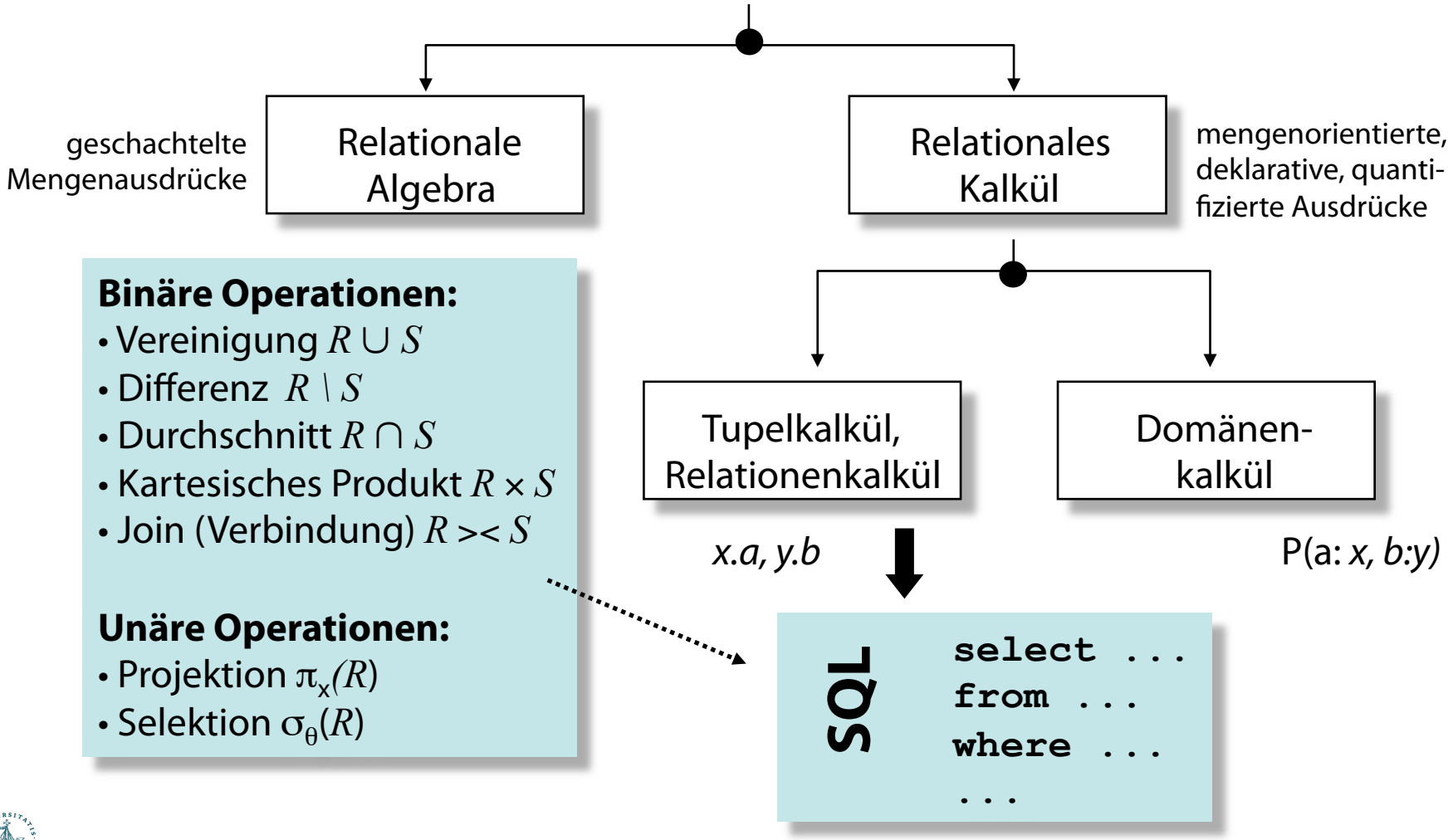
Datenbanken

Prof. Dr. Ralf Möller
Universität zu Lübeck
Institut für Informationssysteme

Marc Stelzner (Übungen)
Torben Matthias Kempfert (Tutor)
Maurice-Raphael Sambale (Tutor)

Wiederholung: Typen von Anfragesprachen

Relationale Anfragesprachen im Überblick:



Binäre Operationen:
- Vereinigung $R \cup S$
- Differenz $R \setminus S$
- Durchschnitt $R \cap S$
- Kartesisches Produkt $R \times S$
- Join (Verbindung) $R >< S$

Unäre Operationen:
- Projektion $\pi_x(R)$
- Selektion $\sigma_\theta(R)$

RDM: Anfragen im relationalen Kalkül (2)

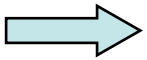
Im **Tupelkalkül** werden die Tupelvariablen *x, y*, ... der logischen und relationalen Quantoren explizit an Tupel aus Relationen gebunden.

Beispiel:

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

Projekte

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



Ergebnisrelation

Nr	*Budget*
100	300.000
300	200.000

RDM: Anfragen im relationalen Kalkül (3)

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

Beispiel:

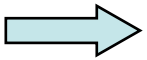
```
x as int, y as float;

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

fiktive Syntax, kein SQL

Projekte

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



Nr	*Budget*
100	300.000

Tupelkalkül vs. Domänenkalkül

- Unendliche Relationen im Domänenkalkül beschreibbar:

```
x as int, y as float;

select x, y
where (x < 100) and (y > 250000)
```

fiktive Syntax, kein SQL

- Unendliche Relationen nicht handhabbar: Syntaktische Einschränkungen im Domänenkalkül notwendig

- Vorteil für Tupelkalkül (SQL)

Tupelkalkül vs. Domänenkalkül

- Unendliche Relationen im Domänenkalkül beschreibbar:

```
x as int, y as int;

select x, y
where (x < 100) and (y > 250000)
```

fiktive Syntax, kein SQL

- Unendliche Relationen nicht handhabbar: Syntaktische Einschränkungen im Domänenkalkül notwendig

- Vorteil für Tupelkalkül (SQL)

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

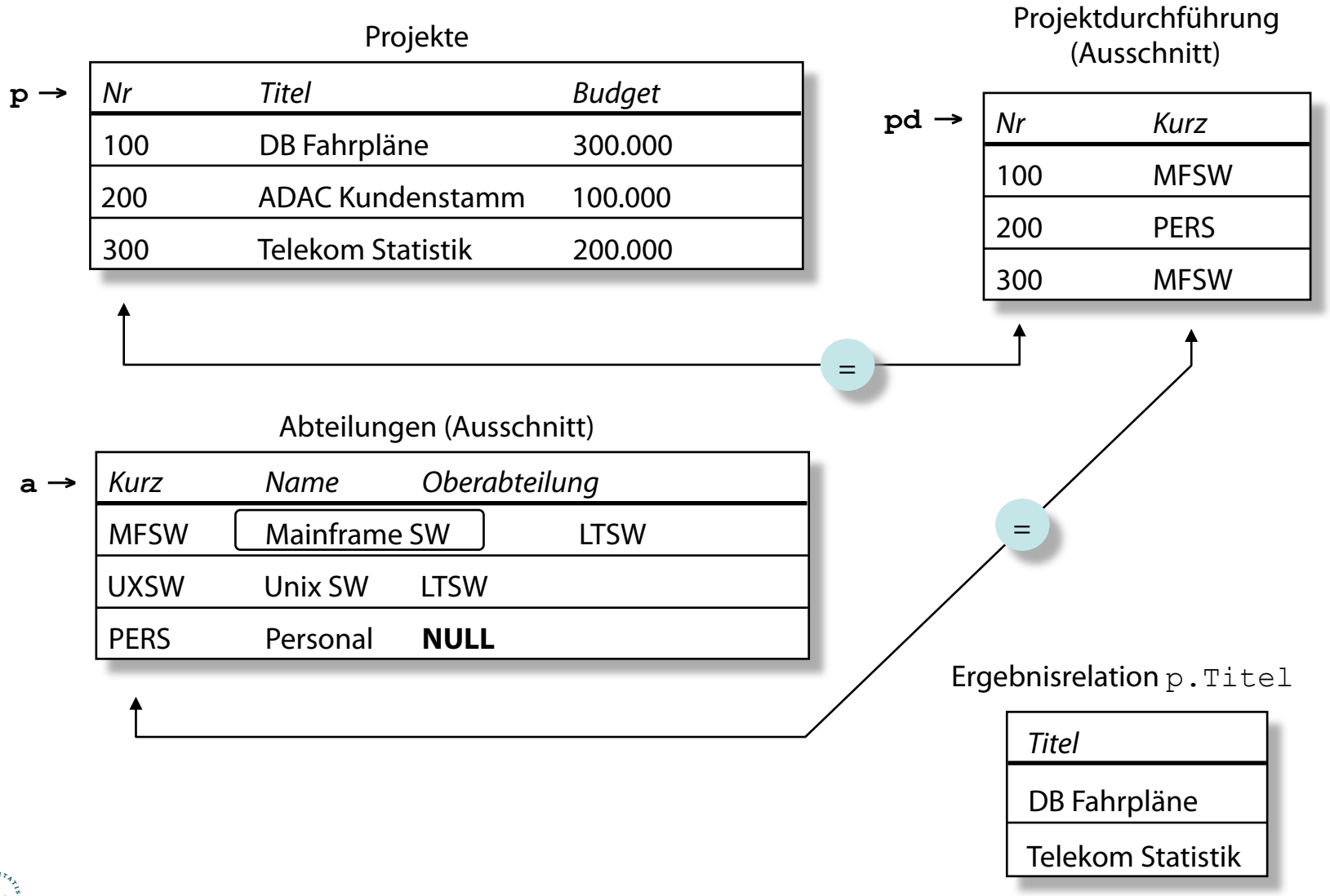
Join: Mehrere Tabellen werden wertbasiert, z.B. über gleiche Werte in zusammengehörigen Primärschlüssel/ Fremdschlüssel-Paaren, miteinander verknüpft.

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 *Projekt-durchführung* 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 typsisiert),
Änderungsoperationen beziehen sich auf Relationen oder Teilrelationen (select …):

- **insert-Statement:**
  - Fügt ein einziges Tupel ein, dessen Attributwerte als Parameter übergeben werden.
  - Fügt eine Ergebnistabelle ein.
- **update-Statement:**
  - Selektion (des) der betreffenden Tupel(s)
  - Neue Werte oder Formeln für zu ändernde Attribute

