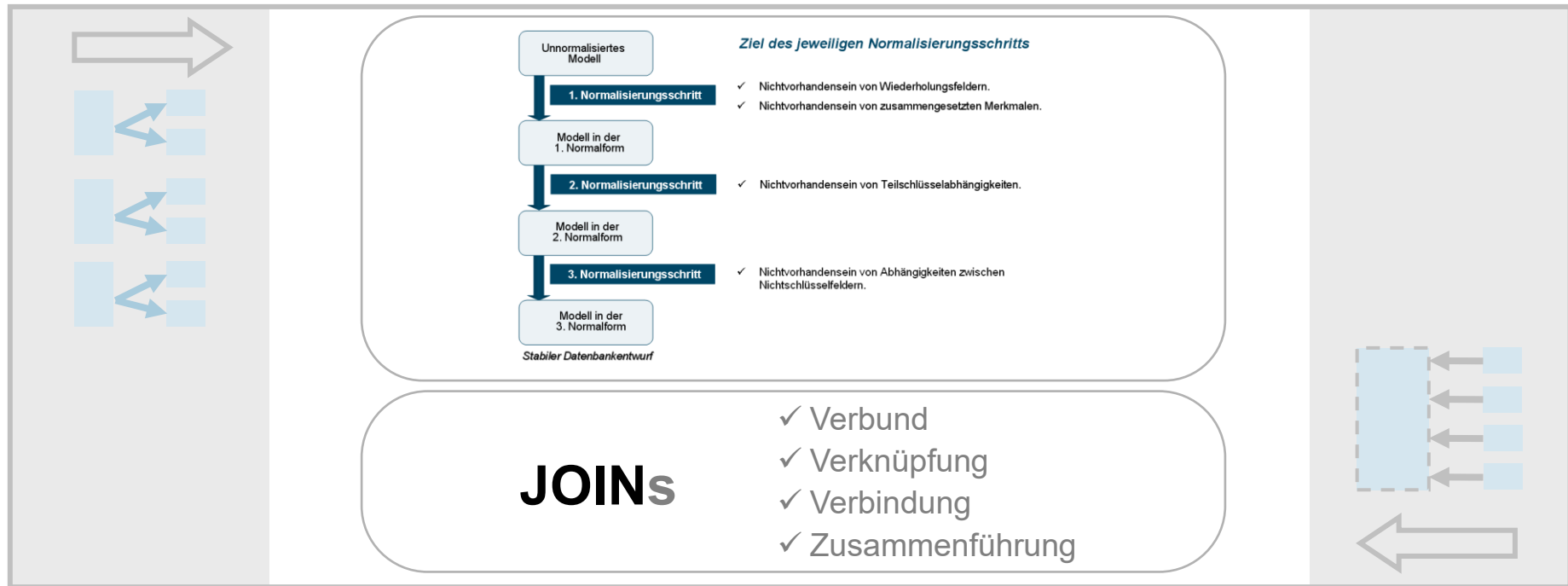
Mengenorientierte Operationen in SELECT – Statements



Mengenorientierte Operationen in SELECT – Statements



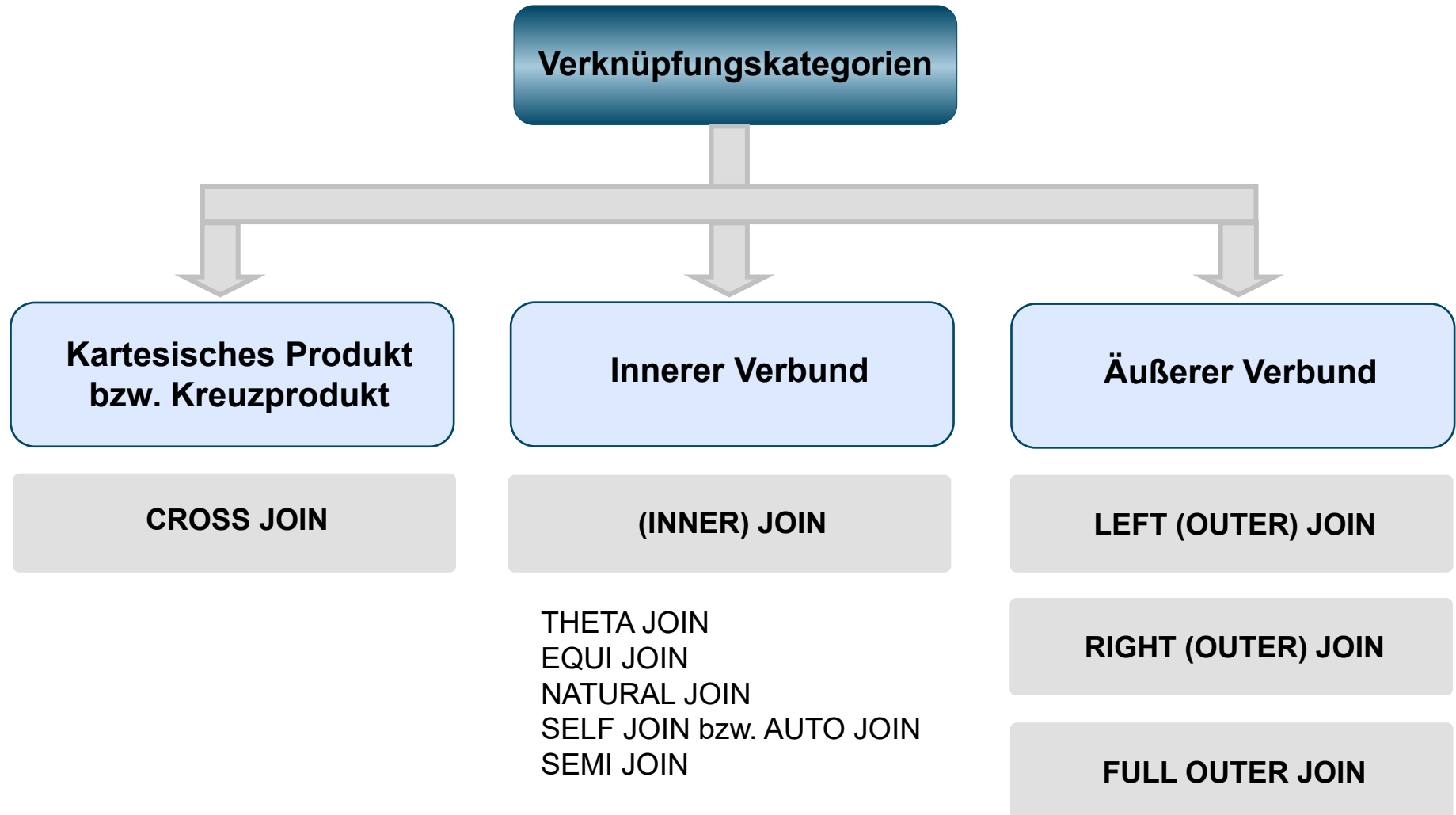
Lesen aus mehreren Tabellen - JOINS



Prinzip:

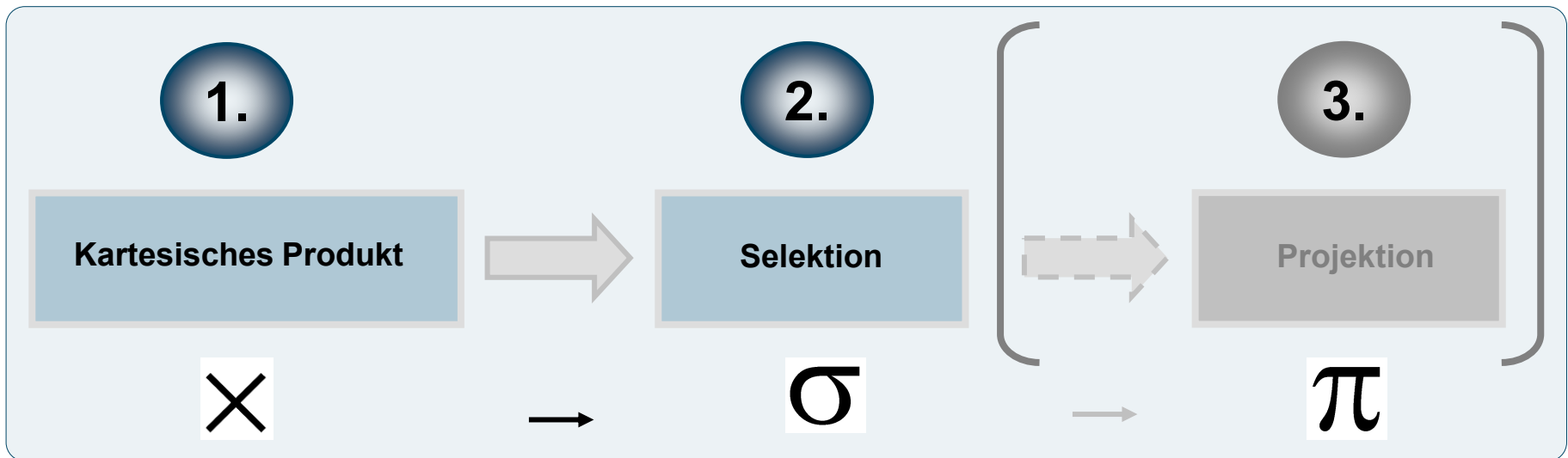
*Aufspaltung einer größeren Tabelle zur Redundanzvermeidung in mehrere kleinere Tabellen und deren Rekonstruktion durch **Joins**!*

Lesen aus mehreren Tabellen - JOINS

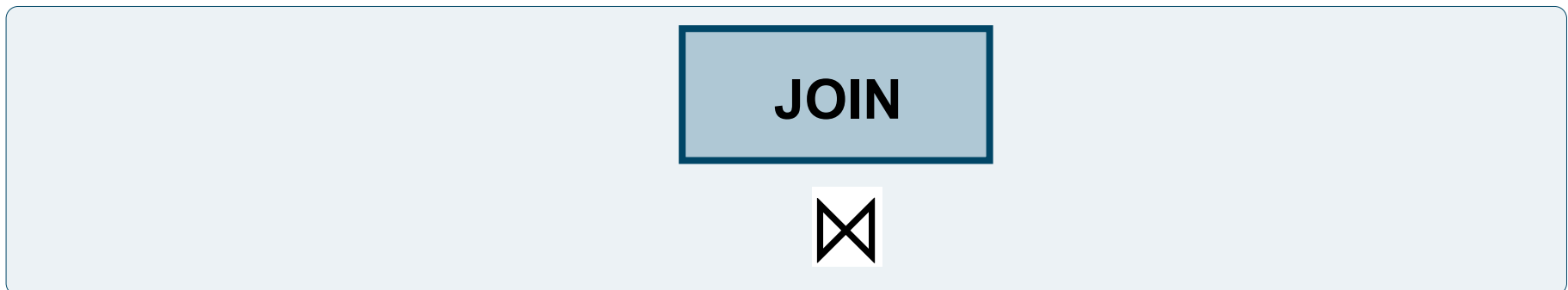


Das grundlegende Prinzip eines JOINS

Kombination der Operationen:

