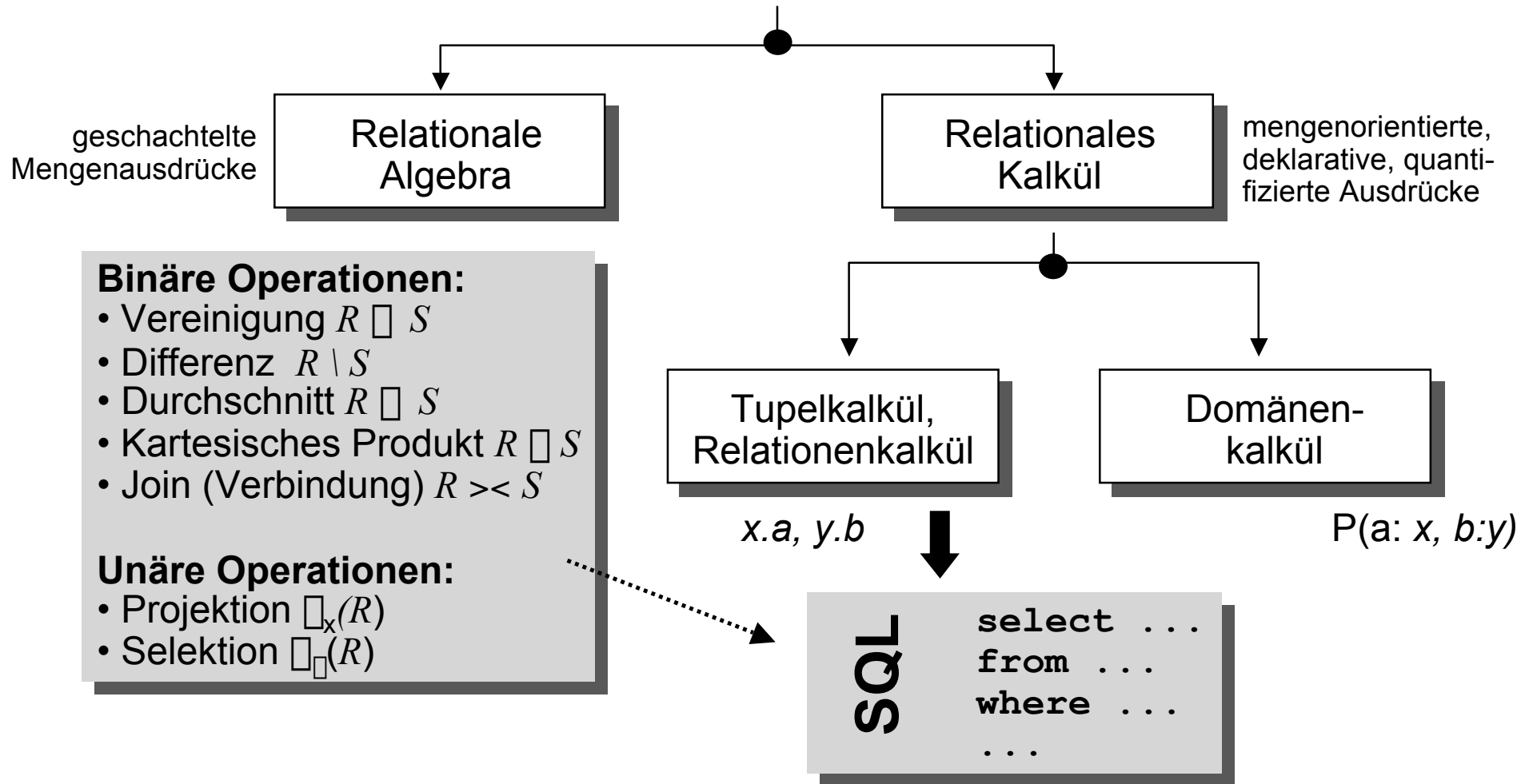


# Kapitel 3: Das relationale DB-Modell & SQL

	Relationales Datenmodell (RDM)	Objekt-orientierte Datenmodelle (OODM)	Objekt-relationale Datenmodelle	Semi-strukturierte Datenmodelle
Überblick über die Konzepte	3.1	4.1	5.1	6.1 6.2
Darstellung von Assoziationen				
Datendefinition				
Anfragen				
Aktualisierungsoperationen				
Spezifika	3.2 SQL	4.2 ODMG	5.2, 5.3	

# RDM: Anfragen

## Relationale Anfragesprachen im Überblick:



# RDM: Anfragen im relationalen Kalkül (1)

---

## Grundlage: Logische Kalkülausdrücke (mit Booleschem Ergebnis)

- ❑ **Basis:** Jede Bedingung  $\phi$ , mit  $\phi \in \{ =, \neq, <, \leq, >, \geq \}$ , die zwei Tupelkomponenten verknüpft, ist eine Formel, z.B.:  $x.a = y.b$ ,  $x.a > y.b$ , ...
- ❑ **Klammerung und Negation:** Ist  $f$  eine Formel, so sind dies auch  $( f )$  und  $\neg( f )$ .
- ❑ **Boolesche Operationen:** Sind  $f$  und  $g$  Formeln, so auch  $f \wedge g$  und  $f \vee g$ .
- ❑ **Quantoren:** Ist  $f$  eine Formel mit  $x$  als freier (Tupel-)Variable, so sind  $\exists x( f )$  und  $\forall x( f )$  Formeln, z.B.  $\exists x(x.a = 3)$ .
- ❑ **Abschluß:** Genau die durch die vorigen Vorschriften erzeugbaren Ausdrücke sind Formeln.

**Relationale Kalkülausdrücke** (z.B. `select x.a, y.b ... where  $\phi(x, y)$` ) mit  $x, y$  als in  $\phi$  freie Tupelvariable liefern relationenwertige Ergebnisse, nämlich alle Tupel mit den Komponenten  $x.a, y.b, \dots$ , die für  $\phi$  den Wahrheitswert `true` liefern (`select` als relationaler Quantor). In relationalen Kalkülausdrücken sind alle Tupelvariablen gebunden.

# RDM: Anfragen im relationalen Kalkül (2)

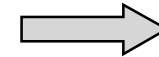
---

Im **Tupelkalkül** werden die Tupelvariablen  $x, y, \dots$  der logischen und relationalen Quantoren explizit an Relationen gebunden.

Beispiel:

```
x in Projekte;  
  
select x.Nr, x.Budget  
where x.Budget > 100000
```

<i>Nr</i>	<i>Titel</i>	<i>Budget</i>
100	DB Fahrpläne	300.000
200	ADAC Kundenstamm	100.000
300	Telekom Statistik	200.000



Ergebnisrelation

<i>Nr</i>	<i>Budget</i>
100	300.000
300	200.000

# RDM: Anfragen im relationalen Kalkül (3)

Im **Domänenkalkül** beziehen sich die verwendeten Variablen  $x, y, \dots$  nicht auf die existierenden Tupel einer Relation, sondern auf die durch den Wertebereich (»»»» *Domäne*) definierten möglichen Werte von Attributen.

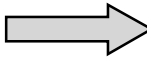
Beispiel:

```
x as int, y as float;
```

```
select x, y
```

```
where Projekte(Nr: x, Budget: y) □ (y > 250000)
```

Projekte	Nr	Titel	Budget
	100	DB Fahrpläne	300.000
	200	ADAC Kundenstamm	100.000
	300	Telekom Statistik	200.000



Nr	Budget
100	300.000

# RDM: Anfragen in SQL (1)

---

## Die Anfragesprache SQL:

Iterationsabstraktion mit Hilfe des **select from where**-Konstrukts:

- ❑ **select**-Klausel: Spezifikation der Projektionsliste für die Ergebnistabelle
- ❑ **from**-Klausel: Festlegung der angefragten Tabellen, Definition und Bindung der Tupelvariablen
- ❑ **where**-Klausel: Selektionsprädikat, mit dessen Hilfe die Ergebnistupel aus dem kartesischen Produkt der beteiligten Tabellen selektiert werden

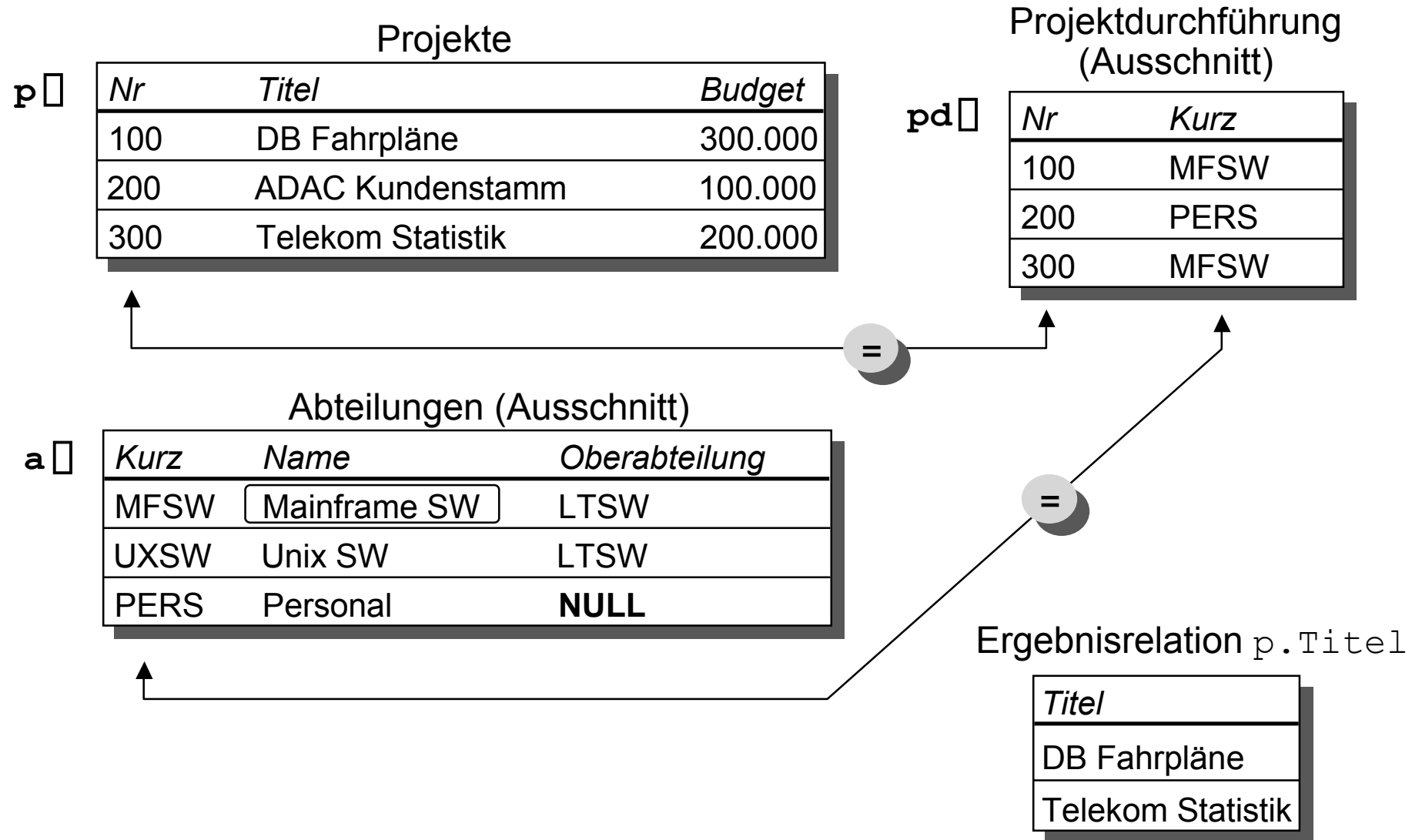
**Join:** Mehrere Tabellen werden wertbasiert, z.B. über gleiche Werte in zusammengehörigen Primärschlüssel/ Fremdschlüssel-Paaren, miteinander verknüpft.