```
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 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';
```

SQL-Standardisierung

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

SQL-2011:

- SQL:2011 or ISO/IEC 9075:2011

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 "_".
- **Schlüsselworte**: SQL definiert über 210 Namen als Schlüsselworte, die nicht kontextsensitiv sind.
- **Begrenzte Namen** sind Zeichenketten in doppelten Anführungszeichen. Durch begrenzte Namen kann verhindert werden, daß neu hinzugekommene Schlüsselworte mit gewählten Bezeichnern kollidieren. (→ *Syntaxerweiterungsproblematik*)
- **Literale** dienen zur Benennung von Werten der SQL-Basistypen
- weitere Symbole (Operatoren etc.)

```
Peter, mary33
```

```
create, select
```

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

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

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

Schemata und Kataloge (1)

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

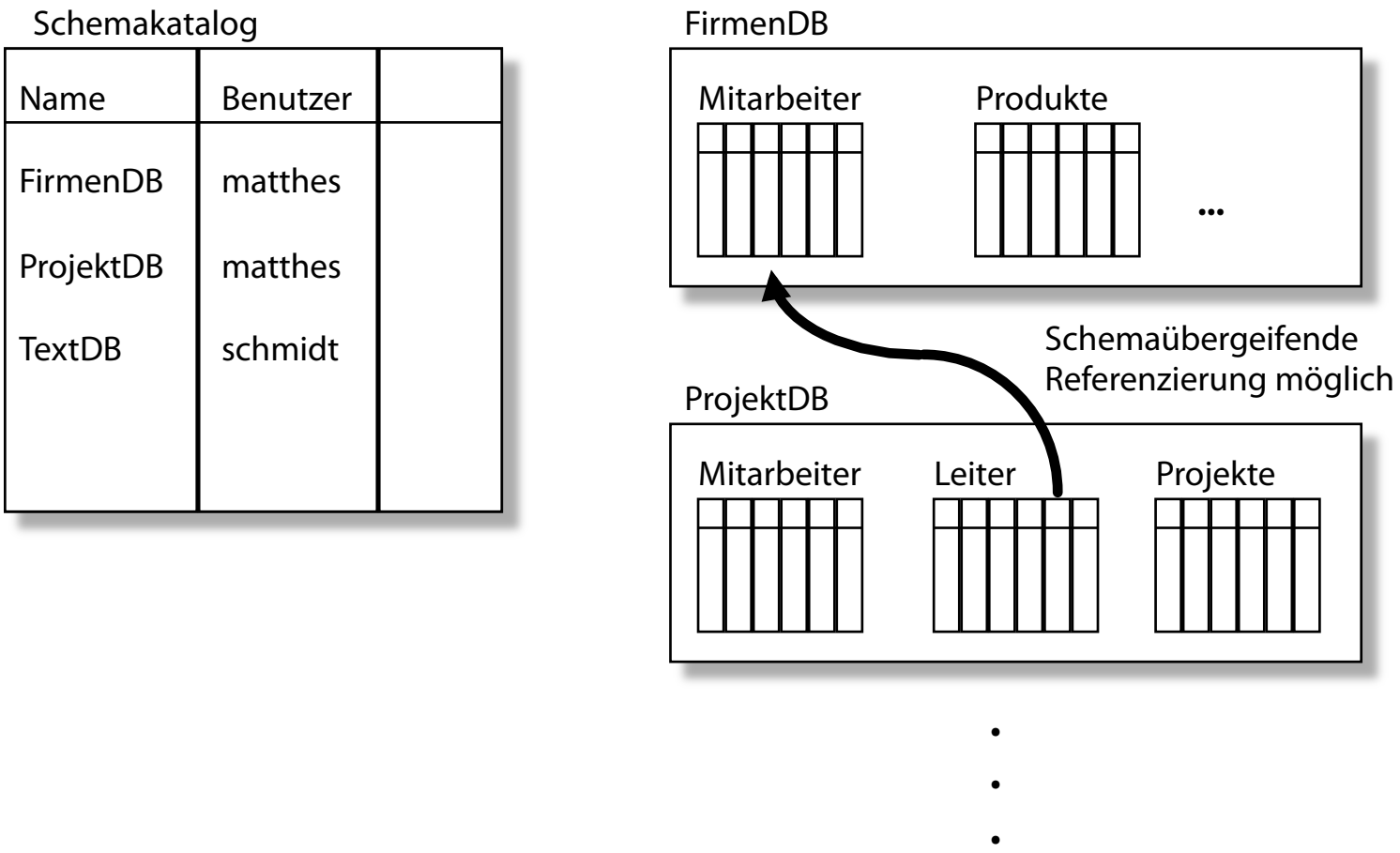
```
create schema FirmenDB;
  create table Mitarbeiter ...;
  create table Produkte ... ;

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.
- Schemata werden
  - zur Übersetzungszeit von SQL-Modulen oder
  - dynamisch als Seiteneffekt von Anweisungen

  definiert.
- 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.