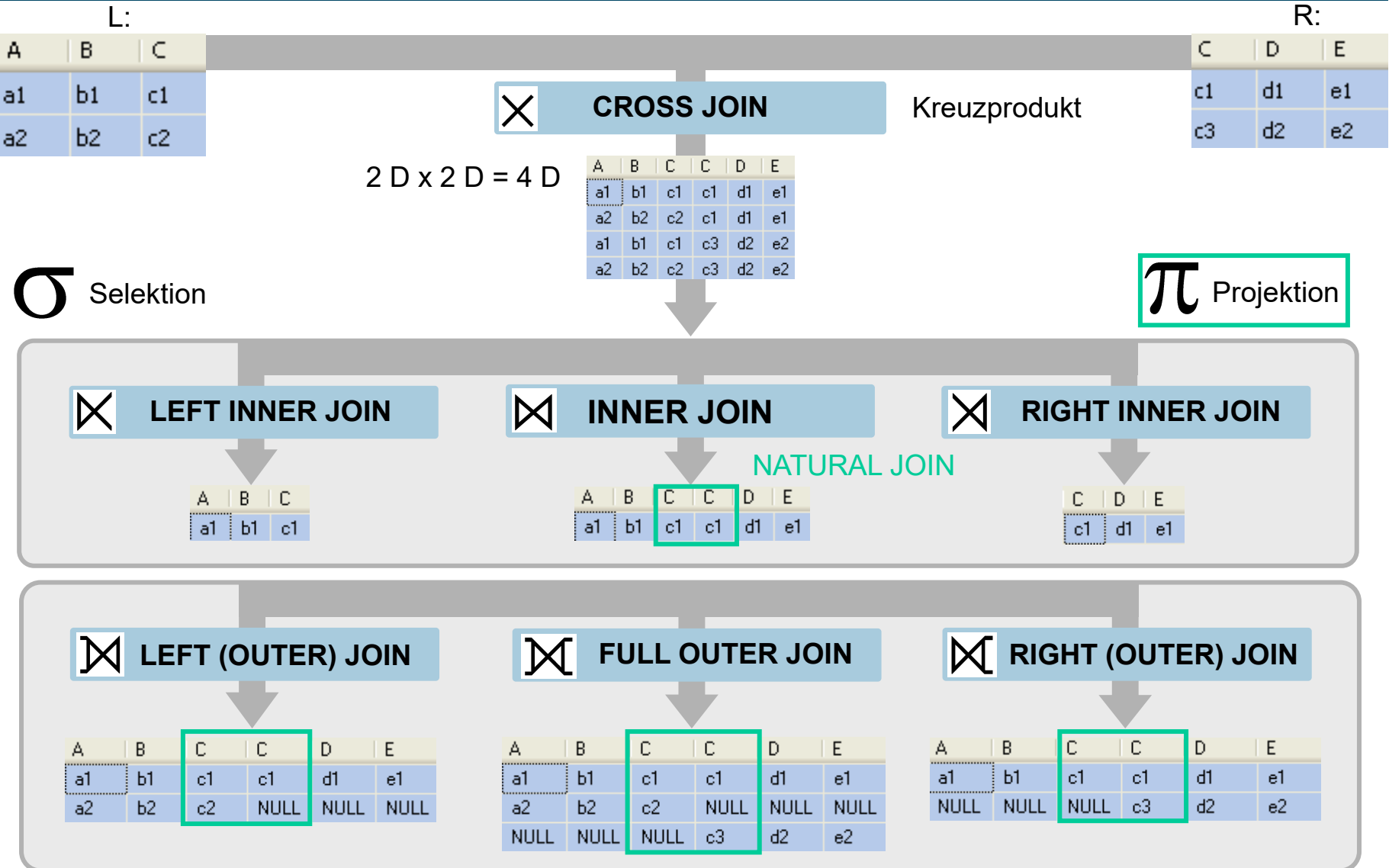


Zu einer einzigen Operation:



Kartesisches Produkt (CROSS JOIN) ausgenommen

Zusammenführung von Tabellen (JOIN)



Mehrfach-JOINS

Ein einzelner JOIN führt immer genau zwei Tabellen zusammen. Im folgenden Beispiel könnten beide Schreibweisen zu dem Eindruck führen, es würden mehr als zwei Tabellen ‚gejoint‘. Tatsächlich ist dies aber nicht der Fall. Der Ausdruck wird in der Regel von links nach rechts abgearbeitet, so dass zunächst die Relationen R und S ‚gejoint‘ werden. Die Ergebnisrelation dieses JOINS wird sodann mit der Relation Z ‚gejoint‘.

Beispiel

```
SELECT A, B, C, D, E
FROM R, S, Z
WHERE R.A = S.A AND R.B = S.B AND S.C = Z.C
```

```
SELECT A, B, C, D, E
FROM R JOIN S ON R.A = S.A AND R.B = S.B
      JOIN Z ON S.C = Z.C
```

Bedeutung

Implizite Schreibweise
(alte Verbundtechnik)

Explizite Schreibweise

Um bei JOINS mit mehr als zwei Tabellen die Abarbeitungsreihenfolge festzulegen bzw. klarzumachen können JOIN-Klauseln geklammert werden. JOINS werden grundsätzlich zwischen zwei Tabellen ausgeführt; die betreffende Ergebnistabelle wird dann ggf. in einem weiteren JOIN verwendet. Sind JOIN-Klauseln nicht geklammert, erfolgt die Abarbeitung grundsätzlich von links nach rechts.

Abschließende Anmerkungen

Platzierung von Verknüpfungskriterien

► **Nachteile impliziter Schreibweise:**

Alle Kriterien (Verknüpfungs- und Suchkriterien) werden in der WHERE-Klausel einer Abfrage angegeben. Wenn das DBMS die Kriterien nicht dem JOIN zuordnen kann, so wird zunächst ein vollständiges Kreuzprodukt gebildet und erst am Ende der Abfrage per WHERE auf die relevante Menge reduziert. Speziell bei einer Abfolge von mehreren JOINS steigt dadurch die Verarbeitungsmenge und die Datenmenge unnötig an.

► **Vorteile expliziter Schreibweise:**

Gezielte Angabe von Verknüpfungsbedingungen in der FROM – Klausel hinter dem Schlüsselwort ON erhöht die Verarbeitungsgeschwindigkeit des DBMS und trägt der Übersichtlichkeit von SQL-Statements bei.



Abschließende Anmerkungen

Im Zusammenhang mit JOINS und normalisierten Tabellen sollte man einige Dinge im Hinterkopf behalten:



Ein JOIN ist und bleibt – je nach Tabellengröße – eine umfangreiche Aktion. Wenn die Performance ein kritischer Faktor ist, sollte man die verwendeten JOINS dahingehend untersuchen, ob:



- sämtliche in den Kriterien (ON) verwendeten **Spalten indiziert** sind,
- der JOIN tatsächlich **nötig** ist und nicht aus reiner Bequemlichkeit einer zweiten Abfrage (Unterabfrage) vorgezogen wird,
- die **Tabellenreihenfolge** optimal gewählt ist.

Als Faustregel gilt: Immer mit der kleinsten Tabelle beginnen.

Wenn das Datenbanksystem Unterabfragen (Subselects) unterstützt, kann es effizienter sein statt der gesamten Tabelle nur eine durch Kriterien beschränkte Teilmenge der Datensätze im JOIN zu verwenden.

Abschließende Anmerkungen

Die Reihenfolge der Tabellen und die Platzierung von Kriterien können sowohl die Performance als auch das Ergebnis beeinflussen!

Bei SELECT-Anweisungen ist besonders darauf zu achten, dass keine unnötigen Spalten (v. a. durch die Angabe von Stern (*)), sondern nur die wirklich benötigten Spalten abgefragt werden.

Gründe:

- Erhöhter Speicherbedarf des Servers
- Erhöhte Netzwerkauslastung durch Übertragung unnötiger Daten



Spalten mit SELECT auswählen, mit WHERE weitere Bedingungen festlegen, die Tupel mit GROUP BY gruppieren, Aggregatfunktionen anwenden, Gruppierungen mit HAVING auswählen, das Ergebnis mit ORDER BY sortieren und so weiter und so fort...

Lesen von Datensätzen aus Tabelle(n)

SELECT-Befehlssatz ausführlich:

SELECT

[DISTINCT I ALL]

Attribut(e)

{bzw. Attributliste},

[[as] Neuer_Attributname],

{Neuer-Attributname ist optional}

{oder}

*

{für die Gesamtheit aller Attribute}

{oder}

(Unterselectanweisung)

FROM

Relation(en)

{oder}

(Unterselectanweisung)

{danach folgen optional keine oder eine Kombination der nachfolgenden Klauseln}

[WHERE]

Bedingung(en)

{Prüfung evtl. gegen Ergebnis einer Unterselectanweisung}

[ORDER BY]

Attribut(e) [ASC I DESC]

{bzw. Attributliste [ASC I DESC]},

[GROUP BY]

Attribut(e)

[HAVING]

Bedingung(en)

{nach **GROUP BY**

Prüfung evtl. gegen Ergebnis einer Unterselectanweisung}

Lesen von Datensätzen aus einer Tabelle

Das Beispiel liefert den gesamten Inhalt der Tabelle Artikel.

```
SELECT *
FROM Artikel
```



ANr	AName	APreis
100	Computer	5000,00
101	Drucker	1000,00
102	Kabel	500,00
103	Neuer Artikel	NULL

Die Sortierung nach der Spalte ‚ZName‘ kommt erst zum Tragen, wenn es mehrere gleiche Werte für Spalte ‚Ort‘ gibt.

```
SELECT Ort, PLZ, ZName, VName
FROM Kunde
ORDER BY Ort ASC, ZName DESC
ORDER BY 1 ASC, 3 DESC
```



Ort	PLZ	ZName	VName
Göppingen	730033	Herrmann	Klaus
Göppingen	73033	Burkhart	Maria
Heidelberg	69123	Müller	Hugo
Heidelberg	69123	Mayer	Georg
Heidelberg	69123	Kurz	Daniel

- ▶ Mit * werden alle Attribute einer Relation ausgewählt. Bezieht sich die Abfrage nur auf einzelne Relation, ist das Ergebnis eine Kopie der Relation selbst. Es können aber auch einzelne Attribute angegeben werden, die in die Ergebnisrelation einbezogen werden sollen (Projektion). Die Ergebnisrelation enthält dann nur die angegebenen Attribute.
- ▶ Die Sortierung der Ergebnisrelation kann mehrstufig (durch Angabe mehrerer Spalten getrennt nach Komma) von links nach rechts erfolgen.