Bestimmung der Projekttitel, an denen die Abteilung für *Mainframe Software* arbeitet:

```
select p.Titel
from Projekte p,
      Projektdurchfuehrung pd,
      Abteilungen a
where p.Nr = pd.Nr
and a.Kurz = pd.Kurz
and a.Name = 'Mainframe SW';
```

hier: *Join* über die Tabellen *Projekte*, *Abteilungen* und *Projektdurchführung* mit Selektion und Projektion

# RDM: Anfragen in SQL (2)



# RDM: Anfragen in SQL (3)

---

## **Bewertung:**

- ❑ Deklarative Formulierung der Anfrage, Auswertungsreihenfolge unspezifiziert
- ❑ Tupelkalkül: Die Variablen (hier:  $p$ ,  $pd$  und  $a$ ) sind explizit an die Tupel der Relationen gebunden (hier die Relationen Projekte, Abteilungen, Projektdurchführungen).
- ❑ Zusätzlich erlaubt SQL einige Algebra-Operationen (`union`).

# RDM: Aktualisierungsoperationen (1)

---

## Klassen von SQL-Modifikationsoperationen auf Datenbanken:

Operationen mit Relationen (Entitäts- oder Beziehungsrelationen) und Werten (für Attribute, Tupel, Teilrelationen) als Parameter.

Notation „wortreich“ (SQL ← SEQUEL = Structured English Query Language):

- ❑ Operationen zum Einfügen neuer Daten (`insert`-Befehl)
- ❑ Operationen zum Ändern von Daten (`update`-Befehl)
- ❑ Operationen zum Löschen von Daten (`delete`-Befehl)

# RDM: Aktualisierungsoperationen (2)

---

**Generische Operationen** (polymorph typisiert), Änderungsoperationen beziehen sich aus Relationen oder Teilrelationen (select ...):

## ❑ insert-Statement:

- Fügt ein einziges Tupel ein, dessen Attributwerte als Parameter übergeben werden.
- Fügt eine Ergebnistabelle ein.

```
insert into Projektdurchfuehrung
values (400, 'XYZA')
```

```
insert into Projektdurchfuehrung
(Nr, Kurz)
select p.Nr, a.Kurz
from Projekte p, Abteilungen a
where p.Titel = 'Telekom Statistik'
and a.Name = 'Unix SW'
```

## ❑ update-Statement:

- Selektion (des) der betreffenden Tupel(s)
- Neue Werte oder Formeln für zu ändernde Attribute

```
update Projekte
set Budget = Budget * 1.5
where Budget > 150000
```

# RDM: Aktualisierungsoperationen (3)

---

## ❑ delete-Statement:

- Selektion (des) der betreffenden Tupel(s)

```
delete  
from Projektdurchfuehrung  
where Kurz = 'MFSW';
```

# RDM: Bewertung (1)

---

## Positive Beiträge des relationalen Datenmodells:

- ❑ Es bietet einen Satz von im Modell implizit enthaltenen generischen Operatoren (modellspezifisch, nicht anwendungsabhängig).
- ❑ Algebraische Sichtweise: Selektion, Projektion, natürlicher Verbund, Mengenoperationen, Umbenennung
- ❑ Positive Eigenschaften der Operatoren:
  - deskriptive Anfragesprache
  - Abgeschlossenheit (⇒ Orthogonalität der Operatoren)
  - Optimierbarkeit
  - Sicherheit: Jeder syntaktisch korrekte Ausdruck liefert in endlicher Zeit ein endliches Ergebnis.

# RDM: Bewertung (2)

---

## Beschränkungen des relationalen Datenmodells:

### ❑ Beschränkungen der Datenstrukturen

- primitive vordefinierte Domänen (keine ADTs)
- "flache" Tupel
- atomare Attribute
- homogene, ungeordnete Mengen (keine Listen oder Gruppen)
- keine Unterstützung für Generalisierung und Spezialisierung

### ❑ Beschränkung der Identifikationsmechanismen

- benutzerdefinierte, wertbasierte Schlüsselwerte
- keine Garantie der referentiellen Integrität

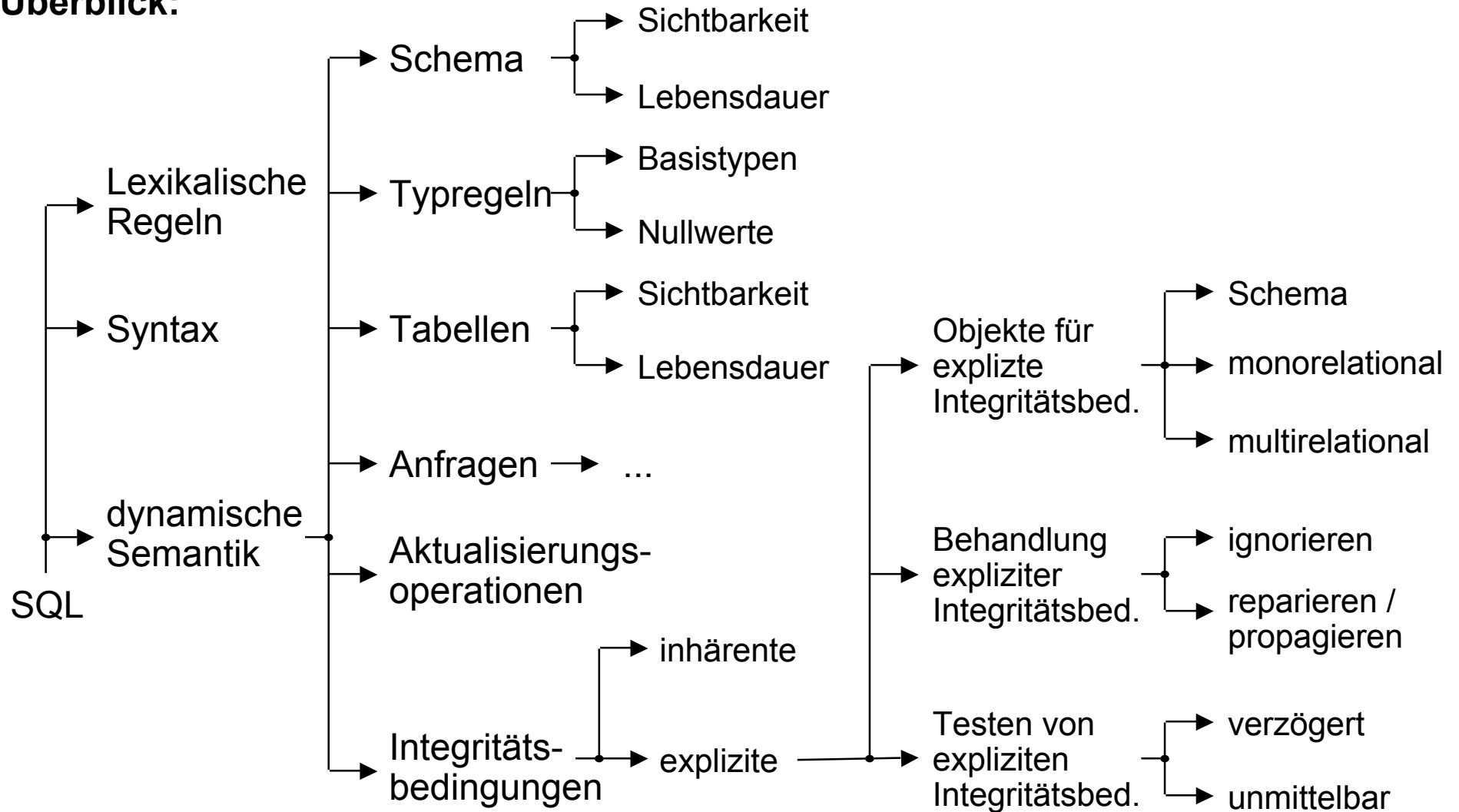
# RDM: Bewertung (3)

---

- ❑ Beschränkung der Anfragesprache und Datenmanipulationssprache
  - vordefinierte Operatoren für atomare Attribute
  - vordefinierte (generische) Operatoren für Mengen
  - keine algorithmische Vollständigkeit (*Turing completeness*)

# 3.2 SQL und relationale DB im Detail

## Überblick:



# SQL und relationale DB im Detail

---

- ❑ Lernziel: Vertiefung der Spezifika des RDM
  
- ❑ SQL ist weit verbreitet und genießt in der Praxis eine hohe Akzeptanz.
- ❑ SQL wurde ursprünglich für relationale Systeme entwickelt, fand aber auch Eingang in andere Systeme.
- ❑ Es gibt aber zahlreiche herstellereigene SQL-Dialekte.
- ❑ Nachfolgend wird der SQL-92-, bzw. der Kern des SQL-99-Standard zugrunde gelegt.
- ❑ Die Darstellung orientiert sich an der Beschreibung "normaler" Programmiersprachen.
  
- ❑ Literatur:
  - SQL Standard (s. nächste Folie)

# SQL-Standards

---

## **SQL-86:**

- ANSI X3.135-1986 Database Language SQL, 1986
- ISO/IEC 9075:1986 Database Language SQL, 1986

## **SQL-89:**

- ANSI X3.135-1989 Database Language SQL, 1989
- ISO/IEC 9075:1989 Database Language SQL, 1989

## **SQL-92:**

- ANSI X3.135-1992 Database Language SQL, 1992
- ISO/IEC 9075:1992 Database Language SQL, 1992
- DIN 66315 Informationstechnik - Datenbanksprache SQL, Aug. 1993

## **SQL-99:**

- ANSI/ISO/IEC Mehrteiliger Entwurf: Database Language SQL
- ANSI/ISO/IEC      9075:1999: Verabschiedung der Teile 1 bis 5  
                         9075:2000: Teil 10                      9075:2001: Teil 9

# Lexikalische und syntaktische Regeln (1)

---

SQL besitzt eine sehr umfangreiche Syntax, die sich durch eine hohe Anzahl optionaler Klauseln und schlüsselwortbasierter Operatoren auszeichnet.

Ein SQL-Quelltext wird von der Syntaxanalyse in eine Folge von Symbolen (□ *Lexeme, Token*) zerlegt.

- Nicht-druckbare Steuerzeichen (z.B. Zeilenvorschub) und Kommentare werden wie Leerzeichen behandelt.
- Kommentare beginnen mit "--" und reichen bis zum Zeilenende.
- Kleinbuchstaben werden in Großbuchstaben umgewandelt, falls sie nicht in Zeichenketten-Konstanten auftreten.

Aufgrund der zahlreichen *Modalitäten*, in denen SQL eingesetzt wird, kann es im Einzelfall weitere lexikalische Regeln geben.

# Lexikalische und syntaktische Regeln (2)

---

Es gibt die folgenden SQL-Symbole:

- ❑ **Reguläre Namen** beginnen mit einem Buchstaben gefolgt von evtl. weiteren Buchstaben, Ziffern und "\_".

```
Peter, mary33
```

- ❑ **Schlüsselworte:** SQL definiert über 210 Namen als Schlüsselworte, die nicht kontextsensitiv sind.

```
create, select
```

- ❑ **Begrenzte Namen** sind Zeichenketten in doppelten Anführungszeichen. Durch begrenzte Namen kann verhindert werden, daß neu hinzugekommene Schlüsselworte mit gewählten Bezeichnern kollidieren. (❑ *Syntaxerweiterungsproblematik*)

```
"intersect", "create"
```

- ❑ **Literale** dienen zur Benennung von Werten der SQL-Basistypen

```
'abc'      character(3)  
123       smallint  
B'101010' bit(6)
```

- ❑ weitere Symbole (Operatoren etc.)

```
<, >, =, %, &, (, ),  
*, +, ...
```

# Dynamische DDL Anweisungen (1)

---

In SQL findet keine strikte Trennung in *data definition language* zur Schemadefinition und eine separate *data manipulation language* zur Datenmanipulation statt.