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

```
create schema FirmenDB
connect FirmenDB
```

```
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 Tabellennamen 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.

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

- **Approximate numerics** bieten aufgrund ihrer Fließkommadarstellung einen flexiblen Wertebereich, sind jedoch wegen der Rundungsproblematik nicht für kaufmännische Anwendungen geeignet.

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

- **Character strings** beschreiben mit Leerzeichen aufgefüllte Zeichenketten fester Länge oder variabel lange Zeichenketten mit fester Maximallänge.

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

- **Bit strings** beschreiben mit Null aufgefüllte Bitmuster fester Länge oder variabel lange Bitfelder mit fester Maximallänge.

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

Basisdatentypen und Typkompatibilität (4)

- **Datetime** Basistypen beschreiben Zeit(punkt)werte vorgegebener Granularität.
- **Time intervals** beschreiben Zeitintervalle vorgegebener Dimension und Granularität.

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

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

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

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 < **`null`** < 1 < 2 < ... ?)
- Algebraische Eigenschaften werden häufig zur *Anfrageoptimierung* ausgenutzt. Eine Anfrage, die Werte eines Datentyps `T´` (wobei `null`$\in$`T´`) verwendet, bietet im Regelfall weniger Optimierungsspielraum als eine Anfrage mit Werten des Datentyps `T` (mit `null`$\notin$`T`) .

Nullwerte und Wahrheitswerte (3)

Wahrheitstabellen der dreiwertigen SQL-Logik:

OR	true	false	null
true	*true*	*true*	*true*
false	*true*	*false*	*null*
null	*true*	*null*	*null*

AND	true	false	null
true	*true*	*false*	*null*
false	*false*	*false*	*false*
null	*null*	*false*	*null*

x	not x	x is null	x is not null
true	*false*	*false*	*true*
false	*true*	*false*	*true*
null	*null*	*true*	*false*

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

Nullwerte und Wahrheitswerte (4)

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 waren 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)

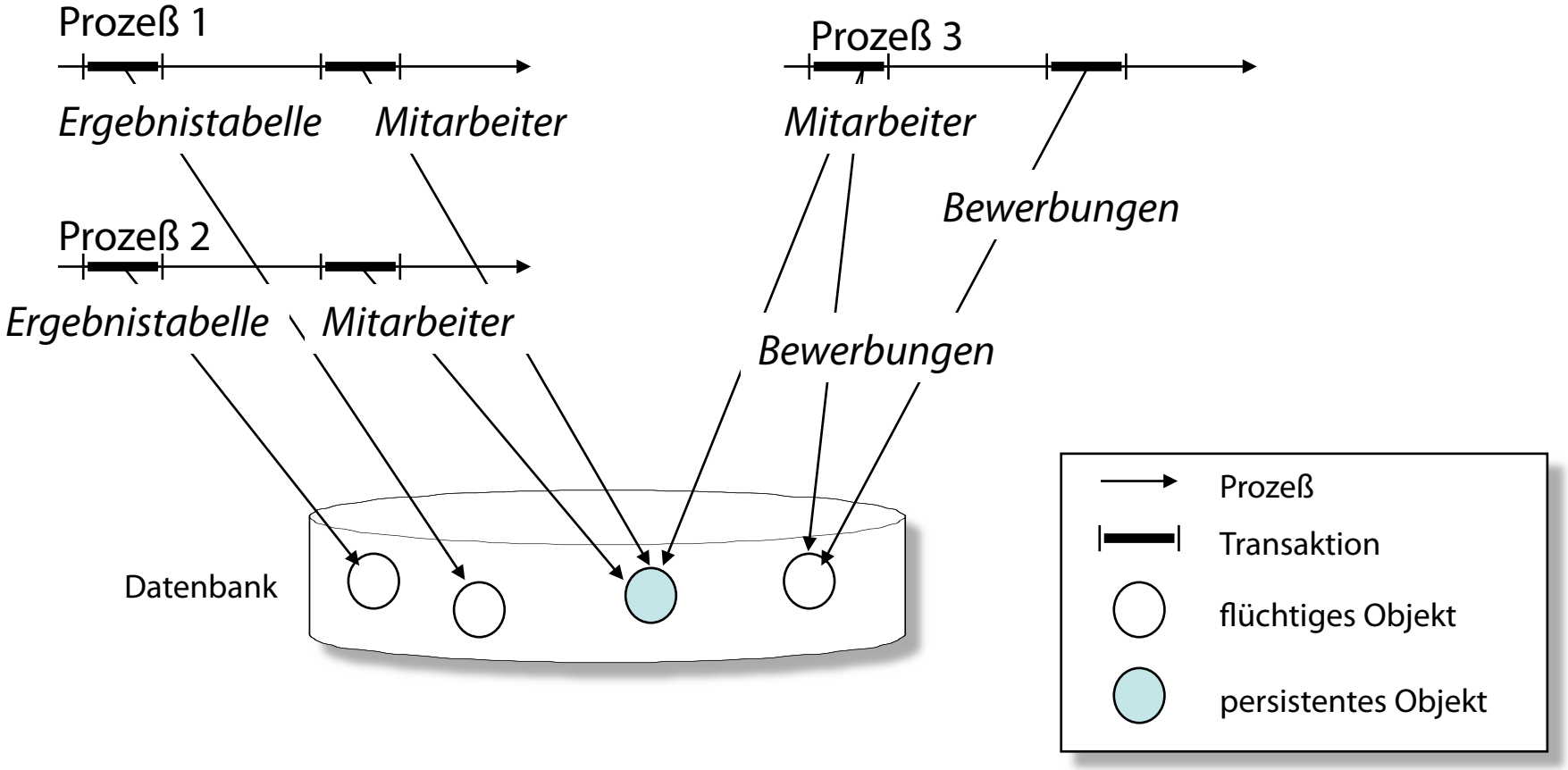
- Die dynamisch variierende Zahl der Reihen bezeichnet man mit *Kardinalität.*
- Im Gegensatz zum RDM sind in Tabellen Duplikate erlaubt.
- Eine Reihe ist ein Duplikat einer anderen Reihe, wenn beide in allen Spalten gemäß der Äquivalenzrelation der jeweiligen Spaltentypen übereinstimmen.
- Tabellen 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 (Details später)

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.
- **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 (3)

Historisch gesehen haben sich Datenbanksysteme auf *persistente globale* Datenobjekte konzentriert, sie unterstützen aber auch flüchtige Objekte.

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* (Sperren, 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)
```

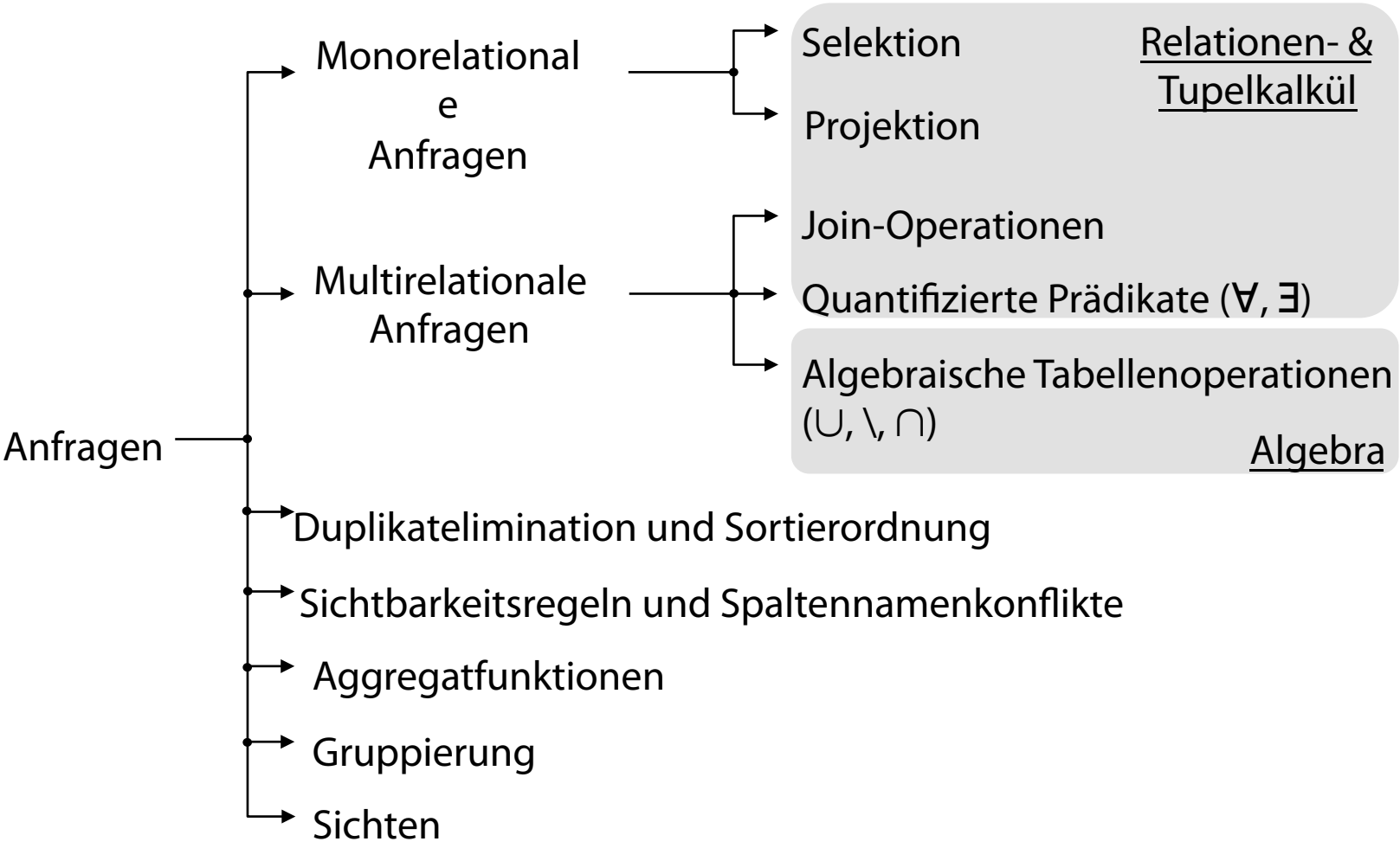
Die fehlenden Werte werden mit **`null`** oder mit bei der Tabellenerzeugung angegebenen Standardwerten belegt.