WHERE - Klausel

Die Auswertung der Suchbedingung ergibt entweder den Wahrheitswert WAHR oder FALSCH. Mit Hilfe des Wahrheitswertes wird sodann vom Datenbankprozessor entschieden, ob die aktuell untersuchte Zeile in die Ergebnisrelation übernommen wird oder nicht. Ist der Wahrheitswert WAHR, wird die Zeile übernommen, bei FALSCH wird sie verworfen.

Syntax der Suchbedingung vereinfacht:

$$[\{ \text{AND} \mid \text{OR} \} \text{ [NOT] } \{ \langle \text{Prädikat} \rangle \mid (\langle \text{Suchbedingung} \rangle) \}]$$

Aus der Syntax ist zu entnehmen, dass eine Suchbedingung mindestens aus einem Prädikat besteht. Prädikate sind Bedingungsausdrücke, die bei Ihrer Auswertung einen Wahrheitswert ergeben. Alternativ kann eine Suchbedingung selbst aus einer in runden Klammern stehenden Suchbedingung (Unteranfrage) bestehen. Hiermit sind komplexe Schachtelungen von Suchbedingungen möglich.

Ein Prädikat kann mit dem booleschen Operator NOT auch logisch negiert werden. Mit dem booleschen Operator AND bzw. OR und NOT können Suchbedingungen aus mehreren Prädikaten bzw. Suchbedingungen verknüpft werden.

Basisprädikat

Werden in einem Prädikat **skalare Werte** miteinander verglichen, liegt ein Basisprädikat vor.

Mengenprädikat

Werden nicht einzelne Werte sondern **Wertemengen** in den Vergleich einbezogen, spricht man von Mengenprädikat.

Zusammengesetzter Prädikat

Ist eine Suchbedingung aus mehreren Prädikaten aufgebaut, so spricht man von einem zusammengesetzten Prädikat.

Übersicht Prädikate

Prädikatart

Prüfungsart

Basisprädikate

Einfacher Vergleichsausdruck

Vergleichsoperator

Einfache Vergleichsprüfung

Vorhandenseinprädikat

IS [NOT] NULL

Prüfung auf unbekannte Werte

Mengenprädikate

Enthaltenseinprädikat

[NOT] IN

Prüfung auf einen Attributwert
in einer Wertmenge

Bereichsselektionsprädikat

[NOT] BETWEEN

Bereichsprüfung

Quantifizierende Prädikate

[NOT] ANY bzw. SOME, ALL

Existenzprädikat

[NOT] EXISTS

Existenzprüfung

Vergleichsmusterprädikat

[NOT] LIKE

Musterprüfung

Zusammengesetzter Vergleichsausdruck

Zusammengesetzter Vergleichsausdruck

AND, OR

Verknüpfung mehrerer
Bedingungen (unter Beachtung
der Klammersetzung)

Mit **NOT** wird der Wahrheitswert einer logischen Aussage umgedreht.

Vorhandenseinprädikat

IS NULL

Mit diesem Prädikat kann man testen, ob ein Wert unbekannt (NULL) ist.

Ausgabe aller Artikel, die noch keinen Preis haben.

<Ausdruck> **IS [NOT] NULL**

```
SELECT *
FROM Artikel
WHERE APreis IS NULL
```

Enthaltenseinprädikat

IN

Ein Vergleichsausdruck ist WAHR, wenn der Ausdruck in der angegebenen Wertemenge (Aufzählung von zulässigen Werten) enthalten ist. Die Wertemenge kann entweder durch eine Aufzählung der zulässigen Werte (Ausdrucksliste) statisch oder als Ergebnis der Ausführung einer Unteranfrage dynamisch sein.

<Ausdruck> **[NOT] IN** (<Ausdrucksliste> | <Unteranfrage>)

4711 IN (4702, 4802, 4711)	ist wahr
4711 IN (4702, 4802, 4712)	ist falsch

```
... WHERE Kunde.KNr IN (SELECT KNr
                        FROM Rechnung)
```

Vergleichsmusterprädikat

LIKE

Mit diesem Prädikat lassen sich Muster-
vergleiche in Zeichenketten verwirklichen.

%: Platzhalter für beliebig viele Zeichen
_: Platzhalter für ein Zeichen

<Vergleichswert> **[NOT] LIKE** <Vergleichsmuster>

```
SELECT KNr, ZName, Ort
FROM Kunde
WHERE Zname LIKE 'M_ller%'
```


Bereichsprädikat

BETWEEN

Eine Suchbedingung ist WAHR, wenn der interessierende Ausdruck innerhalb bzw. außerhalb des mit BETWEEN und AND definierten Wertebereiches (ob ein Wert innerhalb bzw. außerhalb eines Intervalls) liegt. Die beiden Bereichsgrenzen sind im Wertebereich enthalten. Sie und der Ausdruck müssen vom kompatiblen Datentypen sein.

<Ausdruck> **[NOT] BETWEEN** <Ausdruck> **AND** <Ausdruck>

...WHERE Betrag BETWEEN 100 AND 1000

5 BETWEEN 3 AND 10

ist wahr

1 BETWEEN 3 AND 10

ist falsch

5 NOT BETWEEN 3 AND 10

ist falsch

1 NOT BETWEEN 3 AND 10

ist wahr

Existenzprädikat

EXISTS

Ein Vergleichsausdruck ist WAHR, wenn die Unteranfrage mindestens eine Tabellenzeile als Ergebnis liefert. Ist dies der Fall, so wird die aktuelle Zeile der Hauptanfrage in das Ergebnis übernommen.

Ausgabe aller Kunden, die mindestens eine Rechnung erhalten haben.

Ausgabe aller Kunden, die noch keine Rechnung erhalten haben.

[NOT] EXISTS (<Unteranfrage>)

SELECT KNr, ZName, VName

FROM Kunde K

WHERE **EXISTS** (SELECT *

FROM Rechnung R

WHERE K.KNr = R.KNr)

SELECT KNr, ZName, VName

FROM Kunde K

WHERE **NOT EXISTS** (SELECT *

FROM Rechnung R

WHERE K.KNr = R.KNr)

Übersicht Prädikate

Einfacher
Vergleichsausdruck

[NOT] <Ausdruck> <**Vergleichsoperator**> <Ausdruck>
[<logischer Operator>
[NOT] <Ausdruck> <**Vergleichsoperator**> <Ausdruck>] ...

Vorhandenseinprädikat

<Ausdruck> **IS** [NOT] **NULL**

IS NULL: Wert muss in Ergebnismenge unbekannt (leer) sein.

Enthaltenseinprädikat

<Ausdruck> [NOT] **IN** (<Ausdrucksliste> | <Unteranfrage>)

IN: Wert muss in Ergebnismenge enthalten sein.

Vergleichsmusterprädikat

<Vergleichswert> [NOT] **LIKE** <Vergleichsmuster>

LIKE: Zeichenreihenfolge muss in Zeichenkette enthalten sein.

Bereichsprädikat

<Ausdruck> [NOT] **BETWEEN** <Ausdruck> **AND** <Ausdruck>

BETWEEN: Wert muss im Intervall der Ergebnismenge enthalten sein.

Existenzprädikat

[NOT] **EXISTS** (<Unteranfrage>)

EXISTS: Ausdruck wird wahr, wenn die Menge mindestens ein Element enthält.

Quantifizierende Prädikate

<Ausdruck> <Vergleichsoperator> [NOT] **ANY** | **SOME** | **ALL** (<Unteranfrage>)

ALL: Vergleich muss für alle Elemente der Ergebnismenge erfüllt sein.

ANY bzw. SOME: Vergleich muss für mindestens ein Element der Menge erfüllt sein.

Synonyme Verwendung einiger Quantoren möglich:

$x = ANY(a, b, c) \leftrightarrow x IN(a, b, c)$

$x <> ALL(a, b, c) \leftrightarrow x NOT IN(a, b, c)$

Gruppenfunktionen

Welche Artikel werden insgesamt im Verkauf angeboten?

Jeder Artikel (Merkmal) soll in der Ergebnisliste nur einmal auftreten!

ANr	AName	APreis
100	Computer	2,0000
101	Computer	4,0000
102	Computer	3,0000
103	Drucker	7,0000
104	Drucker	1,0000

```
SELECT DISTINCT AName
FROM Artikel
```



```
SELECT AName
FROM Artikel
GROUP BY AName
```



AName
Computer
Drucker



Werte?

- Die Angabe **DISTINCT** bewirkt, dass doppelte Tupel aus der Ergebnisrelation entfernt werden. Ohne Angabe von **DISTINCT** (Voreinstellung für **ALL**) würden alle Tupel, auch die mehrfach vorkommenden, angezeigt werden.

Gruppenfunktionen

- ▶ Berechnungen, die auf Werte aus mehreren Zeilen zurückgreifen, benötigen **Gruppenfunktionen**. In der Regel werden Gruppenfunktionen in Kombination mit einer Gruppierung (GROUP BY – Klausel) eingesetzt. Mit Gruppierungen lassen sich Aufgabenstellungen lösen, in denen Schlüsselbegriffe wie „je“ oder „pro“ vorkommen, wie zum Beispiel:
 - Umsatz je Kunde
 - Stück je Artikelgruppe
 - Einkäufe pro Quartal
- ▶ Die bisherigen Techniken lieferten einen Stapel von Zeilen zurück. Wird eine Gruppenfunktion verwendet, so wird der Zeilenstapel auf eine Zelle, auf jeweils **nur exakt einen Wert** gestaucht (zusammengefasst, aggregiert, konsolidiert). Damit dienen Gruppenfunktionen der Konsolidierung (Aggregation, Gruppierung, Zusammenfassung, Ermittlung) von Kennziffern nach bestimmten Kriterien.
- ▶ Über entsprechende Parameterangaben innerhalb von Gruppenfunktionen lässt sich die Art der Zusammenfassung der einzelnen Werte der Zeilen einer Spalte festlegen.
- ▶ Gruppenfunktionen können auch ohne Verwendung von GROUP BY benutzt werden. Dann dürfen allerdings in der Attributliste der Projektion des SELECT-Befehls außer den Gruppenfunktionen keine weiteren Attributnamen auftreten.

Gruppenfunktionen

Die nachfolgende Grafik zeigt das Schema einer Gruppierung an. Es wird nach der Spalte „AName“ gruppiert, die Werte werden je nach ausgewählter **Aggregatfunktion** zusammengefasst.

AName	APreis
Computer	2.000
Computer	4.000
Computer	3.000
Drucker	7.000
Drucker	1.000

Gruppierung

Computer

Drucker

Aggregatfunktionen

Minimum	Maximum	Durchschnitt	Summe	Anzahl
2	4	3	9	3
1	7	4	8	2

Gruppenfunktionen

Es gibt folgende **5** Aggregatfunktionen:

MIN(Attribut)

- Berechnung des kleinsten Wertes einer Spalte.
- NULL – Werte werden in der Berechnung nicht berücksichtigt.