Alle Anweisungen werden in SQL dynamisch zur Laufzeit ausgewertet.

## Vorteile:

- ❑ Operationen und Deklarationen können innerhalb von Transaktionen zu einer atomaren Operation zusammengefaßt werden.
- ❑ Deklarationen können zur Laufzeit durch Werte einer Gastsprache parametrisiert werden (Tabellen- und Spaltennamen, Typen, Integritätsbeziehungen ...).
- ❑ Deklarationen können innerhalb zusammengesetzter Anweisungen und Schleifen einer Gastsprache auftreten.

## Nachteile:

- ❑ **Dynamische Bindung:** Alle globalen Namen müssen zur Laufzeit dynamisch (in Schemata und Katalogen) identifiziert werden.

```
select * from Mitarbeiter
```

Tipfehler wird erst zur Laufzeit erkannt

# Dynamische DDL Anweisungen (2)

---

## Nachteile:

❑ **Dynamische Typisierung:** Die Kompatibilität zwischen Anfragen und der Struktur der von ihnen benutzten globalen SQL-Objekte kann erst zur Laufzeit getestet werden.

```
select * from Mitarbeiter
where Gehalt > Betrag
```

- Hat `Mitarbeiter` die Spalten `Gehalt` und `Betrag` ?
- Besitzen `Gehalt` und `Betrag` einen kompatiblen Basistyp ?
- Welche Vergleichsoperation muß verwendet werden ?
- Welche Spalten besitzt das Ergebnis ?

Moderne Datenbankmodelle lösen dieses Dilemma zwischen Flexibilität und statischer Konsistenzkontrolle beim Datenbankzugriff, indem sie eine Trennung zwischen der statischen Definition von Schemainformationen und der dynamischen Instantiierung von Datenobjekten einführen.

# Schemata und Kataloge (1)

---

- ❑ Ein SQL-Schema ist ein *dynamischer Sichtbarkeitsbereich* für die Namen geschachtelter (lokaler) SQL-Objekte (Tabellen, Sichten, Regeln ...)
- ❑ Bindungen von Namen an Objekten können durch Anweisungen explizit erzeugt und gelöscht werden (❑ blockstrukturierte Programmiersprachen).

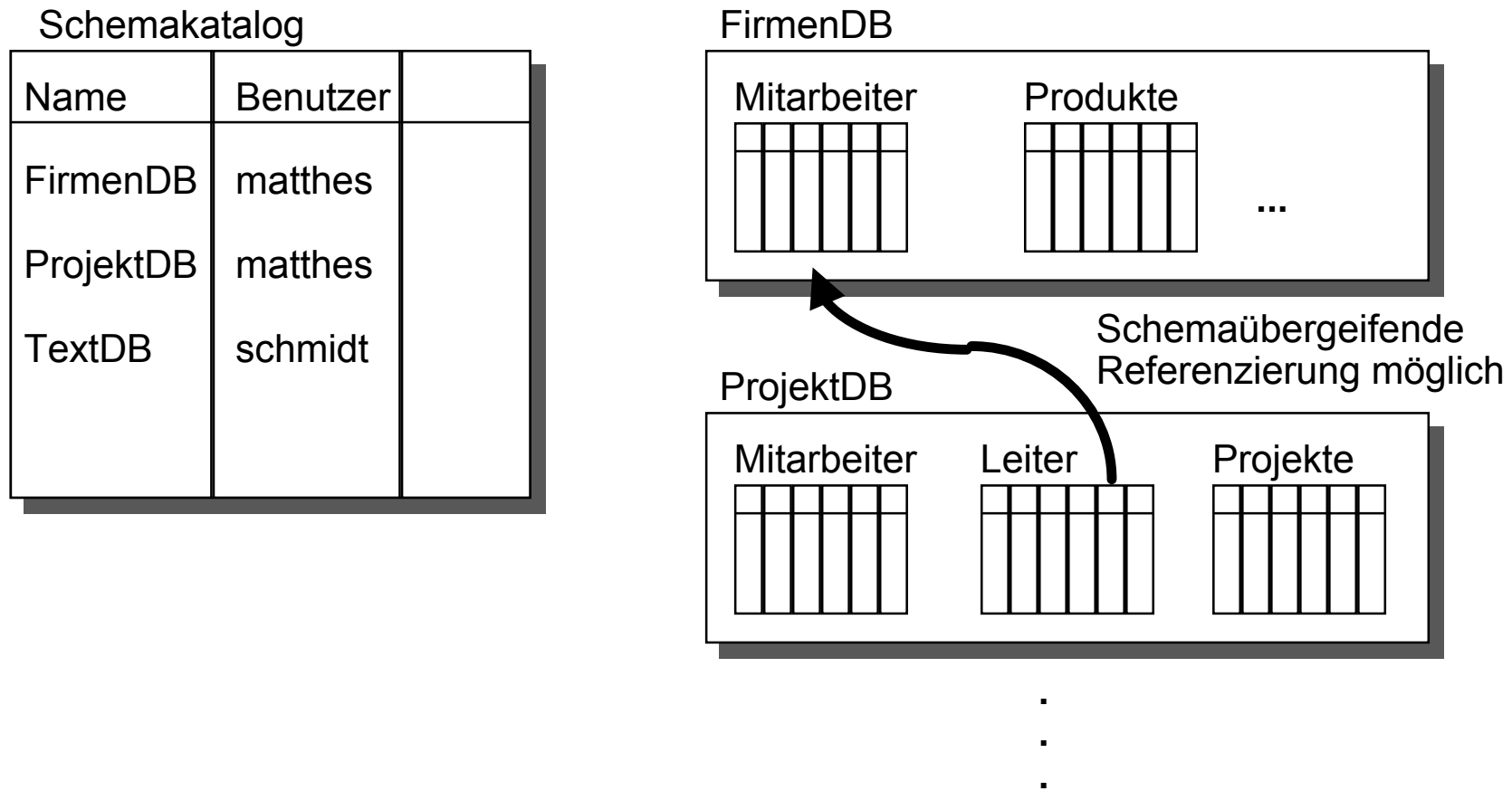
```
create schema FirmenDB;
  create table Mitarbeiter ...;
  create table Produkte ... ;

create schema ProjektDB;
  create table Mitarbeiter ...;
  create view Leiter ...;
  create table Projekte ...;
  create table Test ...;
  drop table Test;

drop schema FirmenDB;
```

- ❑ Die Integration separat entwickelter Datenbankschemata und die Arbeit in verteilten und föderativen Datenbanken erfordert den simultanen Zugriff auf SQL-Objekte mehrerer Schemata.

# Schemata und Kataloge (2)



# Schemata und Kataloge (3)

---

❑ Schemanamen können zur eindeutigen Benennung dienen.

```
FirmenDB.Mitarbeiter  
ProjektDB.Mitarbeiter
```

❑ Schemata werden

- zur Übersetzungszeit von SQL-Modulen oder
- dynamisch als Seiteneffekt von Anweisungen

```
create schema FirmenDB  
connect FirmenDB
```

definiert und kommen damit der Forderung nach dynamischer Bindung zwischen informationsverarbeitenden Algorithmen (Reaktivität) und persistenten Informationsbeständen nach.

❑ Ein SQL-Schema ist persistent.

❑ Anlegen und Löschen eines SQL-Schemas impliziert Anlegen bzw. Löschen der Datenbank, die das Schema implementiert.

❑ Die Lebensdauer geschachtelter SQL-Objekte ist durch die Lebensdauer ihrer Schemata begrenzt.

```
drop schema FirmenDB
```

# Schemata und Kataloge (4)

---

- ❑ *Schemaabhängigkeiten* entstehen durch Referenzen von SQL-Objekten eines Schemas in ein anderes Schema.  
(s. Abbildung auf Folie 3.2.23)

```
create view ProjektDB.Leiter as
select * from FirmenDB.Mitarbeiter
where ...
```

- ❑ Schemaabhängigkeiten müssen beim Löschen eines Schemas berücksichtigt werden. **cascade** erzwingt das transitive Löschen der abhängigen SQL-Objekte (Leiter).

```
drop schema FirmenDB cascade
```

- ❑ Schemata sind wiederum in Sichtbarkeitsbereichen enthalten, den Katalogen.
- ❑ Kataloge enthalten weitere Information wie z.B. Zugriffsrechte, Speichermedium, Datum des letzten Backup, ...

# Schemata und Kataloge (5)

---

- ❑ Die Namen von Katalogen sind in Katalogen abgelegt, von denen einer als *Wurzelkatalog* ausgezeichnet ist.
- ❑ Ein SQL-Objekt kann somit über eine mehrstufige Qualifizierung von Katalognamen, einen Schemanamen und einen Objektnamen ausgehend vom Wurzelkatalog eindeutig identifiziert werden.
- ❑ Katalogverwaltung wird teilweise an standardisierte Netzwerkdatendienste delegiert.

# Basisdatentypen und Typkompatibilität (1)

---

- ❑ Die formale Definition des relationalen Datenmodells basiert auf einer Menge von Domänen, der die atomaren Werte der Attribute entstammen.
- ❑ Anforderungen an die algebraische Struktur einer Domäne  $D$ :
  - Existenz einer Äquivalenzrelation auf  $D$  zur Definition der Relationensemantik (❑ *Duplikatelimination*) und des Begriffs der funktionalen Abhängigkeit.
  - Existenz weiterer Boolescher Prädikate ( $>$ ,  $<$ ,  $>=$ , **substring**, **odd**, ...) auf  $D$  zur Formulierung von Selektions- und Joinausdrücken über Attribute (optional).
- ❑ Moderne erweiterbare Datenbankmodelle unterstützen auch benutzerdefinierte Domänen.

# Basisdatentypen und Typkompatibilität (2)

---

SQL hält den Datenbankzustand und die Semantik von Anfragen unabhängig von speziellen Programmen und Hardwareumgebungen. Es definiert daher ein festes Repertoire an anwendungsorientierten *vordefinierten Basisdatentypen*, deren Definition folgendes umfaßt:

- ❑ **Lexikalische Regeln** für Literale
- ❑ **Evaluationsregeln** für unäre, binäre und n-äre Operatoren (Wertebereich, Ausnahmebehandlung, Behandlung von Nullwerten)
- ❑ **Typkompatibilitätsregeln** für gemischte Ausdrücke
- ❑ **Wertkonvertierungsregeln** für den bidirektionalen Datenaustausch mit typisierten Programmiersprachenvariablen bei der Gastspracheneinbettung.
- ❑ Spezifikation des **Speicherbedarfs** (minimal, maximal) für Werte eines Typs.

SQL bietet zahlreiche standardisierte Operatoren auf Basisdatentypen und erhöht damit die Portabilität der Programme.