- 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

Projekte

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

Projektdurchfuehrungen

Nr	*Kurz*
100	MFSW
100	UXSW
100	LTSW
200	UXSW
200	PERS
300	MFSW

Abteilungen

Kurz	*Name*	*Oberabt*
MFSW	Mainframe SW	LTSW
UXSW	Unix SW	LTSW
PCSW	PC SW	LTSW
LTSW	Leitung SW	**NULL**
PERS	Personal	**NULL**

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.

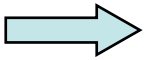
– Direkte Verwendung eines Anfrageergebnisses als Bereichsrelation einer komplexen Anfrage.

  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';
```

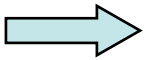


Ergebnistabelle

Name	*Kurz*
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';
```



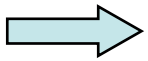
Ergebnistabelle

Kurz	*Name*	*Oberabt*
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

Oberabt
LTSW
LTSW
LTSW
NULL
NULL

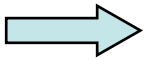
Flüchtige Kopie einer Datenbanktabelle `T`:

```
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

Unter	*Ober*
MFSW	LTSW
UXSW	LTSW
PCSW	LTSW
LTSW	**NULL**
PERS	**NULL**

Multirelationale Anfragen (1)

- Anfragen, bei denen Spalten und Zeilen mehrerer *Bereichsrelationen* (T_1, ..., 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 $T_1, ..., T_n$
where *Selektionsprädikat;*

lokale Bereichsvariable

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

Multirelationale Anfragen (2)

Beispiel: Projekte $><_{(Nr = Nr)}$ Projektdurchfuehrungen

Projekte

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

Projektdurchfuehrungen (Ausschnitt)

Nr	*Kurz*
100	MFSW
200	PERS
300	MFSW

Ergebnisrelation

Nr	*Titel*	*Budget*	*Nr2*	*Kurz*
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 ≥ 2 *Bereichstabellen*
- **Equi-Join:** Als *Selektionsprädikat* wird ein Gleicheitstest (=) 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-, Bereichsvariablennamen) innerhalb kalkülorientierter SQL-Anfragen:

- In den Teilausdrücken P, S, T_1, ..., T_n sind alle globalen Namen von SQL-Objekten (z.B. Tabellen, Sichten, Schemata, Kataloge) sichtbar.

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

- 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.

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;
```



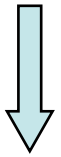
```
select m.*
from Mitarbeiter  m,
     Projekte p
where m.Projekte =
      p.Nr;
```

Sichtbarkeitsregeln und Spaltennamenkonflikte (3)

Beispiel: Reflexive Anfrage

- Hier: Tabelle der Ober- und Unterabteilungen
- Verallgemeinerung:
  - Rekursive Anfragen → nicht mit jedem System möglich

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



Oberabteilung	*Unterabteilung*
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.

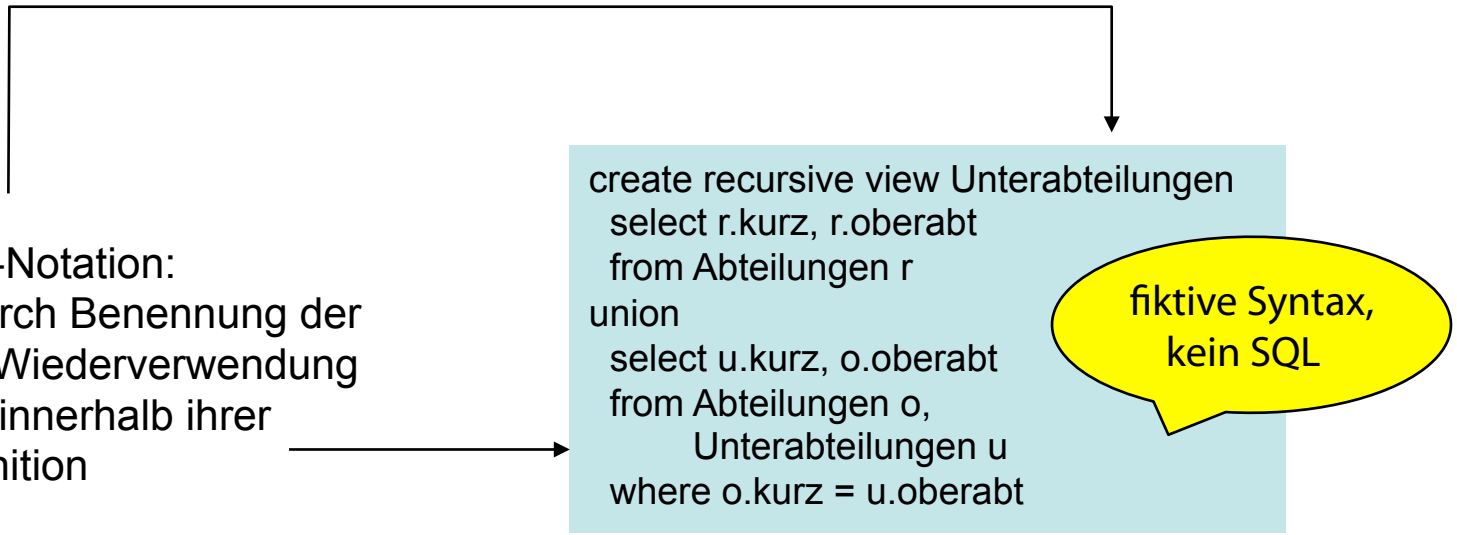
Für jede Hierarchieebene muß eine eigene Anfrage formuliert und mit **union** dem Gesamtergebnis hinzugefügt werden.

```
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



- 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.

Algebraische Tabellenoperationen (1)

Vereinigung $R \cup S$:

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

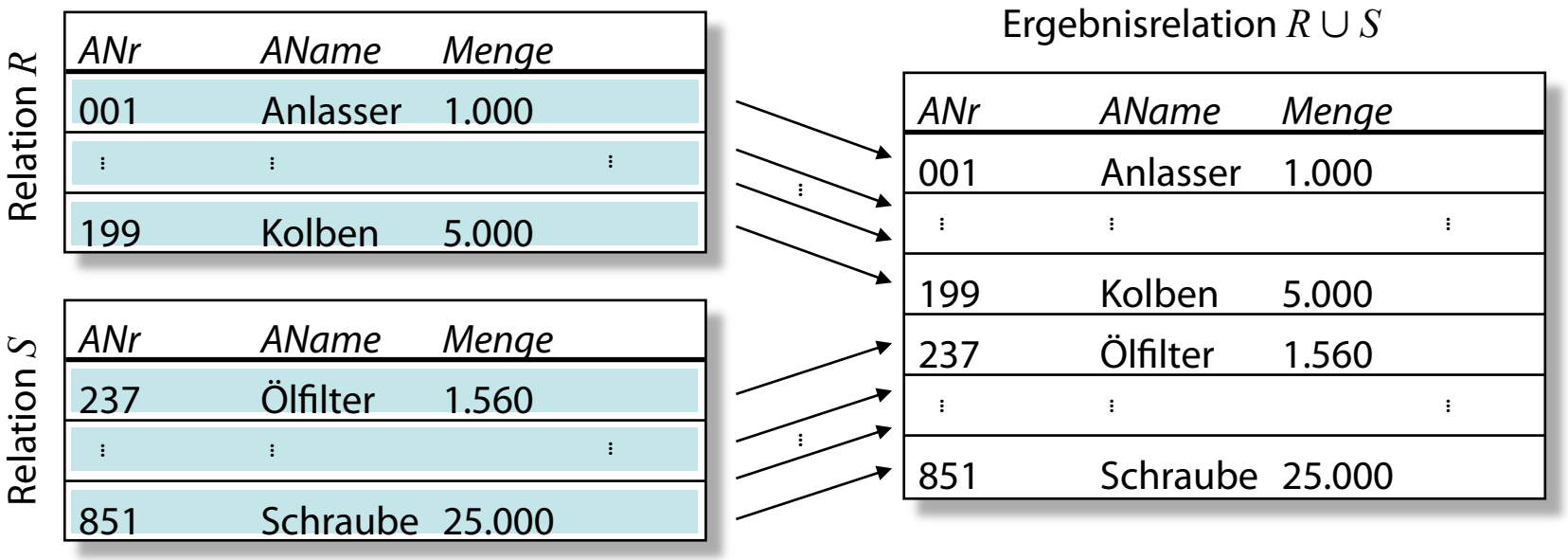
$$R \cup S := \{ r \mid r \in R \vee r \in S \}$$

- 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: $R \cup S$ (s. nächste Folie)

Algebraische Tabellenoperationen (2)

Beispiel für eine Vereinigung:

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



Spezielles Konstrukt: Corresponding

- Bei union muss der Grad der beteiligten Relationen gleich sein
- Die Spaltennamen des Ergebnisses entsprechen der ersten Relation (ggf. hier Umbenennungen mit AS vornehmen)
- Sonderfall: Korrespondierende (gleichnamige) Spalten bilden und UNION durchführen:

```
select count(*)
from ( Professoren
        union corresponding
        Studenten );
```

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

ANr	*AName*	*Menge*
001	Anlasser	1.000
237	Ölfilter	1.560
199	Kolben	5.000

Relation S

ANr	*AName*	*Menge*
851	Schraube	25.000
232	Gummiring	2.000
001	Anlasser	1.000

Ergebnisrelation $R \setminus S$

ANr	*AName*	*Menge*
237	Ölfilter	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.

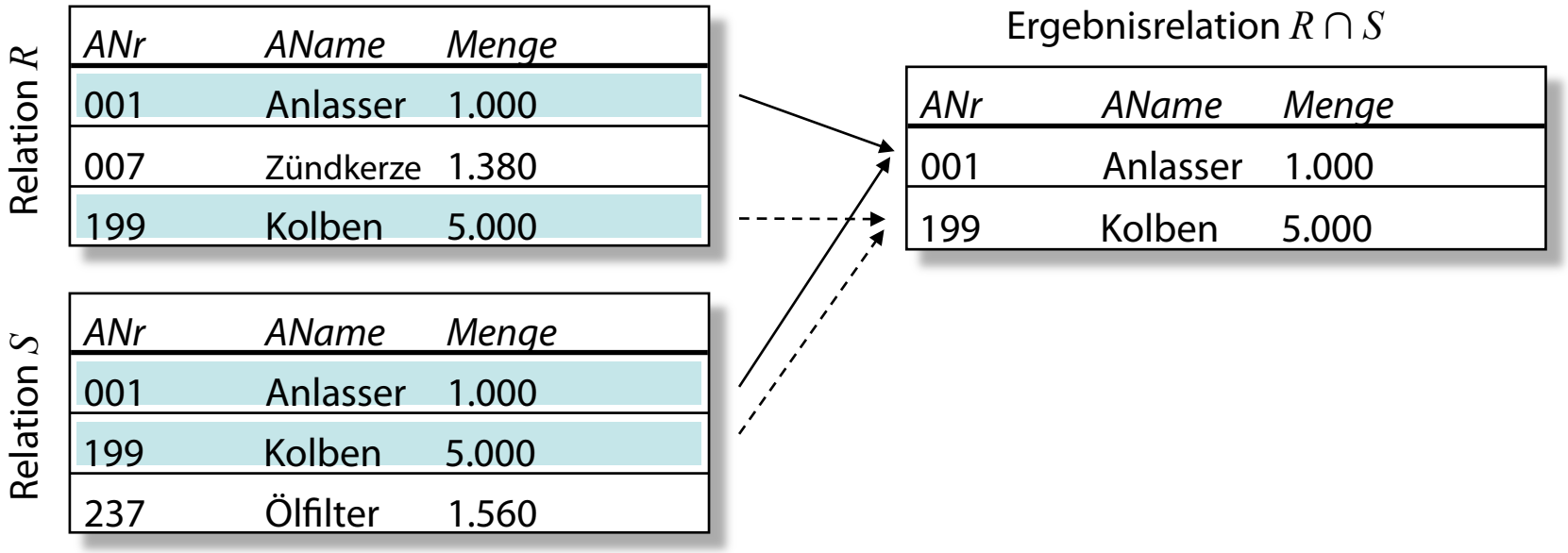
$$R \cap S := \{ r \mid r \in R \wedge r \in S \}$$

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

Algebraische Tabellenoperationen (6)

Beispiel für eine Durchschnittsbildung:

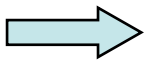
```
select *
from R
intersect
select *
from S;
```



Duplikatelimination und Sortierordnung (1)

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

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

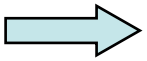


Oberabt
LTSW
NULL

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;
```

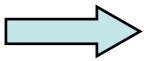


Oberabt
LTSW

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

Kurz	*Name*	*Oberabt*
MFSW	Mainframe SW	LTSW
PCSW	PC SW	LTSW
UXSW	Unix SW	LTSW

Die Sortierung kann mehrere Spalten umfassen:

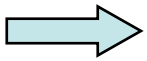
- Aufsteigende Sortierung aller Abteilungen gemäß des Kürzels ihrer Oberabteilung.
- Anschließend werden innerhalb einer Oberabteilung die Abteilungen absteigend gemäß ihres Kürzels sortiert.

```
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;
```

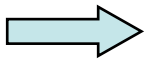


p.Budget

sum	*max*
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;
```



count(*)
5

Duplikatelimination möglich

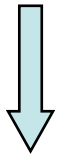
Gruppierung (1)

- 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)

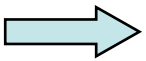
Gruppierung (2)

Beispiel (Fortsetzung):

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



Kurz	*Name*	*Oberabt*
MFSW	Mainframe SW	LTSW
UXSW	Unix SW	LTSW
PCSW	PC SW	LTSW
LTSW	Leitung SW	**NULL**
PERS	Personal	**NULL**



Ergebnistabelle

Oberabt	*count(Kurz)*
LTSW	3
NULL	2

Aggregatfunktion und Gruppierung

Aggregatfunktionen **avg, max, min, count, sum**