MAX(Attribut)

- Berechnung des größten Wertes einer Spalte.

AVG(Attribut)

- Berechnung des Mittelwertes (den Durchschnitt aller Werte) einer Spalte.

SUM(
[DISTINCT] Attribut)

- Berechnung der Summe aller Werte einer Spalte.
- DISTINCT: Mehrfach auftretende Werte werden nur einmal summiert.

COUNT(* |
[DISTINCT] Attribut)

- Ermittlung der Anzahl der Tupel einer Spalte.
- COUNT(*): Anzahl aller Tupel der Ergebnisrelation ohne NULL-Werte.
- COUNT(Attribut): Anzahl aller Tupel der Ergebnisrelation inklusive NULL-Werte.
- DISTINCT: Zusammenzählung nur der voneinander verschiedenen Werte einer Spalte

Gruppenfunktionen

Summe

Ermittlung der Summe aller Werte je Artikelgruppe.

```
SELECT AName, SUM(APreis) AS PreisKummuliert
FROM Artikel
GROUP BY AName
```

ANr	AName	APreis
100	Computer	2,0000
101	Computer	4,0000
102	Computer	3,0000
103	Drucker	7,0000
104	Drucker	1,0000

AName	PreisKummuliert
Computer	9,00
Drucker	8,00

Anzahl

Ermittlung der Anzahl der Artikel (Datensätze).

```
SELECT COUNT(*)
FROM Artikel
```

[Kein Spaltenname]
5

Ermittlung der Anzahl der Artikelgruppen.

```
SELECT COUNT(DISTINCT AName) AS AnzahlAG
FROM Artikel
```

AnzahlAG
2

Minimum

Ermittlung des Preises des günstigsten Artikels.

```
SELECT MIN(APreis) AS MinPreis
FROM Artikel
```

MinPreis
1,00

Maximum

Ermittlung des Preises des teuersten Artikels je Artikelgruppe.

```
SELECT AName, MAX(APreis) AS MaxPreis
FROM Artikel
GROUP BY AName
```

AName	MaxPreis
Computer	4,00
Drucker	7,00

Durchschnitt

Ermittlung des Durchschnittspreises je Artikelgruppe.

```
SELECT AName, AVG(APreis) AS Durchschnittspreis
FROM Artikel
GROUP BY AName
```

AName	Durchschnittspreis
Computer	3,00
Drucker	4,00

Gruppenfunktionen

Ermittlung der Summe aller
Artikelpreise.

```
SELECT SUM(APreis) AS PreisKummuliert
FROM Artikel
```

PreisKummuliert
17

Ermittlung der Summe aller
Preise je Artikelgruppe.

```
SELECT AName, SUM(APreis) AS PreisKummuliert
FROM Artikel
```



viele Werte!

ein Wert!

} Fehlerhaftes SQL-Statement!
„Passt nicht zusammen!“
(Würde 1NF verletzen)

Aggregatfunktionen werden auf ganze Spalten bzw. Wertegruppierungen angewandt und liefern jeweils **nur exakt einen Wert** pro Spalte bzw. Gruppe zurück!

```
SELECT AName, SUM(APreis) AS PreisKummuliert
FROM Artikel
GROUP BY AName
```

AName	PreisKummuliert
Computer	9
Drucker	8

Ermittlung des größten Preises
je Artikelgruppe bzw. Artikel.

```
SELECT AName, ANr, MAX(APreis) AS maxPreis
FROM Artikel
GROUP BY AName
```



Sinnvolle
SQL-
Abfrage?

```
SELECT AName, ANr, MAX(APreis) AS maxPreis
FROM Artikel
GROUP BY AName, ANr
```

AName	ANr	maxPreis
Computer	100	2
Computer	101	4
Drucker	102	3
Computer	103	7
Drucker	104	1

Gruppenfunktionen mit HAVING-Klausel

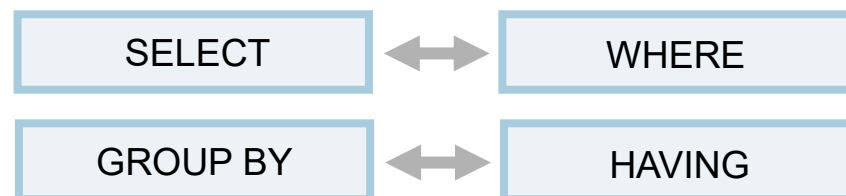
SELECT-Befehlssatz ausführlich:

SELECT	[DISTINCT ALL] Attribut(e) [[as] Neuer_Attributname], {oder} * {oder} (Unterselectanweisung)	{bzw. Attributliste}, {Neuer_Attributname ist optional} {für die Gesamtheit aller Attribute}
FROM	Relation(en) {oder} (Unterselectanweisung)	
{danach folgen optional keine oder eine Kombination der nachfolgenden Klauseln}		
[WHERE]	Bedingung(en)	{Prüfung evtl. gegen Ergebnis einer Unterselectanweisung}
[ORDER BY]	Attribut(e) [ASC DESC]	{bzw. Attributliste [ASC DESC]},
[GROUP BY]	Attribut(e)	
[HAVING]	Bedingung(en)	{nach GROUP BY Prüfung evtl. gegen Ergebnis einer Unterselectanweisung}

- ▶ Mit HAVING können **Bedingungen für Gruppierungen** definiert werden.
Nur Gruppierungen, welche die Bedingung erfüllen, werden in die Ergebnisrelation aufgenommen.
- ▶ HAVING verhält sich zu GROUP BY wie WHERE zu SELECT.

Auswahl von Spalte(n)

Bedingung(en)



Gruppenfunktionen mit HAVING - Klausel

Ermittlung für jede Artikelgruppe die Summe der zugehörigen Artikelpreise. Man möchte aber nur die Artikelgruppe angezeigt bekommen, bei der die Summe größer als 8 ist.

```
SELECT AName, SUM(APreis)
FROM Artikel
WHERE SUM(APreis) > 8
GROUP BY AName
```



In einer WHERE-Bedingung dürfen keine Aggregatfunktionswerte als Vergleichswerte benutzt werden.
Für solche Fälle wird die HAVING – Klausel eingesetzt!

```
SELECT  AName
FROM    Artikel
GROUP BY AName
HAVING SUM(APreis) > 8
```



AName
Computer

```
SELECT  AName, SUM(APreis) AS SUMME
FROM    Artikel
GROUP BY AName
HAVING SUM(APreis) > 8
```

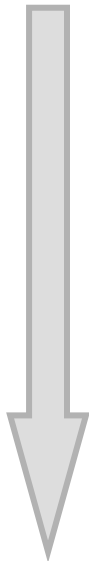


AName	SUMME
Computer	9

(fiktive) Verarbeitungsreihenfolge einer SELECT-Anweisung

1. FROM: Auswahl der Tabelle(n)
2. WHERE: Selektiert die Tupel, die der Bedingung genügen
3. GROUP BY: Gruppiert die Tupel auf Basis gleicher Werte
4. HAVING: Selektiert Gruppen, die der Bedingung genügen
5. SELECT: Selektiert Attribute
6. ORDER BY: Sortiert Tupel

RNr	KNr	Datum	Betrag
100	100	2004-04-04 12:12:00	13000,00
101	100	2004-04-04 00:00:00	2000,00
102	101	2004-04-05 00:00:00	5000,00



```

SELECT      KNr, Sum(Betrag) AS Gesamtbetrag
FROM        Rechnung
WHERE       Datum BETWEEN '04.04.2004' AND '05.04.2004'
GROUP BY   KNr
HAVING      SUM(Betrag) > 2000
ORDER BY    Gesamtbetrag ASC
    
```


KNr	Gesamtbetrag
101	5000,00
100	15000,00

Gesucht werden Kunden, die in einem bestimmten Zeitraum Rechnungen mit einem Gesamtbetrag (kumuliert) über 2.000 ausgestellt bekommen haben. Das Ergebnis soll sortiert nach KNr mit Angabe der Gesamtbeträge in absteigender Reihenfolge ausgegeben werden.

Unteranfrage

Ermittlung des Preises des günstigsten Artikels.


```
SELECT MIN(APreis) AS MinPreis
FROM Artikel
```



MinPreis
1

Ermittlung von *Artikeldaten* des günstigsten Artikels.

```
SELECT *
FROM Artikel
WHERE APreis = ( SELECT MIN(APreis)
                  FROM Artikel )
```



ANr	AName	APreis
104	Drucker	1

```
SELECT A1.*
FROM Artikel A1
WHERE A1.APreis = ( SELECT MIN(A2.APreis)
                    FROM Artikel A2 )
```

Verarbeitung dieser komplexen Bedingung:

Die Tabelle Artikel existiert hier in 2 Versionen: In der Version A wird sie Satz für Satz gelesen und bei jedem Satz erfolgt in einem Subquery (= Unterabfrage) die Anfrage an die Version B der Artikeltabelle:

„*Was ist der kleinste vorkommende Preis?*“

Anschließend erfolgt die Entscheidung, ob der Satz aus der Version A in die Ergebnistabelle übernommen wird oder nicht.

Unteranfragen im SELECT-Befehlssatz

SELECT...

Ermittlung der Preise aller Artikel, ihres Durchschnittspreises sowie der Differenz zwischen dem jeweiligen Artikelpreis und dem Durchschnittspreis.

ANr	AName	APreis	Durchschnittspreis	Differenz
100	Computer	5000,00	2166,6666	2833,3334
101	Drucker	1000,00	2166,6666	-1166,6666
102	Kabel	500,00	2166,6666	-1666,6666

```
SELECT ANr, AName, APreis,
       (SELECT AVG(APreis) FROM Artikel) AS Durchschnittspreis,
       APreis - (SELECT AVG(APreis) FROM Artikel) AS Differenz
FROM Artikel
```

FROM...

Ermittlung des maximalen Umsatzes pro Kunde.

KNr	ZName	VName	Maxumsatz
100	Müller	Hugo	15000,00
101	Mayer	Georg	5000,00

```
SELECT K.KNr, ZName, VName, X.Maxumsatz
FROM Kunde AS K INNER JOIN
      (SELECT R.KNr, SUM(Betrag) AS Maxumsatz
      FROM Rechnung R
      GROUP BY R.KNr) AS X
ON K.KNr = X.KNr
```

WHERE...

Ermittlung des teuersten Artikels.

ANr	AName	APreis
100	Computer	5000,00

```
SELECT A.*
FROM Artikel AS A
WHERE A.APreis = (SELECT MAX(A.APreis)
                  FROM Artikel AS A);
```

Unteranfrage

Unteranfragen werden nach zweierlei Gesichtspunkten unterschieden:

► Liefert die Unteranfrage als Ergebnis einen oder mehrere Werte?

Liefert die Unteranfrage einen skalaren Wert, kann dieser in Basisprädikaten getestet werden.
Liefert sie jedoch mehrere Werte, müssen diese in Mengenprädikaten getestet werden.