# Basisdatentypen und Typkompatibilität (3)

---

Die SQL-Basisdatentypen lassen sich folgendermaßen klassifizieren:

- ❑ **Exact numerics** bieten exakte Arithmetik und gestatten teilweise die Angabe einer Gesamtlänge und der Nachkommastellenzahl.
- ❑ **Approximate numerics** bieten aufgrund ihrer Fließkommadarstellung einen flexiblen Wertebereich, sind jedoch wegen der Rundungsproblematik nicht für kaufmännische Anwendungen geeignet.
- ❑ **Character strings** beschreiben mit Leerzeichen aufgefüllte Zeichenketten fester Länge oder variabel lange Zeichenketten mit fester Maximallänge.
- ❑ **Bit strings** beschreiben mit Null aufgefüllte Bitmuster fester Länge oder variabel lange Bitfelder mit fester Maximallänge.

```
integer, smallint,  
numeric(p, s),  
decimal(p, s)
```

```
real,  
double precision,  
float(p)
```

```
character(n),  
character varying(n)
```

```
bit(n),  
bit varying(n)
```

# Basisdatentypen und Typkompatibilität (4)

---

❑ **Datetime** Basistypen beschreiben  
Zeit(punkt)werte vorgegebener Granularität.

```
date, time(p), timestamp,  
time(p) with time zone,
```

❑ **Time intervals** beschreiben Zeitintervalle  
vorgegebener Dimension und Granularität.

```
interval year(2) to month
```

SQL unterstützt sowohl die implizite Typanpassung (*coercion*), als auch die explizite Typanpassung (*casting*).

Durch die große Zahl von Basisdatentypen und die damit verbundenen Evaluations- und Typkompatibilitätsregeln besitzt ein Datenbanksystem eine hohe algorithmische Komplexität.

# Nullwerte und Wahrheitswerte (1)

---

Bei der Datenmodellierung und -programmierung können Situationen auftreten, in denen anstelle eines Wertes eines Basisdatentyps ein ausgezeichneter *Nullwert* benötigt wird. Z.B.:

- ❑ Ein Tabellenschema definiert, daß in jeder Reihe der Tabelle *Mitarbeiter* die Spalte *Alter* einen Wert des Typs `integer` besitzt. Ist das Alter *unbekannt*, so kann dies mit dem Wert `null` gekennzeichnet werden.
- ❑ Ein Tabellenschema definiert, daß in jeder Reihe der Tabelle *Abteilungen* die Spalte *Oberabt* einen Wert des Typs `string` besitzt. Ist *bekannt*, daß eine Abteilung *keine* Oberabteilung besitzt, so kann diese Information mit dem Wert `null` repräsentiert werden.

Jeder SQL-Basisdatentyp ist zur Unterstützung solcher Modellierungssituationen um den ausgezeichneten Wert `null` erweitert, der von jedem anderen Wert dieses Typs verschieden ist.

Das Auftreten von Nullwerten in Attributen oder Variablen kann verboten werden.

```
integer not null
```

# Nullwerte und Wahrheitswerte (2)

---

## Vorteile:

- ❑ Explizite und konsistente Behandlung von Nullwerten durch alle Applikationen (im Gegensatz zu ad hoc Lösungen, bei denen z.B. der Wert -1, *-MaxInt* oder die leere Zeichenkette als Nullwert eingesetzt wird)
- ❑ Exakte Definition der Semantik von Datenbankoperatoren (Zuweisung, Vergleich, Arithmetik) auf Nullwerten.

## Nachteile:

- ❑ Eine Erweiterung eines Datentyps um Nullwerte steht oft im *Konflikt* mit den algebraischen Eigenschaften (Existenz von Nullelementen, Assoziativität, Kommutativität, Ordnung, ...) des nicht-erweiterten Datentyps.  
(...  $-2 < -1 < 0 < \text{null} < 1 < 2 < \dots$  ?)
- ❑ Algebraische Eigenschaften werden häufig zur *Anfrageoptimierung* ausgenutzt. Eine Anfrage, die Werte eines Datentyps  $T'$  (wobei  $\text{null} \in T'$ ) verwendet, bietet im Regelfall weniger Optimierungsspielraum als eine Anfrage mit Werten des Datentyps  $T$  (mit  $\text{null} \notin T$ ) .

# Nullwerte und Wahrheitswerte (3)

---

## Nachteile (Fortsetzung):

- ❑ Die Einführung von Nullwerten für Variablen führt zu einem *Schneeballeffekt*, indem Nullwerte als (Zwischen-)ergebnisse von beliebigen Ausdrücken auftreten können und separat behandelt werden müssen.
- ❑ Beim Datenaustausch zwischen SQL-Datenbanken und Programmiersprachen werden SQL-Attribute eines Typs  $\mathbb{T}$ , der auch Nullwerte annehmen kann, durch ein Paar bestehend aus einer Programmiersprachenvariablen des Typs  $\mathbb{T}$  und einer (Booleschen) *Indikatorvariable* dargestellt, die angibt, ob der SQL-Wert `null` ist.
  - ❑ Höhere Komplexität der Anwendungsprogramme

Bei Datentypen für Wahrheitswerte führen Nullwerte zu einer *dreiwertigen Logik* (`true`, `false`, `null`), in der die Restriktion aller Operatoren (`and`, `or`, `not`) auf die Argumente *true* und *false* die übliche Boolesche Semantik besitzt.

# Nullwerte und Wahrheitswerte (4)

---

Wahrheitstabellen der dreiwertigen SQL-Logik:

OR	true	false	null
true	<i>true</i>	<i>true</i>	<i>true</i>
false	<i>true</i>	<i>false</i>	<i>null</i>
null	<i>true</i>	<i>null</i>	<i>null</i>

AND	true	false	null
true	<i>true</i>	<i>false</i>	<i>null</i>
false	<i>false</i>	<i>false</i>	<i>false</i>
null	<i>null</i>	<i>false</i>	<i>null</i>

x	not x	x is null	x is not null
true	<i>false</i>	<i>false</i>	<i>true</i>
false	<i>true</i>	<i>false</i>	<i>true</i>
null	<i>null</i>	<i>true</i>	<i>false</i>

Schwierigkeiten bei der konsistenten Erweiterung einer Domäne um Nullwerte werden bereits am einfachen Beispiel der Booleschen Werte und der grundlegenden logischen Äquivalenz  $x \text{ and } \text{not } x = \text{false}$  deutlich, die bei der Erweiterung der Domäne um Nullwerte verletzt wird ( $\text{null and not null} = \text{null}$ )

# Nullwerte und Wahrheitswerte (5)

---

Nullwerte haben in der Theorie und Praxis eine hohe Bedeutung erlangt.

- ❑ Dies ist zum Teil auf die eingeschränkte Datenstrukturflexibilität des RDM zurückzuführen (homogene Mengen flacher Tupel).
- ❑ Modellierungsaufgaben, die im RDM Nullwerte erfordern, können in anderen Modellen durch speziellere Konzepte gelöst werden (Vereinigungstypen, Subtypisierung, Vererbungsbeziehungen).

In der Forschung wurden alternative Modelle für Nullwerte und mehrwertige Logiken vorgeschlagen (z.B. eine Studie der *DataBase Systems Study Group* mit 29 alternativen Bedeutungen für Nullwerte).

Bei der Entwicklung zukünftiger SQL-Standards sind zusätzliche Nullwerte und benutzerdefinierte Nullwerte in der Diskussion.

# Tabellendefinition (1)

---

Eine Tabellendefinition umfaßt die folgenden Teilschritte in einem syntaktischen Konstrukt:

- ❑ **Typdefinition:** Definition einer Tabellenstruktur (Spaltennamen, Spaltentypen)
- ❑ **Variablendefinition:** Definition eines Namens für eine Tabelle dieser Struktur im aktuellen Schema.
- ❑ **Variableninstantiierung:** Dynamisches Anlegen einer Variablen dieser Struktur in der aktuellen Datenbank. Eine Variable wird in SQL mit der leeren Tabelle instantiiert.

```
create table Mitarbeiter (  
    Name    char(29),  
    Gehalt  integer,  
    Urlaub  smallint);
```

Die statisch fixierte Anzahl der Spalten wird als *Grad* der Tabelle bezeichnet.

- ❑ Die Reihenfolge der Spalten ist signifikant.
- ❑ Tabellen mit dem Grad 0 sind nicht erlaubt.

# Tabellendefinition (2)

---

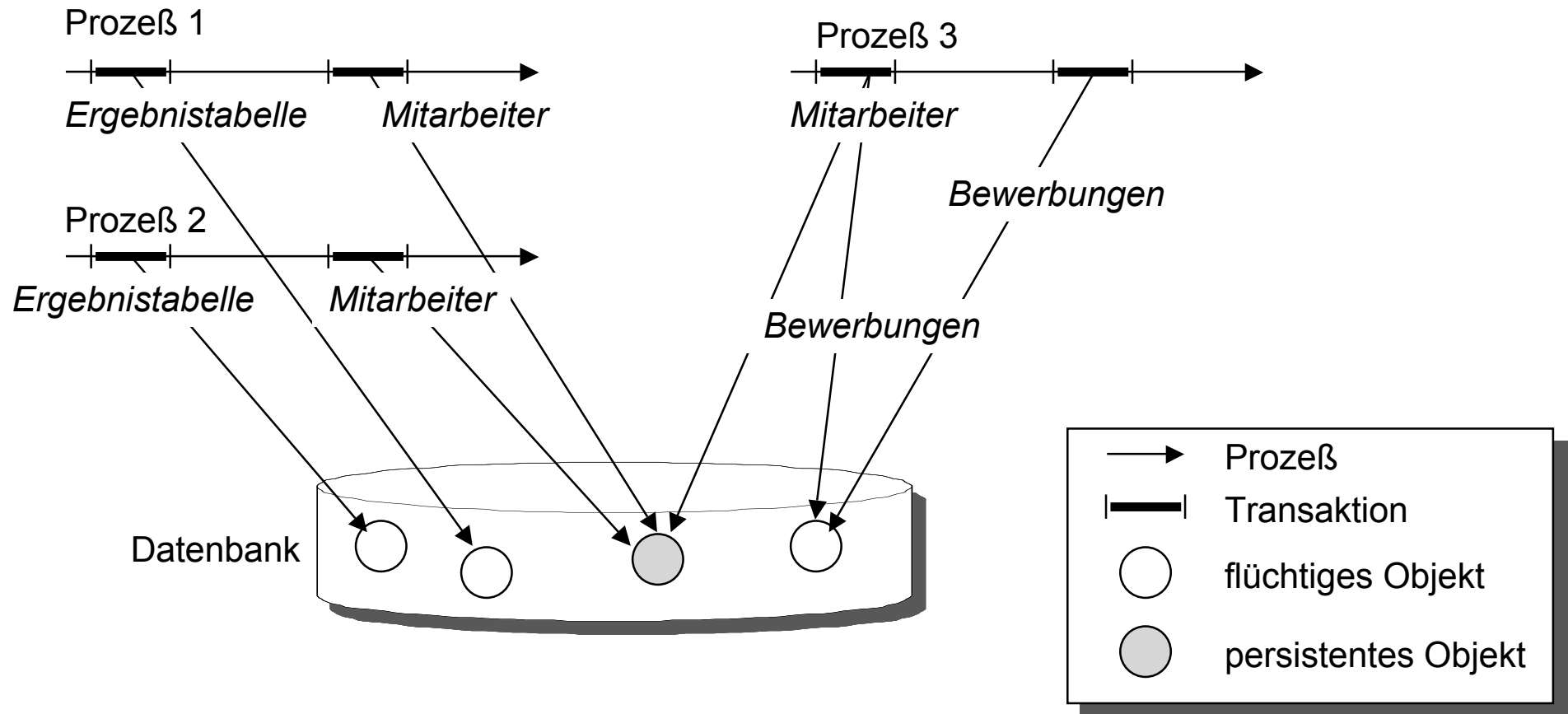
- ❑ Die dynamisch variierende Zahl der Reihen bezeichnet man mit *Kardinalität*.
- ❑ Im Gegensatz zum RDM sind in Tabellen Duplikate erlaubt.
- ❑ Eine Reihe ist ein Duplikat einer anderen Reihe, wenn beide in allen Spalten gemäß der Äquivalenzrelation der jeweiligen Spaltentypen übereinstimmen.
- ❑ Tabellen mit Kardinalität 0 heißen leer.
- ❑ Die Reihenfolge der Reihen ist unspezifiziert.