```
select avg (Semester)
from Studenten;

select gelesenVon, sum (SWS)
from Vorlesungen
group by gelesenVon;

select gelesenVon, Name, sum (SWS)
from Vorlesungen, Professoren
where gelesenVon = PersNr and Rang = 'C4'
group by gelesenVon, Name
        having avg (SWS) >= 3;
```

Professoren

PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

Studenten

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Vorlesungen

VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

voraussetzen

Vorgänger	Nachfolger
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

hören

MatrNr	VorlNr
26120	5001
27550	5001
27550	4052
28106	5041
28106	5052
28106	5216
28106	5259
29120	5001
29120	5041
29120	5049
29555	5022
25403	5022

prüfen

MatrNr	VorlNr	PersNr	Note
28106	5001	2126	1
25403	5041	2125	2
27550	4630	2137	2

Assistenten

PersINr	Name	Fachgebiet	Boss
3002	Platon	Ideenlehre	2125
3003	Aristoteles	Syllogistik	2125
3004	Wittgenstein	Sprachtheorie	2126
3005	Rhetikus	Planetenbewegung	2127
3006	Newton	Keplersche Gesetze	2127
3007	Spinoza	Gott und Natur	2126

Besonderheiten bei Aggregatoperationen

SQL erzeugt pro Gruppe ein Ergebnistupel

Deshalb müssen alle in der **select**-Klausel aufgeführten Attribute - außer den aggregierten – auch in der **group by**-Klausel aufgeführt werden

Nur so kann SQL sicherstellen, dass sich das Attribut nicht innerhalb der Gruppe ändert

Name muß also in der letzten Anfrage auf der vorigen Folie hinzukommen.

Ausführen einer Anfrage mit group by

Vorlesung x Professoren							
VorlNr	Titel	SWS	gelesen Von	PersNr	Name	Rang	Raum
5001	Grundzüge	4	2137	2125	Sokrates	C4	226
5041	Ethik	4	2125	2125	Sokrates	C4	226
...	...	...	...	...	...	...	...
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

⇓ **where**-Bedingung

VorlNr	Titel	SWS	gelesen Von	PersNr	Name	Rang	Raum
5001	Grundzüge	4	2137	2137	Kant	C4	7
5041	Ethik	4	2125	2125	Sokrates	C4	226
5043	Erkenntnistheorie	3	2126	2126	Russel	C4	232
5049	Mäeutik	2	2125	2125	Sokrates	C4	226
4052	Logik	4	2125	2125	Sokrates	C4	226
5052	Wissenschaftstheorie	3	2126	2126	Russel	C4	232
5216	Bioethik	2	2126	2126	Russel	C4	232
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

Gruppierung

VorlNr	Titel	SWS	gelesenVon	PersNr	Name	Rang	Raum
5041	Ethik	4	2125	2125	Sokrates	C4	226
5049	Mäeutik	2	2125	2125	Sokrates	C4	226
4052	Logik	4	2125	2125	Sokrates	C4	226
5043	Erkenntnistheorie	3	2126	2126	Russel	C4	232
5052	Wissenschaftstheo.	3	2126	2126	Russel	C4	232
5216	Bioethik	2	2126	2126	Russel	C4	232
5001	Grundzüge	4	2137	2137	Kant	C4	7
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

⇓ **having**-Bedingung

VorlNr	Titel	SWS	gelesenVon	PersNr	Name	Rang	Raum
5041	Ethik	4	2125	2125	Sokrates	C4	226
5049	Mäeutik	2	2125	2125	Sokrates	C4	226
4052	Logik	4	2125	2125	Sokrates	C4	226
5001	Grundzüge	4	2137	2137	Kant	C4	7
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

⇓ Aggregation (**sum**) und Projektion

Ergebnis

gelesenVon	Name	**sum** (SWS)
2125	Sokrates	10
2137	Kant	8

Elementtest

Beispiel für einen Elementtest

```
select Name
from Professoren
where PersNr in (select gelesenVon
                 from Vorlesungen)
```

```
select Name
from Professoren
where PersNr not in (select gelesenVon
                     from Vorlesungen)
```

Elementtest mit geschachtelter Anfrage häufig ersetzbar durch nichtgeschachtelte Anfrage mit Join

Quantifizierte Prädikate (eingeschränkte Form)

Universelle Quantifizierung:

- $\{ x \in R \mid \forall y \in S : x \, \theta \, y \}$
- 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:

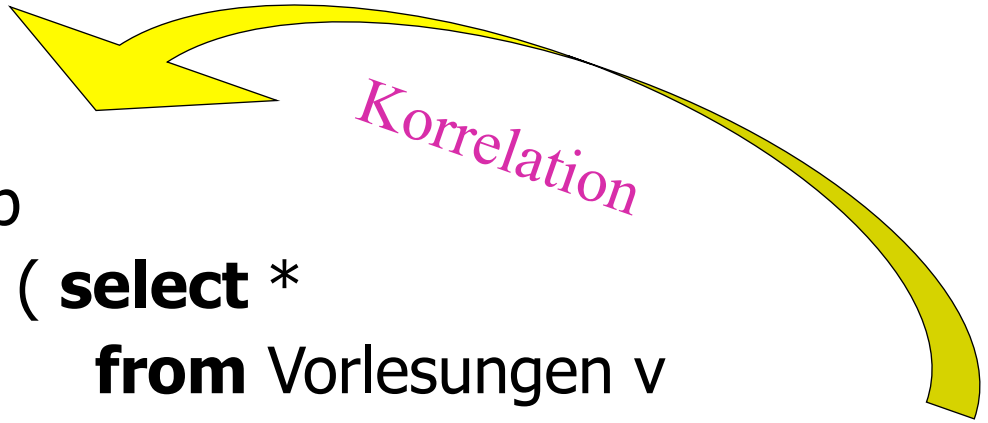
- $\{ x \in R \mid \exists y \in S : x \, \theta \, y \}$
- Hier: Tabelle aller Projekte x, die mindestens an *einer* Projektdurchführung y beteiligt sind
- **= any** synonym zu **in**.

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

Existenzquantor **exists**

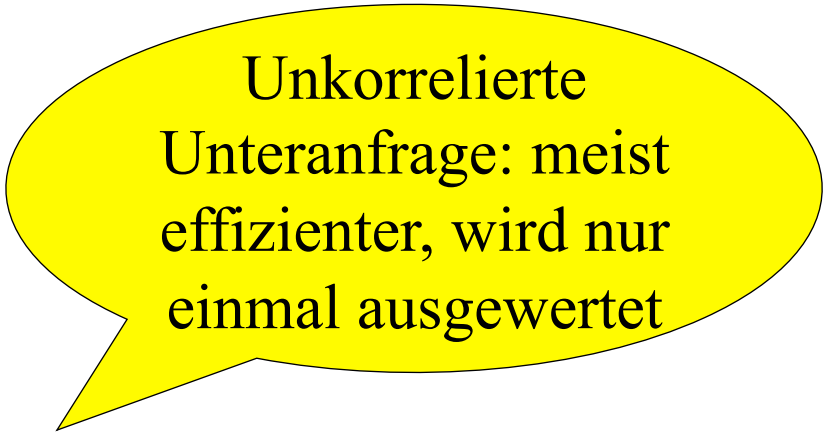
```
select p.Name
from Professoren p
where not exists ( select *
                   from Vorlesungen v
                   where v.gelesenVon = p.PersNr );
```

Existenzquantor **exists**




```
select p.Name
from Professoren p
where not exists ( select *
                   from Vorlesungen v
                   where v.gelesenVon = p.PersNr );
```

Realisierung als Mengenvergleich




```
select Name
from Professoren
where PersNr not in ( select gelesenVon
                      from Vorlesungen );
```

Der Vergleich mit "all"

Kein vollwertiger Allquantor!

```
select Name
from Studenten
where Semester >= all ( select Semester from Studenten );
```

Allquantifizierung

SQL-92 hat keinen Allquantor

Allquantifizierung muß also durch eine äquivalente Anfrage mit Existenzquantifizierung ausgedrückt werden

Logische Formulierung der Anfrage: Wer hat **alle** vierstündigen Vorlesungen gehört?

$$\{s \mid s \in \text{Studenten} \Lambda \forall v \in \text{Vorlesungen}\ (v.SWS=4 \Rightarrow \exists h \in \text{hören}\ (h.VorlNr=v.VorlNr \Lambda\ h.MatrNr=s.MatrNr))\}$$

Elimination von $\forall$ und $\Rightarrow$

Dazu sind folgende Äquivalenzen anzuwenden

$$\forall t \in R\ (P(t)) = \neg(\exists t \in R(\neg\ P(t)))$$

$$R \Rightarrow T = \neg R \vee T$$

Umformung des Kalkül-Ausdrucks ...

Wir erhalten

$$\{s \mid s \in \text{Studenten} \land \neg (\exists v \in \text{Vorlesungen}\ \neg(\neg(v.SWS=4) \lor \exists h \in \text{hören}\ (h.VorlNr=v.VorlNr \land h.MatrNr=s.MatrNr))\}$$

Anwendung von DeMorgan ergibt schließlich:

$$\{s \mid s \in \text{Studenten} \land \neg (\exists v \in \text{Vorlesungen}\ (v.SWS=4 \land \neg(\exists h \in \text{hören}\ (h.VorlNr=v.VorlNr \land h.MatrNr=s.MatrNr))))\}$$

SQL-Umsetzung folgt direkt:

```
select s.*
from Studenten s
where not exists
   (select *
    from Vorlesungen v
    where v.SWS = 4 and not exists
       (select *
        from hören h
        where h.VorlNr = v.VorlNr and h.MatrNr=s.MatrNr ) );
```

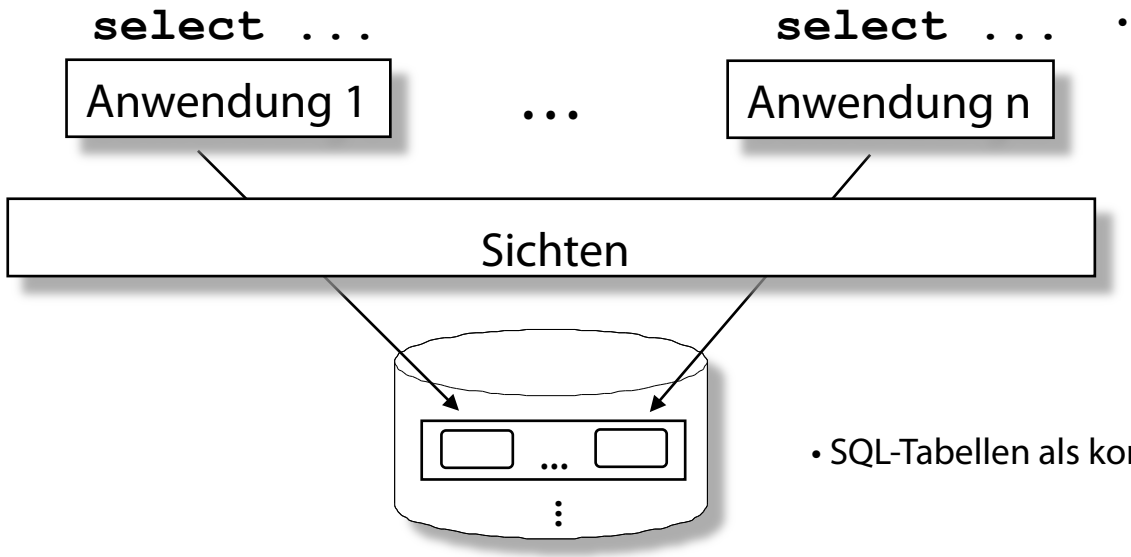
Allquantifizierung durch count-Aggregation

Allquantifizierung kann immer auch durch eine **count-**Aggregation ausgedrückt werden

Wir betrachten dazu eine etwas einfachere Anfrage, in der wir die *(MatrNr* der) Studenten ermitteln wollen, die *alle* Vorlesungen hören:

```
select h.MatrNr
from hören h
group by h.MatrNr
        having count (*) = (select count (*) from Vorlesungen);
```

Sichten (1)



- SQL-Anfragen in der Anwendung
- SQL-View als externe Sicht
- SQL-Tabellen als konzeptuelles Schema

Projektdatenbank

(vgl. ANSI/SPARC)

Ziel:

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

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

Sichten ...

für den Datenschutz

```
create view prüfenSicht as
    select MatrNr, VorlNr, PersNr
    from prüfen
```

Sichten ...

für die Vereinfachung von Anfagen

```
create view StudProf (Sname, Semester, Titel, Pname) as
    select s.Name, s.Semester, v.Titel, p.Name
    from Studenten s, hören h, Vorlesungen v, Professoren p
    where s.MatrNr=h.MatrNr and h.VorlNr=v.VorlNr and
              v.gelesenVon = p.PersNr
```

```
select distinct Semester
from StudProf
where PName='Sokrates';
```

Sichten zur Modellierung von Generalisierung

```
create table Angestellte
        (PersNr      integer not null,
        Name varchar (30) not null);
create table ProfDaten
        (PersNr      integer not null,
        Rang  character(2),
        Raum         integer);
create table AssiDaten
        (PersNr              integer not null,
        Fachgebiet  varchar(30),
        Boss                 integer);
```

```
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;
```

➔ Untertypen als Sicht

```
create table Professoren
        (PersNr integer not null,
        Name            varchar (30) not null,
        Rang            character (2),
        Raum            integer);
create table Assistenten
        (PersNr integer not null,
        Name            varchar (30) not null,
        Fachgebiet      varchar (30),
        Boss            integer);
create table AndereAngestellte
        (PersNr integer not null,
        Name            varchar (30) not null);
```

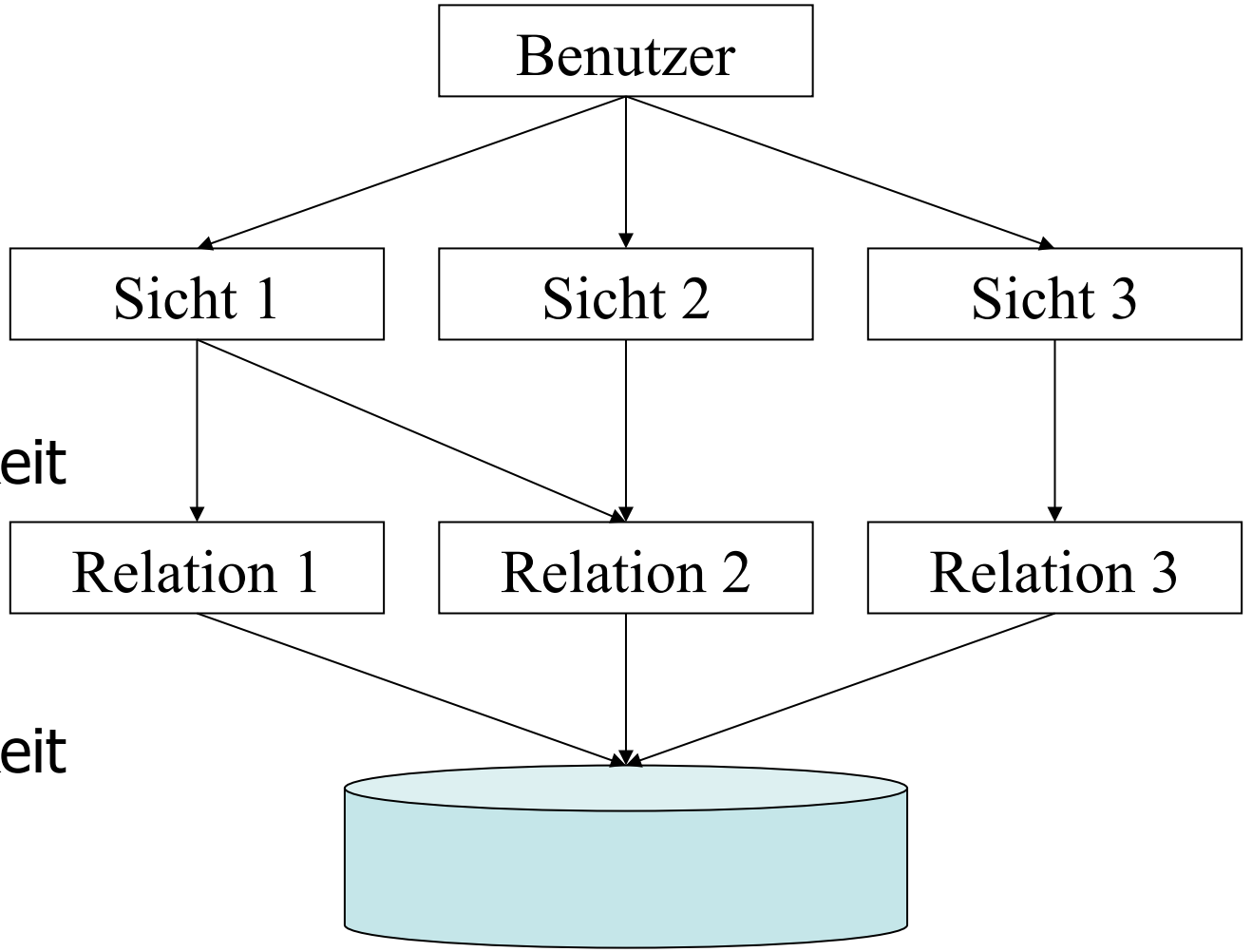
```
create view Angestellte as
        (select PersNr, Name
        from Professoren)
          union
        (select PersNr, Name
        from Assistenten)
         union
        (select *
        from AndereAngestellte);
```

➔ Obertypen als Sicht

Sichten zur Gewährleistung von Datenunabhängigkeit

logische Datenunabhängigkeit

physische Datenunabhängigkeit



Änderbarkeit von Sichten

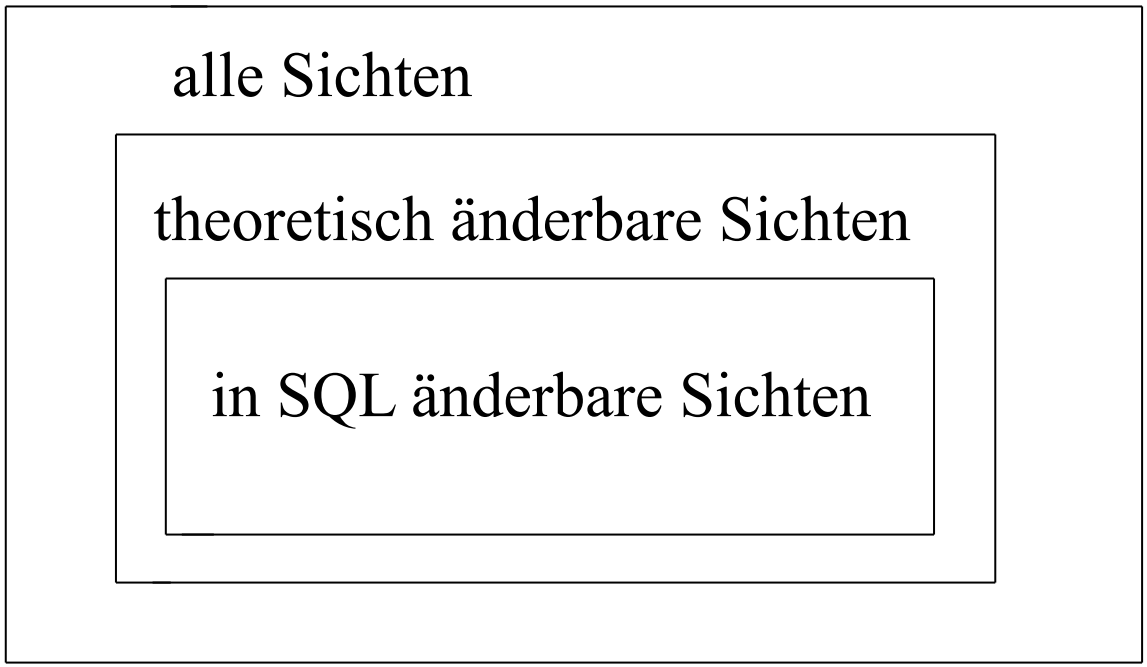
Beispiele für nicht änderbare Sichten