Ergebnis der Abfrage:

Einfache Unteranfrage

Mengenunteranfrage



Prädikatart:

Basisprädikat

Mengenprädikat

► Wird die Unteranfrage nur einmal ausgeführt und ihr Ergebnis sodann in die Hauptanfrage eingesetzt oder wird sie für jede Zeile der Hauptanfrage einmal, also mehrfach, ausgeführt?

Art der Unteranfrage:

Unkorrelierte Unteranfrage

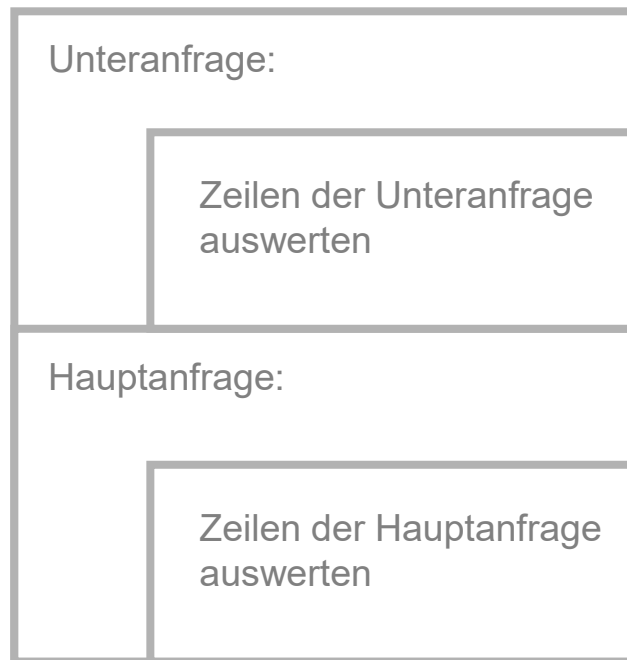


Korrelierte Unteranfrage

Unteranfrage

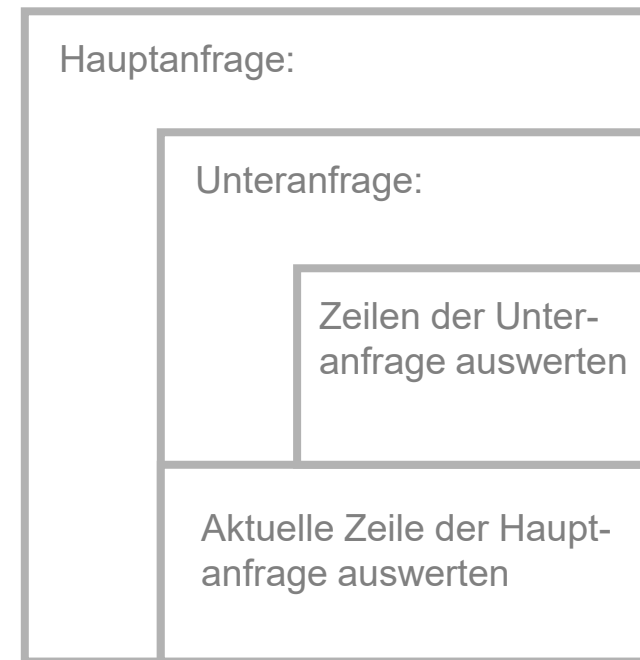
Ablauf bei einer **unkorrelierten** Unteranfrage

- ✓ *statische Unteranfrage*



Ablauf bei einer **korrelierten** Unteranfrage

- ✓ *wiederholte Unteranfrage*
- ✓ *synchronisierte Unteranfrage*
- ✓ *bedingte Unteranfrage*



Unteranfrage

SQL-Statement für **unkorrelierte** Unteranfrage:

Welche Kunden haben Rechnungen mit einem Rechnungsbetrag größer als 5.000 erhalten?

```
SELECT  KNr, ZName, VName
FROM    Kunde
WHERE   KNr IN (SELECT KNr
                FROM Rechnung
                WHERE betrag>5000)
```

SQL-Statement für **korrelierte** Unteranfrage:

```
SELECT  K.KNr, ZName, VName
FROM    Kunde K
WHERE   5000 < (SELECT MAX(Betrag)
                FROM Rechnung R
                WHERE K.KNr = R.KNr)
```

KNr	ZName	VName
100	Müller	Hugo
101	Mayer	Georg



Welche Artikel wurden mehr als 3 mal verkauft?

```
SELECT  AName
FROM    Artikel
WHERE   ANr IN (SELECT ANr
                FROM Position
                WHERE SUM(Anzahl)>3
                GROUP BY ANr)
```

```
SELECT  A.AName
FROM    Artikel A
WHERE   3 < (SELECT SUM (Anzahl)
                FROM Position P
                WHERE A.ANr = P.ANr)
```

AName
Computer
Drucker



Anmerkungen zu Unteranfragen

- Korrelierte Unterabfragen können auch in einer HAVING-Klausel der übergeordneten Abfrage verwendet werden, ebenso wie statische Unterabfragen, beziehen sie sich dann auf die jeweilige Gruppierung.

HAVING...

Welche Artikel wurden überdurchschnittlich verkauft?

AName	AnzahlVerkäufe
Computer	3
Drucker	4

```
SELECT  AName, SUM(Anzahl)AS AnzahlVerkäufe
FROM    Artikel A INNER JOIN Position P ON A.ANr = P.ANr
GROUP BY AName
HAVING  SUM(Anzahl) >= (SELECT SUM (Anzahl) / ((SELECT COUNT(APreis) FROM Artikel))
                        FROM Position P)
```

- Der Zugriff auf Ergebniswert/menge einer Unterabfrage ist nur mit den vorgestellten Mengenquantoren bzw. Mengenprädikaten möglich und nicht z.B. durch die von früher bekannten Aggregatfunktionen!

SELECT __FROM__WHERE__ ~~=Max~~(SELECT __...



Anmerkungen zu Unteranfragen

▶ Unteranfragen können häufig **ergebnisgleich** zu einer entsprechenden Abfrage mit **Joins** verwendet werden.

Ausgabe aller Kunden, die mindestens eine Rechnung erhalten haben.

*Korrelierte
Unteranfrage*

```
SELECT KNr, ZName, VName
FROM Kunde K
WHERE EXISTS(SELECT * FROM Rechnung R WHERE K.KNr = R.KNr)
```

*Statische
Unteranfrage*

```
SELECT KNr, ZName, VName
FROM Kunde K
WHERE K.Knr IN (SELECT Knr FROM Rechnung)
```

EQUI JOIN

```
SELECT DISTINCT K.KNr, ZName, VName
FROM Kunde K INNER JOIN Rechnung R ON K.KNr = R.KNr
```

▶ Mit Hilfe der Unteranfragen können insbesondere aufwändige Verarbeitungen der JOINS an solche Stellen in einer SELECT-Anweisung verlagert werden, an denen komplexe Verarbeitungen nur noch über Teildatenbestände mit geringer Datensatzanzahl ausgeführt werden müssen. Zum Teil lassen sich Verbunde aber auch ganz durch Unteranfragen ersetzen.

Anmerkungen zu Unteranfragen

- ▶ Im Gegensatz zu imperativen Programmiersprachen, die genauen Ablauf einer Berechnung festlegen, beschreibt eine SQL-Anfrage deklarativ lediglich die zu liefernde Daten. RDBS entscheidet selbstständig, wie Ergebnis effizient berechnet werden kann (Details später).
- ▶ Korrelierte Unterabfragen sind oftmals „**Performance-Killer**“ und sollten für leistungskritische Abfragen möglichst vermieden werden. Der MS SQL Server wandelt intern im Rahmen der Anfrageoptimierung **häufig** korrelierte Unteranfragen in statische Unteranfragen um.
- ▶ Beim Entwickeln solcher Abfragen ist daher zu beachten, dass die **Korrelationsnamen** für die Tabelle(n) in der untergeordneten Abfrage anders gewählt werden als in der übergeordneten Abfrage. Enthält die Unterabfrage Ausdrücke, die nicht aufgelöst werden können, so wird in der übergeordneten Abfrage nach einem entsprechenden Ausdruck gesucht. Wird ein solcher gefunden, handelt es sich um eine korrelierende Unterabfrage.

Gliederung

3.

Datenbankimplementierung



DBMS und SQL



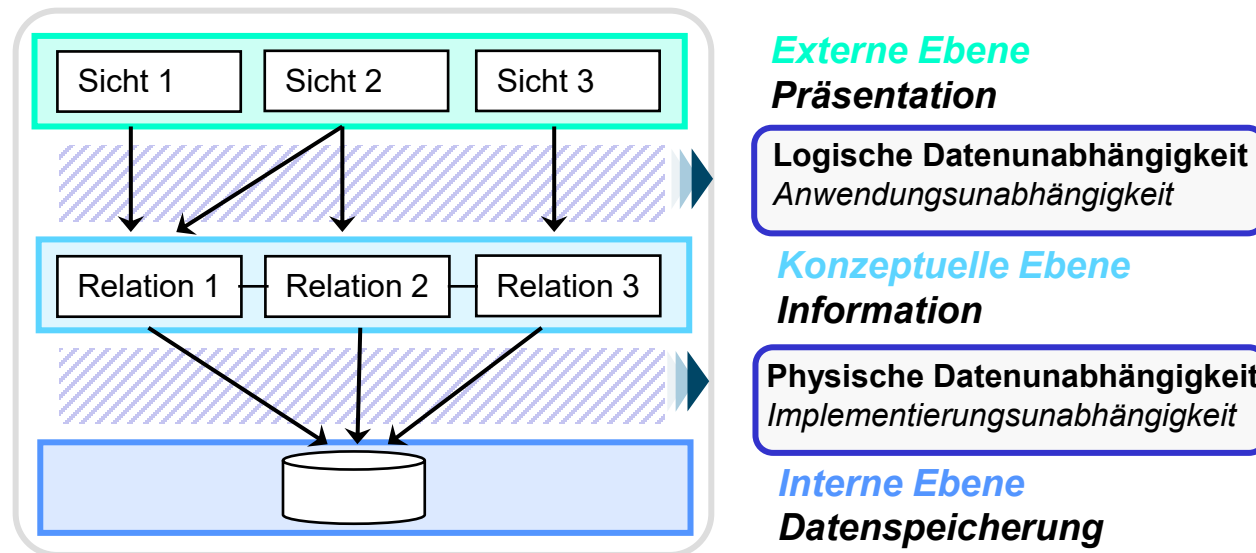
SQL – Sprachelemente inkl. Datenintegrität



Sichten

Sichten zur Gewährleistung der logischen Datenunabhängigkeit

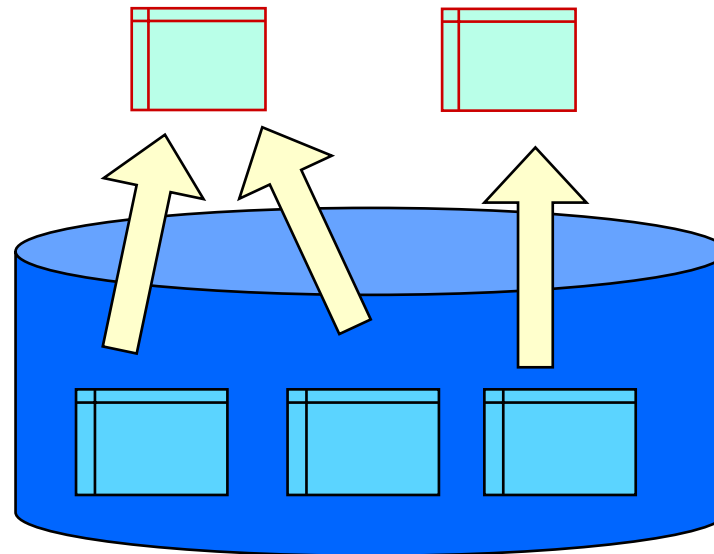
- Die in den Datenbanken eines Datenbanksystems gespeicherten Informationen werden normalerweise aus verschiedenen bestimmten, fachlich motivierten Blickwinkeln heraus betrachtet. Hierbei sind nur Teile der gespeicherten Daten von Interesse. Eine solche Betrachtungsweise von Ausschnitten von Daten aus Datenbank(en) wird als Benutzer-**Sicht** (VIEW) bezeichnet.