Tabellendefinitionen können dynamisch modifiziert werden:

```
alter table Mitarbeiter
  add column Adresse varchar(40)
  alter column Name unique
  drop column Urlaub;
drop table Mitarbeiter;
```

# Lebensdauer, Sichtbarkeit, gemeinsame Nutzung (1)

Die gleiche Datenbank kann von verschiedenen informationsverarbeitenden Prozessen simultan oder sequentiell nacheinander benutzt werden.



Transaktionen schützen simultanen Zugriff (s. Kapitel 8)

# Lebensdauer, Sichtbarkeit, gemeinsame Nutzung (2)

---

Bei der Deklaration von Datenbankobjekten wie SQL-Tabellen sind drei Objekteigenschaften zu definieren:

- ❑ **Lebensdauer** (*extent*): Der Zustand eines Objektes kann *flüchtig* oder *persistent* gespeichert werden.
- ❑ **Sichtbarkeit** (*scope*): Der Name eines Objektes kann *global* für alle Prozesse, die eine Datenbank benutzen oder nur *lokal* für einen Prozeß sichtbar sein. Der Sichtbarkeitsbereich eines Prozesses kann durch SQL-Module noch weiter partitioniert werden.
- ❑ **Gemeinsame Nutzung** (*sharing*): Ein Name kann entweder eine *Referenz* auf ein für mehrere Prozesse zugreifbares Objekt oder eine *prozeßlokale Kopie* eines Objektes bezeichnen. Referenzen und Kopien unterscheiden sich in der Wirkung von Seiteneffekten.

**Beachte:** Diese Objekteigenschaften sind nicht vollständig orthogonal.  
**Und:** Namen mit globaler Sichtbarkeit können Objekte mit flüchtiger Lebensdauer bezeichnen (❑ *dangling reference*).

# Lebensdauer, Sichtbarkeit, gemeinsame Nutzung <sup>(3)</sup>

---

Historisch gesehen haben sich Datenbanksysteme auf *persistente globale* Datenobjekte konzentriert.

Vorteile durch Unterstützung flüchtiger lokaler Objekte:

- ❑ Vermeidung von *Namenskonflikten* im globalen Sichtbarkeitsbereich der Datenbank.
- ❑ Automatische *Speicherfreigabe* durch das Datenbanksystem am Prozedur-, Transaktions- bzw. Prozeßende.
- ❑ Effizienzgewinn durch die Möglichkeit zur *prozeßlokalen Speicherung* (z.B. im Hauptspeicher)
- ❑ Effizienzgewinn durch die *Vermeidung von Synchronisationsoperationen* (Sperrern, Nachrichten, ...) zwischen Prozessen.

Für *temporäre Tabellen* läßt sich die Lebensdauer der enthaltenen Datensätze auf einen gesamten Prozeß oder nur eine Transaktion einschränken. Dabei müssen keine aufwendigen Fehlererholungsinformationen zum Rücksetzen des Tabellenzustandes im Falle eines Fehlers gespeichert werden.

```
on commit preserve rows
on commit delete rows
```

# Standardwerte für Spalten

---

Beim Einfügen von Reihen in eine Tabelle können einzelne Spalten unspezifiziert bleiben.

```
insert into Mitarbeiter
(Name, Gehalt, Urlaub)
values ("Peter", 3000, null)
```

```
insert into Mitarbeiter
(Name, Gehalt)
values ("Peter", 3000)
```

Die fehlenden Werte werden mit `null` oder mit bei der Tabellenerzeugung angegebenen Standardwerten belegt.

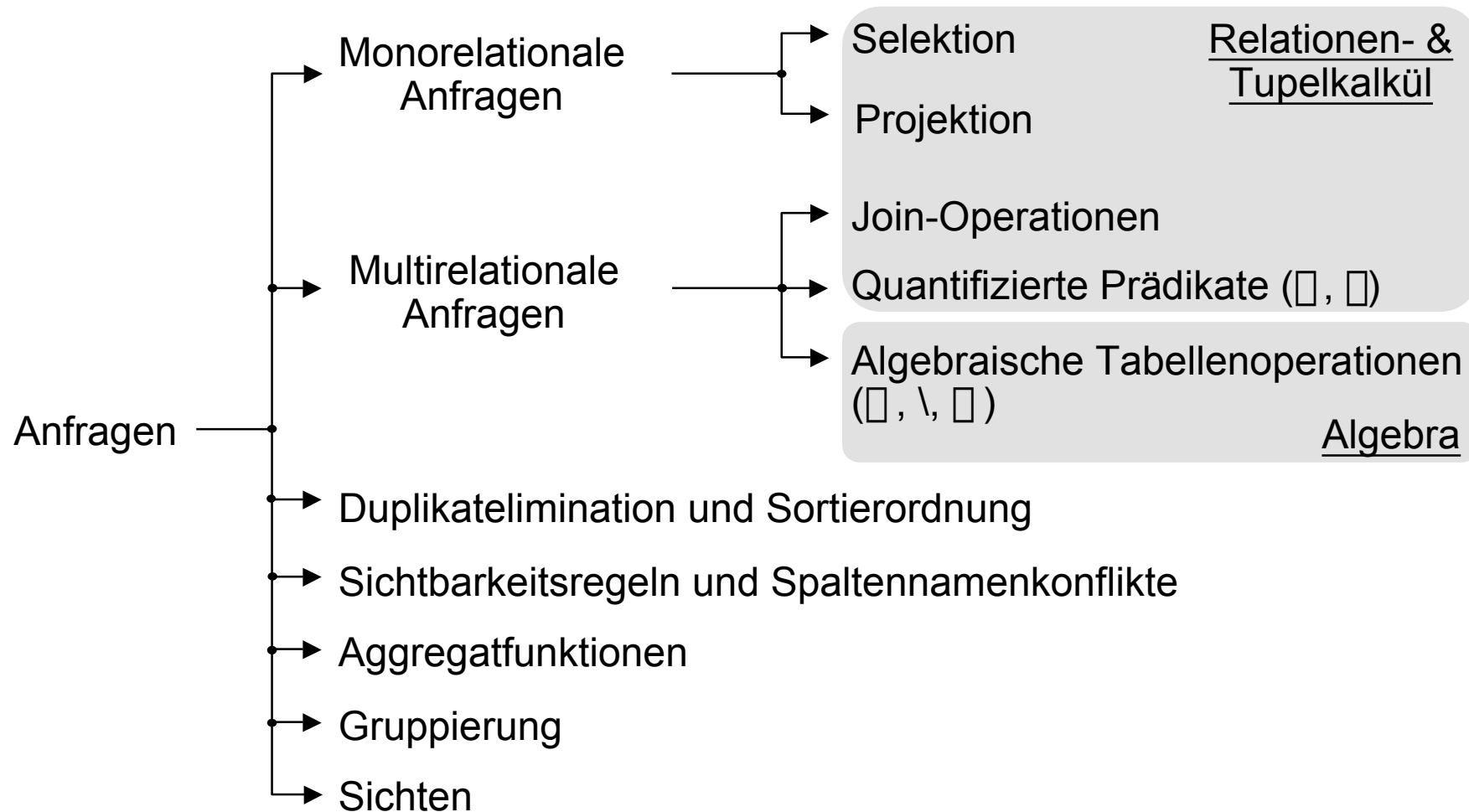
- ❑ Standardwerte können Literale eines Basisdatentyps sein.
- ❑ Standardwerte können eine parameterlose SQL-Funktion sein, die zum Einfügezeitpunkt ausgewertet wird.

Standardwerte leisten einen nicht zu unterschätzenden Beitrag zur *Datenunabhängigkeit* und *Schemaevolution*:

- ❑ Existierende Anwendungsprogramme können auch nach dem Erweitern einer Relation konsistent mit neu erstellten Anwendungen interagieren.

# Anfragen: Überblick

---



# Grundlegendes SQL-Sprachkonstrukt

---

□ *Mengenorientierte select from where-Anfrage*, die aus

- einer *Projektionsliste*,
- einer Liste von *Bereichstabellen*
- und einem *Selektionsprädikat*

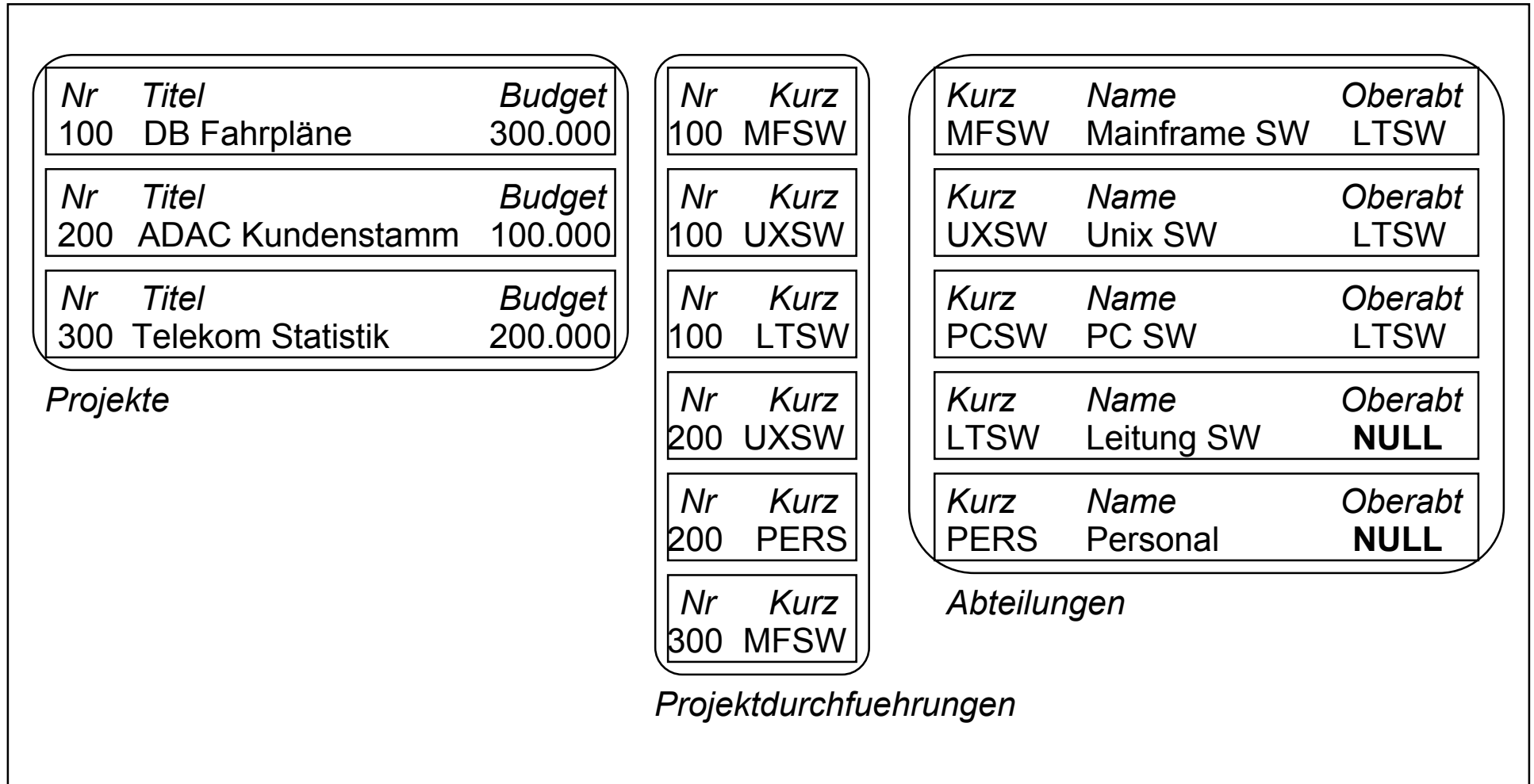
besteht.