- Durch das SQL-Sprachkonzept der VIEWS kann einem Benutzer eine spezielle Sicht auf einen Datenbestand ermöglicht werden, die sich aus der Bereitstellung der Daten an fachlichen Erfordernissen orientiert. So wird einem Endbenutzer in Abhängigkeit eines konkreten Anwendungsfalls oftmals nur ein für seine Arbeit relevanter Ausschnitt aus den Daten der Datenbank angezeigt (benutzerbezogene Datenkapselung).
- Der Entwickler einer Datenbankanwendung hat somit die Aufgabe, die Sichten des Endanwenders auf den Datenbestand zu analysieren und entsprechend an der Benutzungsoberfläche bereitzustellen.

Das Konzept der Sichten

- ▶ Eine Sicht ist eine virtuelle Tabelle, deren Inhalt durch eine Abfrage, meist ein komplexes SQL-Befehl, definiert wird. „Virtuell“ heißt, dass in der Datenbank keine neue Tabelle angelegt wird, vielmehr wird sie bei jeder Verwendung zur Laufzeit aus einer oder mehreren Basistabellen intern neu berechnet. Man kann eine Sicht als eine Art Makro verstehen.
- ▶ Auf eine Sicht wird genauso zugegriffen wie auf eine Tabelle auch. Oft weiß der Endbenutzer gar nicht, ob er auf eine Tabelle oder eine Sicht zugreift. Es spielt auch keine Rolle. Eine Sicht muss wie eine Tabelle abgefragt werden, wenn man Daten über sie (und nicht aus ihr) erhalten möchte. Der Unterschied liegt daran, dass eine Sicht keine eigenen Daten besitzt. Wenn man auf eine Sicht zugreift, werden die Daten zu diesem Zeitpunkt über die mit ihr definierte SELECT-Anweisung „just in time“ ausgelesen. Die Daten kommen immer aus den der Sicht zugrunde liegenden Tabellen. Deshalb zeigen Sichten auch immer die aktuellen Daten an.



Sichten

*intensionale Datenbasis (IDB)
(hergeleitete Relationen)*

Basisrelationen

*extensionale Datenbasis (EDB)
(Basis-Relationen)*

VIEW anlegen

```
CREATE VIEW <Viewname> [(Spaltenliste)] AS  
<SELECT-Befehl>  
[WITH CHECK OPTION]
```

Viewname

Views werden als eigene Datenbankobjekte (so wie Tabellen) direkt in der Datenbank gespeichert, sind damit dauerhaft vorhanden und werden im Datenbankkatalog verwaltet. Der Viewname muss unter den View- und Tabellennamen innerhalb der Datenbank eindeutig sein.

Spaltenliste

Spaltenliste ist definiert als:
<Spaltenname> [, <Spaltenname>]...

In der Spaltenliste können Namen für die Spalten definiert werden, die den View strukturieren. Werden sie angegeben, so sind sie die extern sichtbaren Spaltennamen. Sie blenden somit die Sichtbarkeit der Spaltennamen der Basistabelle(n) aus.

Wird die Spaltenliste weggelassen, so werden die Spaltennamen der Basistabelle(n) übernommen, aus der bzw. denen das VIEW-Ergebnis gebildet wird (SELECT-Befehl). Sie sind dann auch bei der Viewanwendung nach außen hin sichtbar.

Die Anzahl der in der Spaltenliste definierten und extern sichtbaren Spaltennamen muss mit der Anzahl der Spalten bzw. Ausdrücken übereinstimmen, die sich aus dem SELECT-Befehl ergeben. Alternativ zur Spaltenliste können die Namen der Viewspalten auch mit AS-Klauseln in den jeweiligen Ausdrücken des SELECT-Befehls gebildet werden.

VIEW anlegen

```
CREATE VIEW <Viewname> [(Spaltenliste)] AS
<SELECT-Befehl>
[WITH CHECK OPTION]
```

SELECT-Befehl

Der SELECT-Befehl ist prinzipiell der gleiche SELECT-Befehl, der auch zur üblichen Datenwiedergewinnung verwendet wird. Lediglich die **ORDER BY- Klausel** darf nicht verwendet werden.

CHECK-Klausel

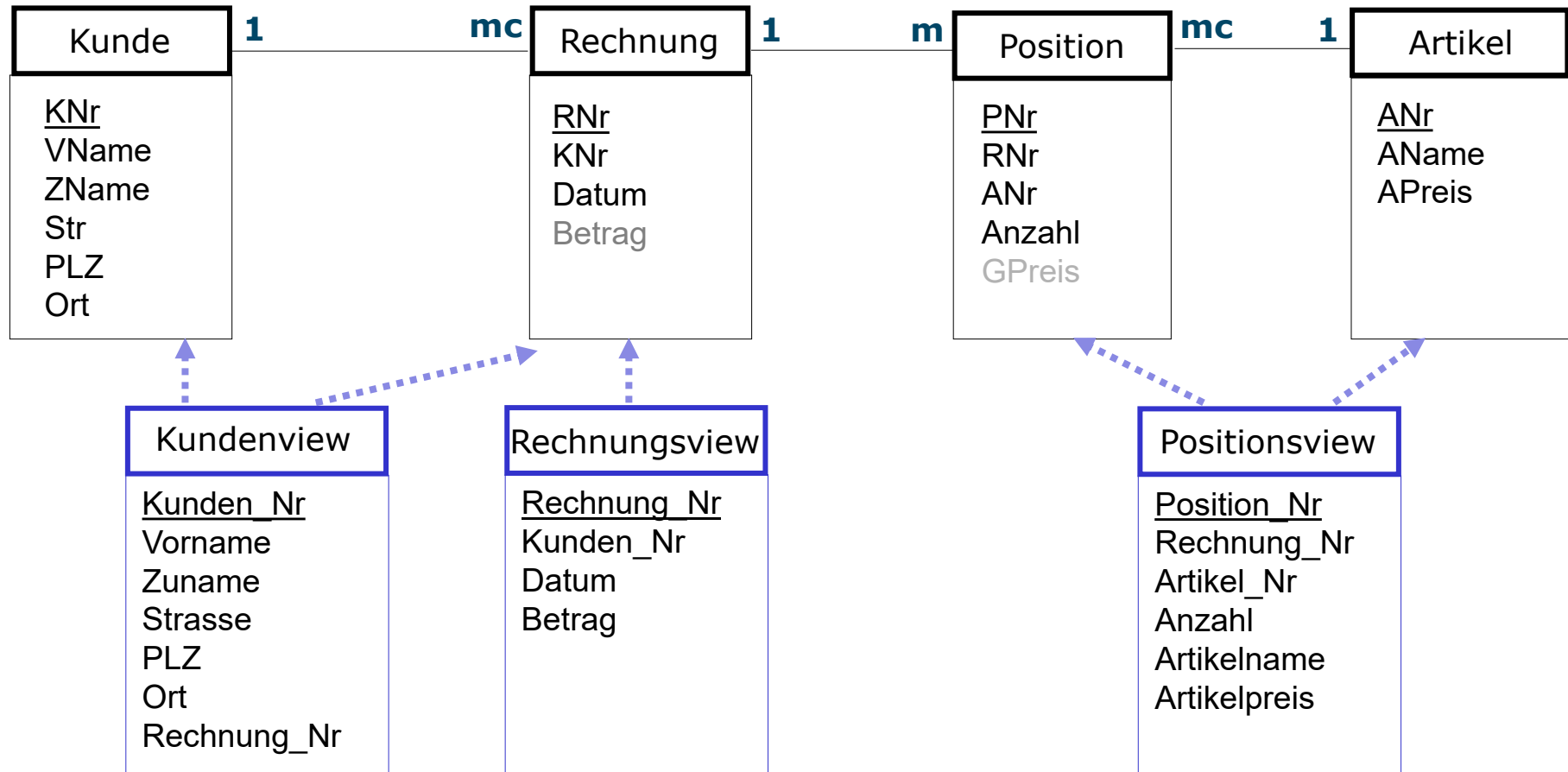
Die optionale CHECK-Klausel einer VIEW-Definition ermöglicht die Spezifikation von Einschränkungen der Veränderbarkeit eines VIEW-Inhalts; ist VIEW CHECK OPTION angegeben, so werden nur Einfügungen bzw. Veränderungen ausgeführt, welche die den View definierende Bedingungen erfüllen.

Derartige CHECK-Klauseln können sogar ‚von außen‘, also z.B. von Tabellen, welche in die VIEW-Definition eingehen, ererbt werden.

VIEWS anlegen und anwenden

Relationales Datenmodell des konzeptionelles Schemas

<<tables>>



<<views>>

Relationales Datenmodell des externen Schemas

VIEWS anlegen und anwenden

Views erstellen

```
CREATE VIEW Kundenview (Kunden_Nr, Vorname, Zuname, Strasse, PLZ, Ort, Rechnung_Nr) AS
SELECT      K.*, R.RNr
FROM        Kunde K JOIN Rechnung R
ON          K.KNr = R.KNr
```

```
CREATE VIEW Rechnungsvew (Rechnung_Nr, Kunden_Nr, Datum, Betrag) AS
SELECT      *
FROM        Rechnung
```

```
CREATE VIEW Positionsvew (Position_Nr, Rechnung_Nr, Artikel_Nr, Anzahl, Artikelname, Artikelpreis) AS
SELECT      P.PNr, P.RNr, P.ANr, P.Anzahl, A.AName, A.APreis
FROM        Position P JOIN Artikel A
ON          P.ANr = A.ANr
```

Views anwenden (Daten aus einer Sicht abrufen)

```
SELECT      *
FROM        Kundenview
WHERE       Rechnung_Nr = '100'
```



Kunden_Nr	Vorname	Zuname	Strasse	PLZ	Ort	Rechnung_Nr
100	Hugo	Müller	Gartenstr. 4a	69123	Heidelberg	100

VIEW auf VIEW

Ein VIEW kann sowohl aus Basisrelationen (physischen Tabellen) als auch aus anderen VIEWS definiert werden. Letztendlich liegen natürlich Basistabellen den Views zu Grunde.

Welche Artikel wurden überdurchschnittlich (oft) verkauft?

CREATE VIEW Durchschnittsverkauf (Durchschnittsverkauf) AS

SELECT SUM (Anzahl) / ((SELECT COUNT(APreis) FROM Artikel))
FROM Position P

SELECT *
FROM **Durchschnittsverkauf**

Durchschnittsverkauf
3

CREATE VIEW maxArtikel (Artikelbezeichnung, Anzahl_V Verkäufe) AS

SELECT AName, SUM(Anzahl)AS AnzahlVerkäufe
FROM Artikel A INNER JOIN Position P ON A.ANr = P.ANr
GROUP BY AName
HAVING SUM(Anzahl) >= (SELECT * FROM Durchschnittsverkauf)

SELECT *
FROM **maxArtikel**

Artikelbezeichnung	Anzahl_V Verkäufe
Computer	3
Drucker	4

SELECT ANr, Artikelbezeichnung, APreis, Anzahl_V Verkäufe
FROM **maxArtikel** INNER JOIN Artikel
ON Artikelbezeichnung = AName

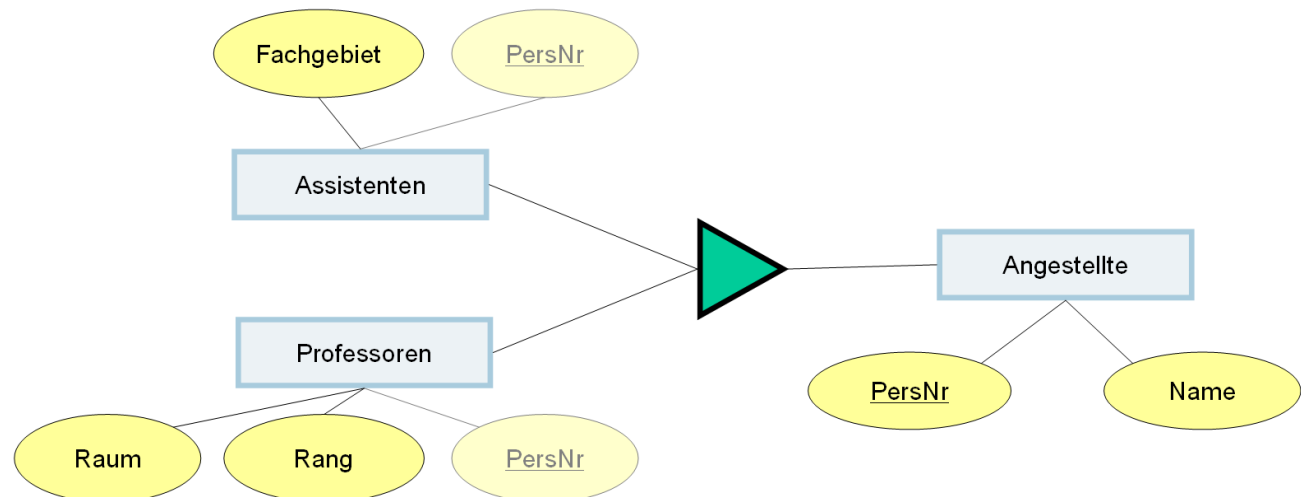
ANr	Artikelbezeichnung	APreis	Anzahl_V Verkäufe
100	Computer	5000,00	3
101	Drucker	1000,00	4

Sichten zur Modellierung von Generalisierungen

- Bei der Modellierung von Generalisierungen dienen Sichten zur Realisierung von Inklusion und Vererbung: Objekte (hier Tupel) eines Untertyps einer Generalisierungshierarchie sollen auch automatisch zu ihrem Obertyp gehören und die Attribute des Obertyps erben. Dabei kann entweder der Obertyp oder der Untertyp als Sicht definiert werden.

Relationale Modellierung der Generalisierung

Generalisierung?
Spezialisierung?
Vererbung?
Teilmengen?
Partiell/total?
Disjunkt?
FK / PK?



Entity-Typ Angestellte

Angestellte=((PersNr, Name),(PersNr))

Entity-Typ Professoren

Professoren=((PersNr, Rang, Raum),(PersNr))

Entity-Typ Assistenten

Assistenten=((PersNr, Fachgebiet),(PersNr))

Modellierungsmöglichkeiten für Generalisierungen

Untertypen als Sicht

```
CREATE TABLE Angestellte
( PersNr      INT NOT NULL,
  Name        VARCHAR(30) NOT NULL )
```

```
CREATE TABLE ProfDaten
( PersNr      INT NOT NULL,
  Rang        CHAR(2) NULL,
  Raum        INT NULL )
```

```
CREATE TABLE AssiDaten
( PersNr      INT NOT NULL,
  Fachgebiet  VARCHAR(30) NULL,
  Raum        INT NULL )
```

```
CREATE VIEW Professoren AS
SELECT *
FROM   Angestellte A, ProfDaten D
WHERE  A.PersNr = D.PersNr
```

```
CREATE VIEW Assistenten AS
SELECT *
FROM   Angestellte A, AssiDaten D
WHERE  A.PersNr = D.PersNr
```

Obertypen als Sicht

```
CREATE TABLE Professoren
( PersNr      INT NOT NULL,
  Name        VARCHAR(30) NOT NULL,
  Rang        CHAR(2) NULL,
  Raum        INT NULL )
```

```
CREATE TABLE Assistenten
( PersNr      INT NOT NULL,
  Name        VARCHAR(30) NOT NULL,
  Fachgebiet  VARCHAR(30) NULL,
  Raum        INT NULL )
```

```
CREATE TABLE Andere_Angestellte
( PersNr      INT NOT NULL,
  Name        VARCHAR(30) NOT NULL )
```

```
CREATE VIEW Alle_Angestellte AS
( SELECT PersNr, Name
  FROM Professoren )
UNION
( SELECT PersNr, Name
  FROM Assistenten )
UNION
( SELECT *
  FROM Andere_Angestellte )
```

geerbte
Attribut
e
vs.
speziell
e
Attribut
e



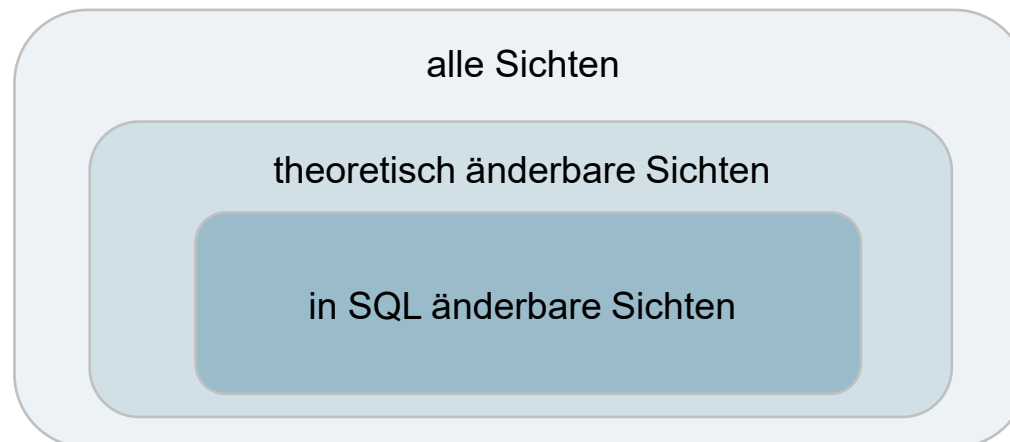
Die Beispiele zeigen, wie **Sichten zur Gewährleistung der logischen Datenunabhängigkeit** eingesetzt werden können. Die logische Datenunabhängigkeit schützt die Benutzer vor Veränderung am Datenbankschema. Unabhängig davon, ob beispielsweise der Ober- oder der Untertyp als Sicht definiert wurde, den Benutzern wird eine **einheitliche Schnittstelle** geboten.

Einschränkungen für VIEWS

▶ Mit VIEWS können Änderungen in der Datenbank durchgeführt werden. Allerdings sind bei der Anwendung von VIEWS vom Anwender einige Einschränkungen gegenüber der Anwendung von Basistabellen zu beachten. Diese Einschränkungen betreffen die Befehle zur Datenbankänderung (DELETE, UPDATE, INSERT).

▶ Im Allgemeinen sind Sichten veränderbar, wenn

- sie nur genau **eine Tabelle** (also Basisrelation oder Sicht) verwenden, die ebenfalls veränderbar sein muss, d. h. es liegen keine Verbunde (Joins) vor.
- in der SELECT-Liste nur **eindeutige Spaltennamen** stehen und der **Schlüssel der Basisrelation** enthalten ist und
- sie weder **Aggregatfunktionen**, noch Anweisungen wie **DISTINCT**, **GROUP BY** und **HAVING** enthalten,



Einschränkungen für VIEWS

DELETE

DELETE FROM Kundenview
WHERE Zuname= 'Müller'



UPDATE

UPDATE Rechnungsview
SET Datum = '5-4-2008'
WHERE Datum = '5-4-2004'

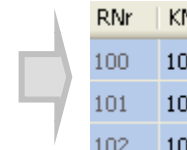


Tabelle Rechnung

RNr	KNr	Datum	Betrag
100	100	04.04.2004 12:12:00	13000,0000
101	100	04.04.2004 00:00:00	2000,0000
102	101	05.04.2008 00:00:00	5000,0000

CREATE VIEW
SELECT
FROM

WHERE

WITH CHECK OPTION

Rechnungsview (Rechnung_Nr, Kunden_Nr, Datum, Betrag) AS
*
Rechnung
Datum BETWEEN '01-01-2004' AND '31.12.2007'

UPDATE
SET
WHERE

Rechnungsview
Datum = '5-4-2008'
Datum = '5-4-2004'



INSERT

INSERT INTO Rechnungsview
(Kunden_Nr, Datum, Betrag)
VALUES (101, '5-4-2006', 2000)

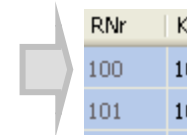


Tabelle Rechnung

RNr	KNr	Datum	Betrag
100	100	04.04.2004 12:12:00	13000,0000
101	100	04.04.2004 00:00:00	2000,0000
102	101	05.04.2004 00:00:00	5000,0000
103	101	05.04.2006 00:00:00	2000,0000