□ Anfrageergebnis: Tabelle

□ Iterationsabstraktion (deklarativ)

```
select Projektionsliste  
from Bereichstabelle(n)  
where Selektionsprädikat;
```

# Zur Erinnerung: Projektdatenbank



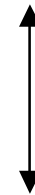
Projektdatenbank

# Monorelationale Anfragen (1)

---

- ❑ Anfrage mit Bezug auf *eine Bereichstabelle*
- ❑ Ergebnis: Flüchtige, anonyme Tabelle, deren Spaltenstruktur durch die *Projektionsliste* bestimmt wird.
- ❑ Die *Projektionsliste* besteht aus einer durch Kommata getrennten Liste von Ausdrücken, die Werte der *SQL-Basisdatentypen* liefern müssen.
- ❑ Das *Selektionsprädikat* ist ein beliebiger Boolescher Ausdruck, der zu *true*, *false* oder *null* evaluieren kann.
- ❑ Für jede Zeile der *Bereichstabelle*, die das *Selektionsprädikat* erfüllt, wird die *Projektionsliste* ausgewertet und eine neue Zeile mit den berechneten Spaltenwerten in die Ergebnistabelle eingefügt.
- ❑ undefinierte Reihenfolge der Zeilen in der Ergebnistabelle

```
select Projektionsliste  
from Bereichstabelle  
where Selektionsprädikat;
```



```
select Name, Kurz  
from Abteilungen  
where Oberabt  
       = 'LTSW';
```

# Weiterverwendung von Anfrageergebnissen

---

- ❑ Sicherung des Anfrageergebnisses in einer separaten, persistenten Tabelle, Beispiel:

```
create table SWUnterabteilungen as  
select Name, Kurz from Abteilungen where Oberabt = 'LTSW';
```

- Schnappschuss der Daten zum Zeitpunkt der Anfrage.
- Kann unabhängig von Änderungen der Ausgangsdaten weiterverwendet werden.

- ❑ Sicherung in einer temporären Tabelle, Beispiel:

```
create temporary table SWUnterabteilungen as ...
```

- Tabellendaten sind nur während derselben Transaktion oder Datenbankverbindung gültig.
- Beim Transaktions- oder Verbindungsende werden Daten der temporären Tabelle automatisch gelöscht.

# Weiterverwendung von Anfragen

---

- Definition einer Sicht (View), Beispiel:

**create view** SWUnterabteilungen **as**

**select** Name, Kurz **from** Abteilungen **where** Oberabt = 'LTSW';

- Nicht das Ergebnis, sondern die Anfrage wird benannt.
- Bei jeder Verwendung wird die Basisanfrage über dem aktuellen Datenbestand ausgewertet, Beispiel:

**select** u.name, p.nr

**from** SWUnterabteilungen u, Projektdurchfuehrungen p

**where** u.kurz = p.kurz

Die Sicht SWUnterabteilungen wird wie eine gewöhnliche Basistabelle verwendet.

- Direkte Verwendung eines Anfrageergebnisses als Bereichsrelation einer komplexen Anfrage.

**select** u.Name, p.Nr

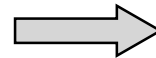
**from** (select Name, Kurz **from** Abteilungen **where** Oberabt = 'LTSW') u,  
Projektdurchfuehrungen p

**where** u.Kurz = p.Kurz

# Monorelationale Anfragen (2)

- Beispiel: SQL-Anfrage zur Bestimmung der Namen und des Kürzels aller Abteilungen, die der Abteilung "Leitung Software" mit dem Kürzel *LTSW* untergeordnet sind

```
select Name, Kurz
from Abteilungen
where Oberabt = 'LTSW';
```



Ergebnistabelle

<i>Name</i>	<i>Kurz</i>
Mainframe SW	MFSW
Unix SW	UXSW
PC SW	PCSW

**Selektion:** Aufzählung *aller* Spalten (durch \* in der *Projektionsliste*) der Bereichstabelle unter Beibehaltung der Spaltenreihenfolge

```
select *
from Abteilungen
where Oberabt
      = 'LTSW';
```



Ergebnistabelle

<i>Kurz</i>	<i>Name</i>	<i>Oberabt</i>
MFSW	Mainframe SW	LTSW
UXSW	Unix SW	LTSW
PCSW	PC SW	LTSW

# Monorelationale Anfragen (3)

---

**Projektion:** Entsteht durch Weglassen der where-Klausel (entspricht der Angabe des *Selektionsprädikats true*, so daß für jede Zeile der *Bereichstabelle* eine Zeile in der Ergebnistabelle existiert).

```
select Oberabt  
from Abteilungen;
```



Ergebnistabelle

<i>Oberabt</i>
LTSW
LTSW
LTSW
<b>NULL</b>
<b>NULL</b>

**Flüchtige Kopie einer Datenbanktabelle T:**

```
select * from T;
```

# Monorelationale Anfragen (4)

---

**Explizite Definition der Spaltennamen der Ergebnistabelle in der *Projektionsliste*:**

```
select Kurz as Unter,  
        Oberabt as Ober  
from Abteilungen;
```



Ergebnistabelle

<i>Unter</i>	<i>Ober</i>
MFSW	LTSW
UXSW	LTSW
PCSW	LTSW
LTSW	<b>NULL</b>
PERS	<b>NULL</b>

# Multirelationale Anfragen (1)

---

- ❑ Anfragen, bei denen Spalten und Zeilen mehrerer *Bereichsrelationen* ( $T_1, \dots, T_n$ ) miteinander verknüpft werden.
- ❑ Ziel: Formulierung von Anfragen über *Objektbeziehungen* im Relationalen Modell
- ❑ Typisierungs- und Auswertungsregeln siehe "Monorelationale Anfragen" mit dem Unterschied, daß das *Selektionsprädikat* und die *Projektionslisten* für alle möglichen Kombinationen der Zeilen der  $n$  *Bereichstabellen* ausgewertet werden.
- ❑ Konzeptionell findet also eine Selektion und Projektion über das *Kartesische Produkt* der angegebenen Bereichstabellen statt.
- ❑ Beispiel: **Equi-Join** mit zwei Tabellen (s. nächste Folie)

```
select Projektionsliste
from  $T_1, \dots, T_n$ 
where Selektionsprädikat;
```

**lokale Bereichsvariable**  
(s. Folie 3.2.42)

```
select p.*, pd.Nr as Nr2,
       pd.Kurz
from Projekte p,
       Projektdurchfuehrungen pd
where p.Nr = pd.Nr;
```

# Multirelationale Anfragen (2)

Beispiel: Projekte  $\bowtie_{(Nr = Nr)}$  Projektdurchfuehrungen

Projektdurchfuehrungen  
(Ausschnitt)

Projekte

<i>Nr</i>	<i>Titel</i>	<i>Budget</i>
100	DB Fahrpläne	300.000
200	ADAC Kundenstamm	100.000
300	Telekom Statistik	200.000

<i>Nr</i>	<i>Kurz</i>
100	MFSW
200	PERS
300	MFSW

Ergebnisrelation

<i>Nr</i>	<i>Titel</i>	<i>Budget</i>	<i>Nr2</i>	<i>Kurz</i>
100	DB Fahrpläne	300.000	100	MFSW
200	ADAC Kundenstamm	100.000	200	PERS
300	Telekom Statistik	200.000	300	MFSW

# Multirelationale Anfragen (3)

---

- ❑ Verwendet man einen Stern (\*) in der *Projektionsliste*, so besitzt die Ergebnistabelle alle Spalten der *Bereichstabellen* in der Reihenfolge, in der die *Bereichstabellen* in der from-Klausel aufgelistet wurden.
- ❑ Festlegung der *Join-Bedingung* in der where-Klausel der Anfrage
- ❑ **n-Weg-Join:** Anfragen über  $n \geq 2$  *Bereichstabellen*
- ❑ **Equi-Join:** Als *Selektionsprädikat* wird ein Gleichheitstest (=) zwischen Spaltenwerten benutzt.
- ❑ **Theta-Join:** Als *Selektionsprädikat* wird ein anderes Boolesches Prädikat anstatt des Gleichheitstests benutzt (<, >, ≥, ≤, ≠, like, ...).

# Sichtbarkeitsregeln und Spaltennamenkonflikte (1)

---

## Sichtbarkeitsregeln für *lokale* Namen (z.B. Spalten-, Bereichsvariablenamen) innerhalb kalkülorientierter SQL-Anfragen:

- ❑ In den Teilausdrücken  $P$ ,  $S$ ,  $T_1, \dots, T_n$  sind alle globalen Namen von SQL-Objekten (z.B. Tabellen, Sichten, Schemata, Kataloge) sichtbar.
- ❑ Im *Selektionsprädikat*  $S$  und in der *Projektionsliste*  $P$  sind zusätzlich die *lokalen* Namen aller Spalten aller *Bereichstabellen*  $T_i$  sichtbar.
- ❑ Analoge Sichtbarkeitsregeln für *any* und *some*-Klauseln
- ❑ Ein lokaler Name überdeckt dabei einen globalen Namen.

```
select P
from T1, ..., Tn
where S;
```

# Sichtbarkeitsregeln und Spaltennamenkonflikte (2)

## Definition *lokaler Bereichsvariablen* (correlation names, alias names):

- ❑ Zur Vermeidung von Namenskonflikten zwischen den Spaltennamen verschiedener Tabellen sowie zwischen Spaltennamen und globalen Namen
- ❑ Einsetzung der *lokalen Bereichsvariablen* zur Qualifizierung von Spaltennamen mittels Punktnotation im *Selektionsprädikat* und der *Projektionsliste*
- ❑ Ziel bei der Verwendung von *Bereichsvariablen* in SQL-Anfragen:
  - Lesbarkeit: Zu welcher *Bereichstabelle* gehört ein Spaltenname?
  - Ausdrucksmächtigkeit:  *reflexive Anfragen* (s. nächste Folie)

```
select P
from T1 X1, ...,
     Tn Xn
where S;
```



```
select m.*
from Mitarbeiter m,
     Projekte p
where m.Projekte =
     p.Nr;
```

# Sichtbarkeitsregeln und Spaltennamenkonflikte (3)

## Beispiel: Reflexive Anfrage

- ❑ Hier: Tabelle der Ober- und Unterabteilungen

- ❑ Verallgemeinerung:

- Rekursive Anfragen ❑ in SQL nicht möglich

```
select o.Name as Oberabteilung,  
       u.Name as Unterabteilung  
from Abteilungen o,  
     Abteilungen u  
where u.Oberabteilung = o.Kurz;
```

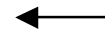


<i>Oberabteilung</i>	<i>Unterabteilung</i>
Leitung SW	Mainframe SW
Leitung SW	Unix SW
Leitung SW	PC SW

# Probleme rekursiver Anfragen

Beispiel: Bestimmung aller Oberabteilungen einer Abteilung

```
select    u.name as Unterabteilung,  
          o1.name as ersteOberabteilung,  
          o2.name as zweiteOberabteilung  
...  
from      abteilungen u,  
          abteilungen o1,  
          abteilungen o2  
...  
where     u.oberabt = o1.kurz  
and       o1.oberabt = o2.kurz  
...
```



Tabellenbreite wird in der Anfrage spezifiziert (select-Teil), müßte aber von den tatsächlichen Daten abhängen dürfen.