VIEW löschen

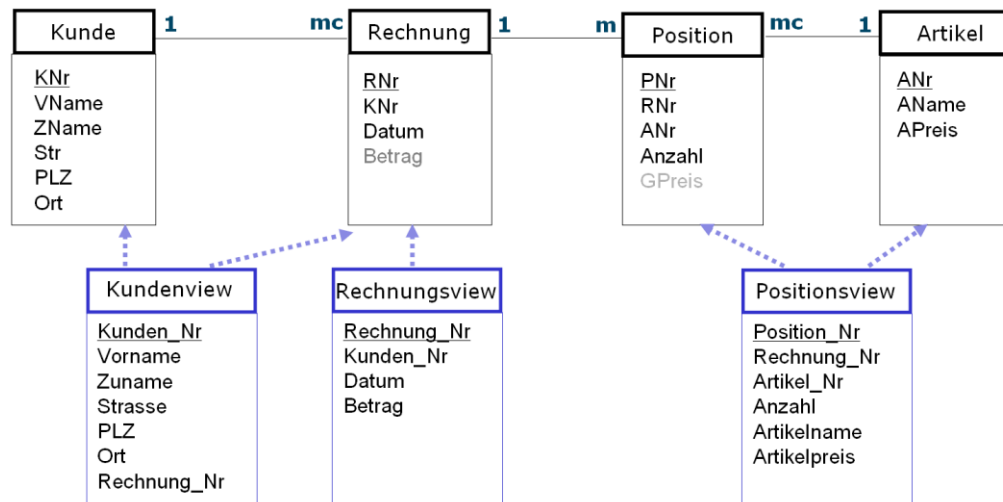
Bei diesem Vorgang werden die Metadaten des gelöschten VIEWS aus dem Katalog entfernt.

DROP VIEW <Viewname>

DROP VIEW Kundenview

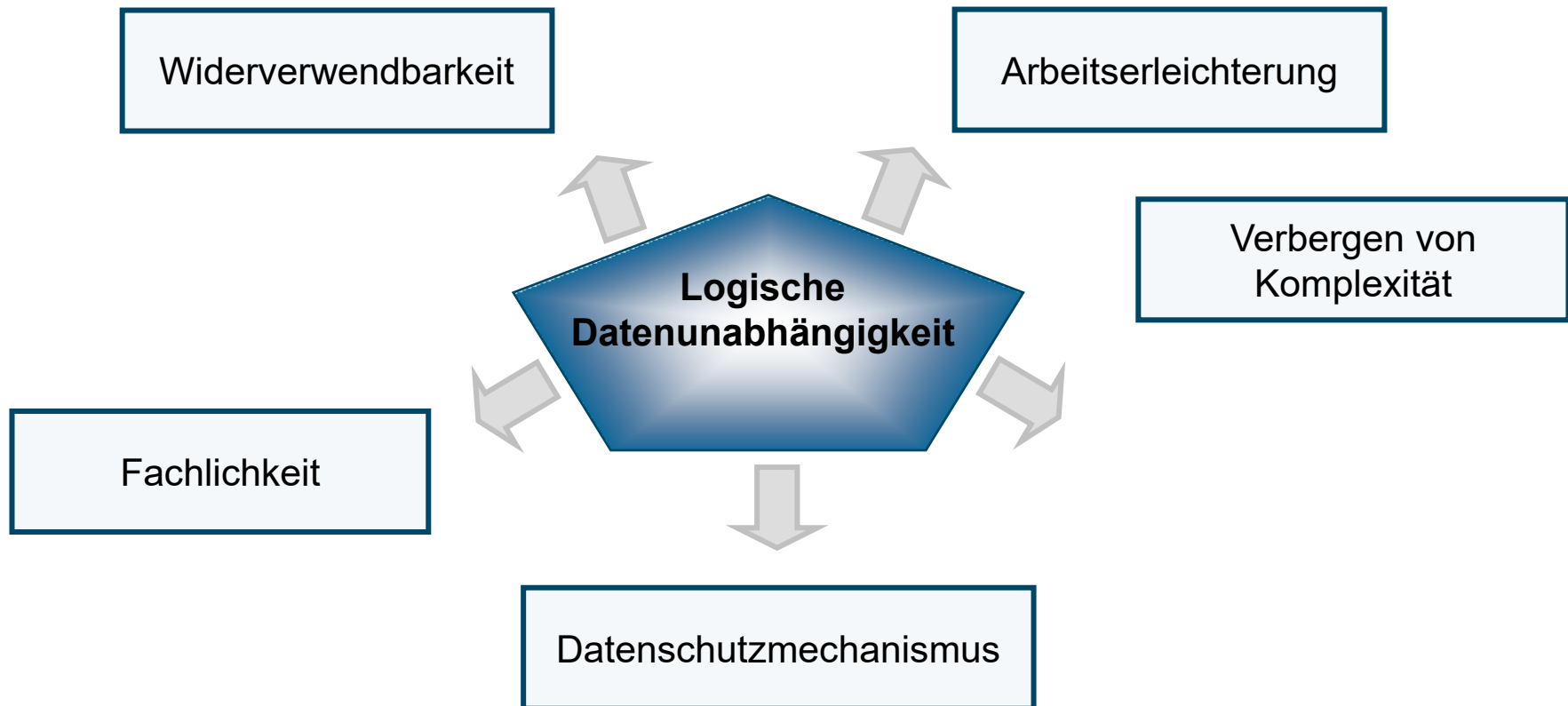
DROP VIEW Rechnungsview

DROP VIEW Positionsview



Gründe für den Einsatz von VIEWS

► Prinzipiell sind Sichten lediglich SELECT-Anweisungen. Diese sind auch für sich alleine anwendbar.
Was sind nun Gründe für den Einsatz von Sichten?



Gründe für den Einsatz von VIEWS

Fachlichkeit

- Ein wichtiges Konzept, um ein Datenbanksystem an die Bedürfnisse unterschiedlicher Benutzer(gruppen) anpassen zu können, sind Sichten.

Arbeitserleichterung

- Man spart sich sehr viel Arbeit, wenn man viele Tabellen nicht jedes Mal erneut miteinander verknüpfen muss, sondern das Zwischenergebnis bereits fertig vorliegen hat, um mit wenig Aufwand darauf zurückgreifen zu können.

Verbergen von Komplexität

- Realisierung komplexer Zusammenhänge (Verknüpfungen mehrerer Tabellen, Berechnungen usw.) in Form von Sichten kann den Zugriff auf Daten und damit die Benutzung der Datenbank vereinfachen. In einer Tabelle ist zu sehen, was eigentlich auf vielen Tabellen verteilt ist. Man benötigt dann recht einfache SQL-Anweisungen, um auf diese Sichten zuzugreifen.

Widerverwendbarkeit

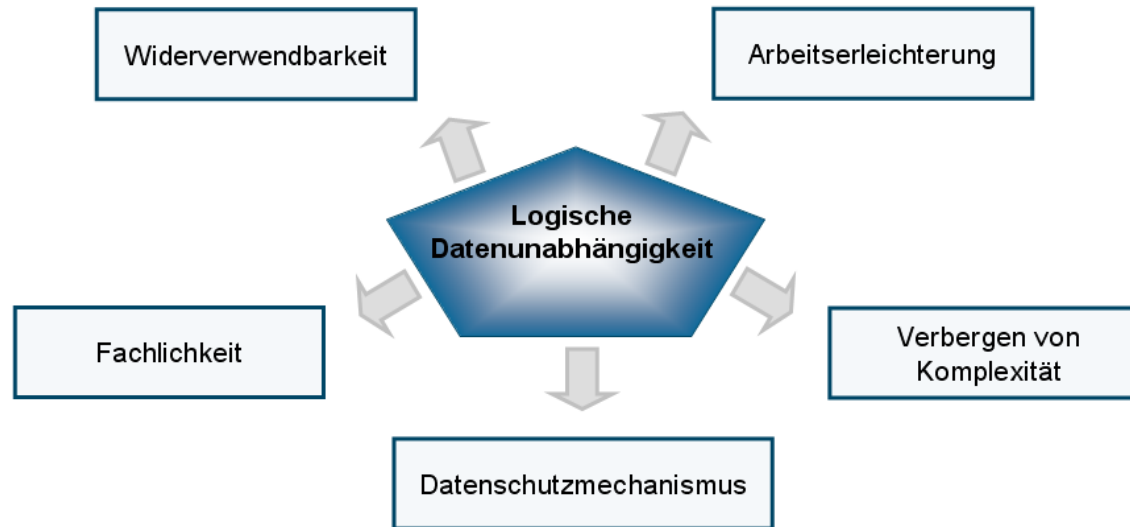
- Die Betrachtung der Daten kann durchaus wiederkehrend stattfinden, d. h., dass die gleichen Informationen zu unterschiedlichen Zeitpunkten wiederholt abgefragt werden.
- Das Datenbanksystem kann das Ergebnis dieser meist komplexen Abfragen unter Umständen cachen und somit beim zweiten Zugriff das Ergebnis schneller zur Verfügung stellen.

Gründe für den Einsatz von VIEWS

Datenschutz- Mechanismus

- Im Unternehmen gibt es Daten, die nicht jeder Mitarbeiter sehen darf (z.B. Personaldaten). Ein wesentlicher Datenschutzmechanismus besteht darin, nur den berechtigten Benutzern lediglich die für ihre Tätigkeit notwendigen Tabellen und Tabellenausschnitte zur Verfügung zu stellen.
- Sichten unterstützen für diesen Vorgang **indirekte Berechtigungen**. Ein Benutzer, der die Leseberechtigung für eine Sicht hat, darf die über die Sicht gelieferten Daten lesen, auch wenn er über keine Zugriffsberechtigung auf die zugrunde liegenden Tabellen verfügt.
- Somit bleiben dem Anwender die Einzelheiten, wie etwa die tatsächliche Struktur der Tabellen, die Verteilung der Daten über die verschiedenen Tabellen als auch die Komplexität der SELECT-Anweisung verborgen. Er muss sich auch nicht mit Ihnen befassen, um die notwendigen Selektionen erstellen zu können.
- Durch das Sichtenkonzept wird es möglich, auf der externen Ebene einer Datenbank Daten nach bestimmten Kriterien sichtbar bzw. unsichtbar zu machen (Einschränkung auf Spalten- bzw. Datensatzebene) und somit das Datenbankschema vor der Veränderung durch den Benutzer zu schützen.

Gründe für den Einsatz von VIEWS



Logische Datenunabhängigkeit:

- Ein VIEW wirkt wie ein Transformator (Datenstruktur-, Datentyp- und Zugriffspfad-Unabhängigkeit). Er passt die Erwartungen des externen Benutzers an die internen Gegebenheiten in den Tabellen und dem sie enthaltenden konzeptionellen Schema an.