Für jede Hierarchieebene muß eine eigene Anfrage formuliert und mit **union** dem Gesamtergebnis hinzugefügt werden.

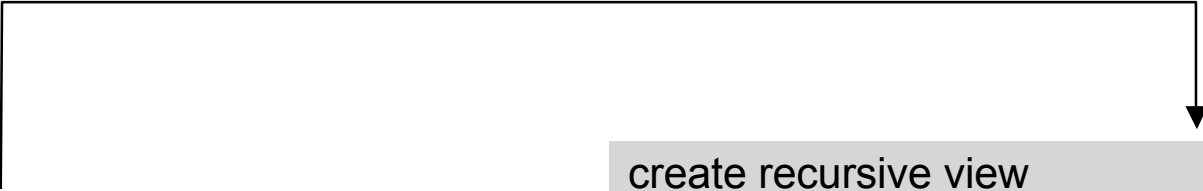


```
select    u.name as Unterabteilung,  
          o1.name as Oberabteilung  
from      abteilungen u,  
          abteilungen o1  
where     u.oberabt = o1.kurz  
  
union  
  
select    u.name as Unterabteilung,  
          o2.name as Oberabteilung  
from      abteilungen u,  
          abteilungen o1,  
          abteilungen o2  
where     u.oberabt = o1.kurz  
and       o1.oberabt = o2.kurz  
...
```

# Mögliche Struktur rekursiver Anfragen

---

Pseudo-SQL-Notation:  
Rekursion durch Benennung der  
Anfrage und Wiederverwendung  
des Namens innerhalb ihrer  
eigenen Definition



```
create recursive view
Unterabteilungen
select r.kurz, r.oberabt
from abteilungen r
union
select u.kurz, o.oberabt
from Abteilungen o,
     Unterabteilungen u
where o.kurz = u.oberabt
```

- Beim Start der Rekursion enthält „Unterabteilungen“ nur die Tupel der ersten Teilanfrage.
- Tupel, die sich durch den Join in der zweiten Teilanfrage ergeben, werden der Extension von „Unterabteilungen“ für die nächste Iteration hinzugefügt.
- Abbruch der Rekursion, sobald die zweite Teilanfrage bei Verwendung der Ergebnisse aus der vorigen Iteration keine zusätzlichen Ergebnistupel mehr liefert □ Fixpunkt.

# Quantifizierte Prädikate

---

## Universelle Quantifizierung:

- $\{x \in R \mid \forall y \in S : x \sqsupset y\}$
- Hier: Tabelle aller Projekte  $x$ , die ein höheres Budget als *alle* externen Projekte  $y$  haben.

```
select *
from Projekte x
where x.Budget > all
      (select y.Budget
       from ExterneProjekte y);
```

## Existentielle Quantifizierung:

- $\{x \in R \mid \exists y \in S : x \sqsupset y\}$
- Hier: Tabelle aller Projekte  $x$ , die mindestens an *einer* Projektdurchführung  $y$  beteiligt sind.

```
select *
from Projekte as x
where x.Nr = some
      (select y.Nr
       from Projektdurchfuehrungen y);
```

# Algebraische Tabellenoperationen (1)

---

## Vereinigung $R \sqcup S$ :

- ❑ Alle Tupel zweier Relationen werden in einer Ergebnisrelation zusammengefaßt.
- ❑ Das Ergebnis enthält keine Duplikate (❑ union-Befehl).

$$R \sqcup S := \{ r \mid r \in R \vee r \in S \}$$

- ❑ Möchte man eventuelle Duplikate nicht beseitigen, so ist der Befehl `union all` zu verwenden.
- ❑ Voraussetzung für Verknüpfung mit einer algebraischen Tabellenoperation:  
Kompatibilität der Spaltenstruktur der beteiligten Tabelle  
(❑ gleiche Spaltennamen und Datentypen)
- ❑ Beispiel:  $R \sqcup S$  (s. nächste Folie)
- ❑ Unterdrückung der Kompatibilitätsforderung durch `corresponding`-Klausel

# Algebraische Tabellenoperationen (2)

Beispiel für eine Vereinigung:

```
select *  
from R  
union  
select *  
from S;
```

Relation  $R$

<i>ANr</i>	<i>AName</i>	<i>Menge</i>
001	Anlasser	1.000
⋮	⋮	⋮
199	Kolben	5.000

Relation  $S$

<i>ANr</i>	<i>AName</i>	<i>Menge</i>
237	Ölfiler	1.560
⋮	⋮	⋮
851	Schraube	25.000

Ergebnisrelation  $R \sqcup S$

<i>ANr</i>	<i>AName</i>	<i>Menge</i>
001	Anlasser	1.000
⋮	⋮	⋮
199	Kolben	5.000
237	Ölfiler	1.560
⋮	⋮	⋮
851	Schraube	25.000

# Algebraische Tabellenoperationen (3)

---

## Differenz $R \setminus S$ :

- ❑ Die Tupel zweier Relationen werden miteinander verglichen.
- ❑ Die in der ersten, nicht aber in der zweiten Relation befindlichen Tupel werden in die Ergebnisrelation aufgenommen.

$$R \setminus S := \{ r \mid r \in R \wedge r \notin S \}$$

- ❑ Differenzbildung mit dem Operator **except**, Verwendung s. union-Befehl
- ❑ Beispiel:  $R \setminus S$  (s. nächste Folie)

# Algebraische Tabellenoperationen (4)

Beispiel für eine Differenzbildung:

```
select *  
from R  
except  
select *  
from S;
```

Relation  $R$

<i>ANr</i>	<i>AName</i>	<i>Menge</i>
001	Anlasser	1.000
237	Ölfiler	1.560
199	Kolben	5.000

Relation  $S$

<i>ANr</i>	<i>AName</i>	<i>Menge</i>
851	Schraube	25.000
232	Gummiring	2.000
001	Anlasser	1.000

Ergebnisrelation  $R \setminus S$

<i>ANr</i>	<i>AName</i>	<i>Menge</i>
237	Ölfiler	1.560
199	Kolben	5.000

# Algebraische Tabellenoperationen (5)

---

## Durchschnitt $R \cap S$ :

- Alle Tupel, die sowohl in der Relationen  $R$  als auch in der Relation  $S$  enthalten sind, werden in der Ergebnisrelation zusammengefaßt.

$$R \cap S := \{ r \mid r \in R \wedge r \in S \}$$

- Durchschnittbildung mit dem Operator `intersect`, Verwendung s. union-Befehl
- Beispiel:  $R \cap S$  (s. nächste Folie)

# Algebraische Tabellenoperationen (6)

Beispiel für eine Durchschnittsbildung:

```
select *  
from R  
intersect  
select *  
from S;
```

Relation  $R$

<i>ANr</i>	<i>AName</i>	<i>Menge</i>
001	Anlasser	1.000
007	Zündkerze	1.380
199	Kolben	5.000

Relation  $S$

<i>ANr</i>	<i>AName</i>	<i>Menge</i>
001	Anlasser	1.000
199	Kolben	5.000
237	Ölfiler	1.560

Ergebnisrelation  $R \cap S$

<i>ANr</i>	<i>AName</i>	<i>Menge</i>
001	Anlasser	1.000
199	Kolben	5.000

# Duplikatelimination und Sortierordnung (1)

---

Elimination von Duplikaten im Anfrageergebnis mit dem Schlüsselwort `distinct`:

```
select distinct Oberabt  
from Abteilungen;
```

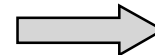


<i>Oberabt</i>
LTSW
<b>NULL</b>

Hier: Umwandlung einer Ergebnistabelle in eine *Ergebnismenge*

Erkennung und Vermeidung von Nullwerten in Spalten durch das Prädikat `is null`:

```
select distinct Oberabt  
from Abteilungen  
where Oberabt is not null;
```



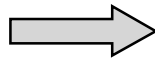
<i>Oberabt</i>
LTSW

# Duplikatelimination und Sortierordnung (2)

---

Sortierte Darstellung der Anfrageergebnisse über die `order by`-Klausel mit den Optionen `asc` (*ascending, aufsteigend*) und `desc` (*descending, absteigend*):

```
select *  
from Abteilungen  
where Oberabt  
      = 'LTSW'  
order by Kurz asc;
```



Ergebnistabelle

<i>Kurz</i>	<i>Name</i>	<i>Oberabt</i>
MFSW	Mainframe SW	LTSW
PCSW	PC SW	LTSW
UXSW	Unix SW	LTSW

Die Sortierung kann mehrere Spalten umfassen:

- Aufsteigende Sortierung aller Abteilungen gemäß des Kürzels ihrer Oberabteilung.
- Anschließend werden innerhalb einer Oberabteilung die Abteilungen absteigend gemäß ihres Kürzels sortiert.

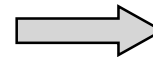
```
select *  
from Abteilungen  
order by Oberabt asc,  
         Kurz desc;
```

# Aggregatfunktionen

---

- ❑ Nutzung in der select-Klausel einer SQL-Anwendung
- ❑ Berechnung aggregierter Werte (z.B. Summe über alle Werte einer Spalte einer Tabelle)
- ❑ Beispiel: Summe und Maximum der Budgets aller Projekte

```
select sum(p.Budget) ,  
       max(p.Budget)  
from Projekte p;
```

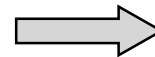


*p.Budget*

<i>sum</i>	<i>max</i>
600.000	300.000

- ❑ Außerdem Funktionen für Minimum (**min**), Durchschnitt (**avg**) und zum Zählen der Tabellenwerte einer Spalte (**count**) bzw. der Anzahl der Tupel (**count (\*)**)
- ❑ Beispiel: Anzahl der Tupel in der Relation Abteilungen

```
select count(*)  
from Abteilungen;
```



<i>count(*)</i>
5

Duplikatelimination möglich (s. Folie 3.2.53)

# Gruppierung (1)

---

- ❑ Zusammenfassung von Zeilen einer Tabelle in Abhängigkeit von Werten in bestimmten Spalten, den *Gruppierungsspalten*
- ❑ Alle Zeilen einer Gruppe enthalten in dieser Spalte bzw. diesen Spalten den gleichen Wert
- ❑ Mit Hilfe der group-Klausel erhält man auf diese Weise eine Tabelle von Gruppen, für die die Projektionsliste ausgewertet wird.
- ❑ Beispiel: Gib zu jeder Oberabteilung die Anzahl der Unterabteilungen an (s. nächste Folie)

# Gruppierung (2)

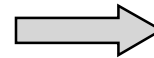
---

## Beispiel (Fortsetzung):

```
select Oberabt,  
       count(Kurz)  
from Abteilungen  
group by Oberabt;
```



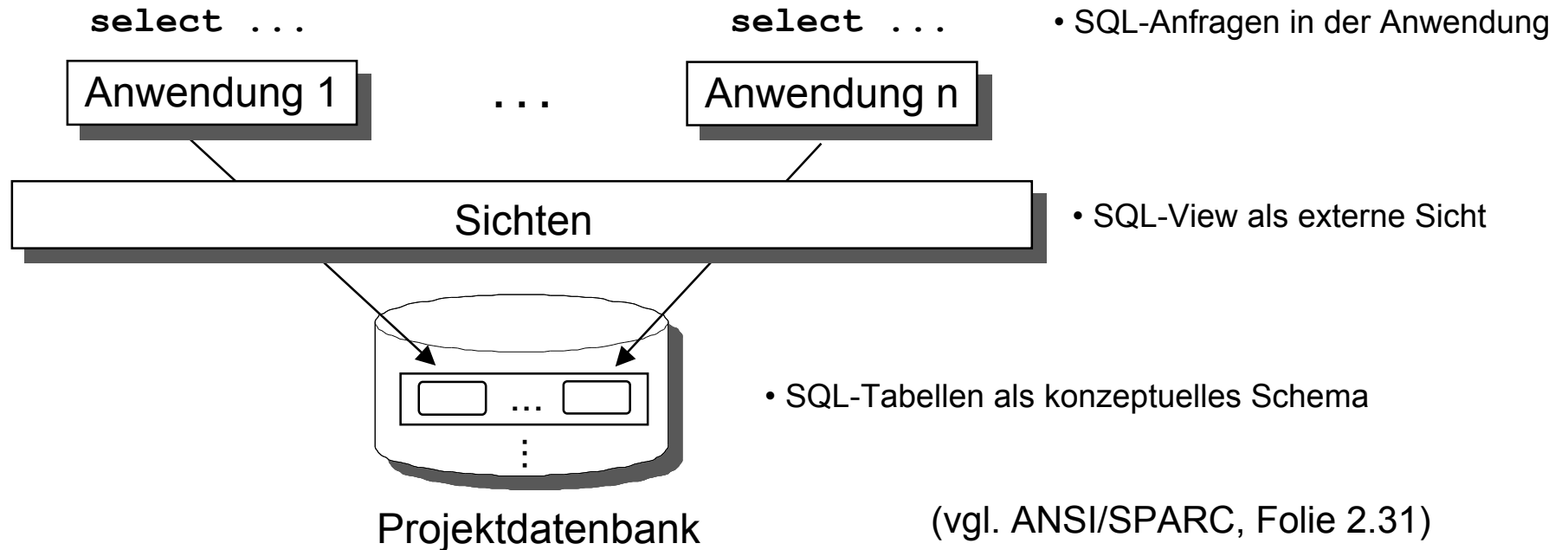
<i>Kurz</i>	<i>Name</i>	<i>Oberabt</i>
MFSW	Mainframe SW	LTSW
UXSW	Unix SW	LTSW
PCSW	PC SW	LTSW
LTSW	Leitung SW	NULL
PERS	Personal	NULL



Ergebnistabelle

<i>Oberabt</i>	<i>count(Kurz)</i>
LTSW	3
NULL	2

# Sichten (1)



## Ziel:

- Kapselung der Anwendung
- Entkopplung ... (Schemaevolution)
  - Anwendung: Externe Sicht
  - DB: Konzeptuelle Sicht

```
create view ReicheProjekte  
as select *  
from Projekte  
where Budget ≥ 200000;
```

# Sichten (2)

---

## Möglichkeiten zur monorelationalen Sichtendefinition:

- Strukturelle Einschränkung
- Prädikative Einschränkung
- Operationale Einschränkung

## Multirelationale Sichtendefinition:

- Problem: Propagierung von Änderung der Sicht auf die Datenbank ("View-Update-Problematik")
- Eine Lösung: get/set-Methoden in OODM
- Weitere Informationen:
  - F. Matthes, J.W. Schmidt. Datenbankmodelle und Datenbanksprachen. Springer, Berlin u.a., 1997.

# Aktualisierungsoperationen (1)

---

## update-Anweisung:

- Zur Modifikation von Attributwerten
- Festlegung der zu ändernden Tupel durch ein in der where-Klausel spezifiziertes Prädikat
- Beispiel:

```
update Projekte  
set Budget = 500000  
where Titel = 'DB Fahrpläne';
```

- Fehlt die where-Klausel, werden alle Tupel der Tabelle modifiziert.
- Prädikat kann sehr komplex sein.
- Mengenorientierte Berechnung des Ergebnisses vor der Durchführung des eigentlichen Updates

# Aktualisierungsoperationen (2)

---

## insert-Anweisung:

- ❑ Unspezifizierte Position des einzuführenden Tuples in der Tabelle (abhängig von der Implementierung des DBMS)
- ❑ Beispiel: Direkte Aufnahme eines neuen Tupels

```
insert into Projekte  
        (Nr, Titel, Budget)  
values (471, 'Mensaabrechnung', 900000);
```

- ❑ Auf die Angabe der Attributnamen kann bei korrekter Reihenfolge der Attributwerte verzichtet werden.
- ❑ Vorteile der Angabe von Attributnamen:
  - Vermeidung von Fehlern in der Datenbank
  - Verbesserung der Robustheit gegen Schemamodifikationen
  - Erhöhung der Lesbarkeit der Operationen

# Aktualisierungsoperationen (3)

---

## insert-Anweisung (Fortsetzung):

- ❑ Ganze Tabellen oder Teile davon können in andere kopiert werden.
- ❑ Beispiel:

```
insert into Projekte
select Nr, Titel, Budget
from ExterneProjekte
where Budget > 500000;
```

# Aktualisierungsoperationen (4)

---

## delete-Anweisung:

- ❑ Zum Löschen einzelner Tupel aus einer Tabelle
- ❑ Aufbau siehe select from where-Klausel
- ❑ Aber: Beschränkung aller Modifikationsoperationen auf *eine* Tabelle  
(❑ Angabe *mehrerer* Tabellen in der from-Klausel ist unzulässig)
- ❑ Beispiel:

```
delete  
from Projekte  
where Titel = 'Mensaabrechnung';
```

# Aktualisierungsoperationen (5)

---

**delete-Anweisung (Forts.)** - Löschen von Objekten unterschiedlicher *Granularität* :

☐ auf Tupelebene:

- Einzelne Attributwerte können zum Löschen mit der update-Anweisung auf den Wert `null` gesetzt werden.
- Löschen einzelner oder mehrerer Tupel mit Hilfe der delete-Anweisung. Die where-Klausel spezifiziert, welche Tupel gelöscht werden sollen.
- Löschen aller Werte einer Spalte (update-Anweisung ohne einschränkende where-Klausel)
- Löschen des gesamten Inhalts einer Tabelle (delete ohne where-Klausel)

☐ auf Schemaebene:

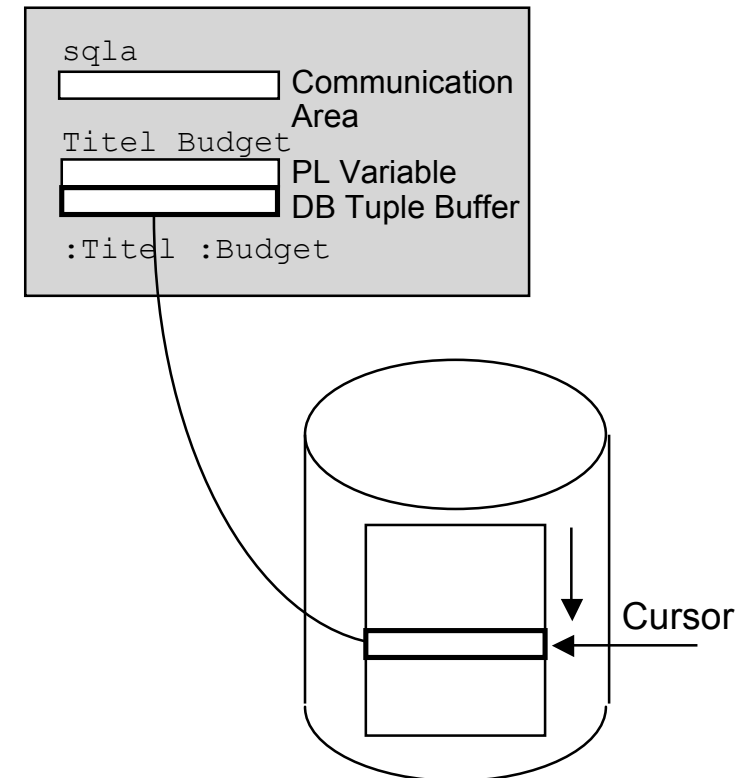
- Löschen ganzer Spalten (☐ `alter table ... drop ...`)
- Löschen einer gesamten Tabelle mitsamt der Datenstruktur (☐ `drop table ...`)

☐ Weitere Informationen:

- SQL-Standard

# SQL Programmierspracheneinbettung (1)

```
program reicheProjekte;  
EXEC SQL INCLUDE SQLCA; {global decl.}  
EXEC SQL BEGIN DECLARE SECTION;  
var  
  Titel: string;  
  Budget: float4;  
EXEC SQL END DECLARE SECTION;  
EXEC SQL DECLARE bpcursor FOR  
  SELECT Titel, Budget  
  FROM Projekte  
  WHERE Budget > 200000;  
begin  
  EXEC SQL CONNECT FirmenDB;  
  EXEC SQL OPEN bpcursor;  
  while (sqla.sqlcode = 0) do begin  
    EXEC SQL FETCH bpcursor  
      INTO :Titel, :Budget;  
    writeln(Titel, Budget);  
  end;  
  EXEC SQL CLOSE bpcursor;  
  EXEC SQL DISCONNECT  
end.
```

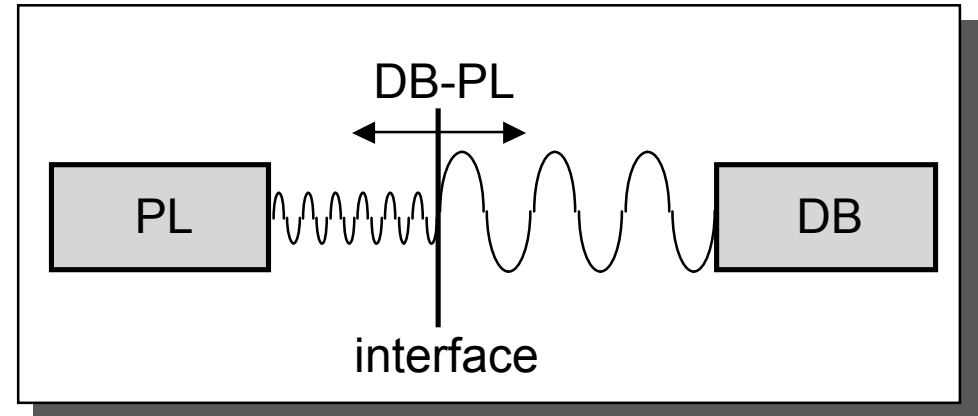


- Datenbank:  
Mengenorientierte Verarb.
- Programmiersprache:  
Tupelorientierte Verarb.

# SQL Programmierspracheneinbettung (2)

## Programmlaufzeit

- ❑ Langsame Interprozeßkommunikation
- ❑ Wiederholte Namensbindung und Typüberprüfung
- ❑ Komplizierte Cursornavigation und Fehlerbehandlung



## Programmentwicklungszeit

- ❑ Programmierer müssen zwei Sprachen sowie zusätzliche Regeln für ihre Kompatibilität erlernen.
- ❑ Aufwendige Codierung des Datenaustauschs zwischen Hauptspeicher und Sekundärspeicher (30 % des Gesamtcode)
- ❑ "Low-level" Typkonversionen
- ❑ Ungeeignete Werkzeuge für Schemaänderungen und Debugging

**„Impedance Mismatch“**