```
create view WieHartAlsPrüfer (PersNr, Durchschnittsnote) as
    select PersNr, avg(Note)
    from prüfen
    group by PersNr;
create view VorlesungenSicht as
    select Titel, SWS, Name
    from Vorlesungen, Professoren
    where gelesen Von=PersNr;
insert into VorlesungenSicht
    values ('Nihilismus', 2, 'Nobody');
```

Änderbarkeit von Sichten

in SQL

- nur eine Basisrelation
- Schlüssel muss vorhanden sein
- keine Aggregatfunktionen, Gruppierung und Duplikateliminierung



Integritätssicherung in SQL (1)

SQL erzwingt die folgenden *SQL-inhärenten Integritätsbedingungen* durch textuelle Analyse der Anweisungen unter Benutzung der Schemainformationen (→ *statische Typisierung* in Programmiersprachen):

- **Typisierung der Spalten**: In einer Spalte können nur typkompatible Werte gespeichert werden.
- **Homogenität der Reihen**: Alle Reihen einer Tabelle besitzen eine identische Spaltenstruktur.

Zur *applikationsspezifischen Integritätssicherung* stehen zwei syntaktische Konstrukte zur Verfügung, die beide Boole' sche Prädikate zur *deklarativen* Integritätssicherung benutzen und zur Laufzeit erzwungen werden:

- **Domänenzusicherungen**:
  Basistypen mit zugehörigen Zusicherungen können in Form benannter *SQL-Domänen* im aktuellen Schema definiert werden:

```
create domain Schulnote integer
  constraint NoteDefiniert check(value is not null)
  constraint NoteZwischen1und6 check(value in(1,2,3,4,5,6));
```

Integritätssicherung in SQL (2)

- **Tabellenzusicherungen** werden syntaktisch in Tabellendefinitionen geschachtelt. Sie garantieren, daß die Auswertung des Prädikats in jedem Datenbankzustand den Wert *true* liefert (universelle Quantifizierung).

```
create table Tabellenname (...
  constraint Zusicherungsname
    check (Prädikat))
```

- **Schemazusicherungen** sind SQL-Objekte, die dynamisch dem aktuellen SQL-Schema hinzugefügt werden können. Sie garantiert, daß in jedem Datenbankzustand die Auswertung des Prädikats den Wert *true* liefert.

```
create assertion Zusicherungsname
  check(Prädikat);
```

Ein Datenbankzustand heißt konsistent, wenn er alle im Schema deklarierten Zusicherungen erfüllt. Logisch gesehen sind alle Tabellen- und Schemazusicherungen konjunktiv verknüpft.

Spaltenwertintegrität

Eine Tabellenzusicherung, deren Prädikat sich nur auf einen Spaltennamen bezieht, garantiert die Spaltenintegrität und wird in folgenden Modellierungssituationen eingesetzt:

- Vermeidung von Nullwerten
- Definition von Unterbereichstypen
- Definition von Formatinformationen durch Stringvergleiche
- Definition von Aufzählungstypen

```
check(Alter is not null)
```

```
check(Alter >=0 and Alter <=150)
```

```
check(Postleitzahl like 'D-_____')
```

```
check(Note in (1,2,3,4,5,6))
```

Reihenintegrität

Eine Tabellenzusicherung, deren Prädikat sich auf mehrere Spaltennamen bezieht, definiert eine Reihenintegritätsbeziehung, die von jeder Reihe einer Tabelle erfüllt sein muß.

```
check(Ausgaben <= Einnahmen)

check((HatVordiplom, HatDiplom) in values(
  ('nein', 'nein')
  ('ja',   'nein')
  ('ja',   'ja')))
```

Tabellenintegrität (1)

Die Überprüfung quantifizierter Prädikate kann im Gegensatz zu den bisher besprochenen Zusicherungen im schlimmsten Fall die Auswertung einer kompletten mengenorientierten Anfrage zur Folge haben:

```
check((select sum(Budget) from Projekte) >= 0)

check(exists(select * from Abteilung
             where Oberabt = 'LTSW'))
```

Einige in der Praxis häufig vorkommende quantifizierte Zusicherungen können mittels *Indexstrukturen* (z.B. B-Bäume, Hash-Tabelle) effizient überprüft werden und sogar zu einem *Effizienzgewinn* bei Anfragen und Änderungsoperationen führen.

Tabellenintegrität (2)

Für häufig auftretende Muster von quantifizierten Zusicherungen bietet SQL syntaktische Konstrukte an, was die *Lesbarkeit* erhöht und *optimierende Implementierung* ermöglicht:

1. **Spaltenwerteindeutigkeit**: Die Eindeutigkeit von Spaltenwertkombinationen in einer Tabelle gestattet eine wertbasierte Identifikation von Tabellenelementen (→ *Schlüsselkandidat*).

   Beispiel zweier semantisch äquivalenter Zusicherungen:

```
create table Projekte(...
  unique(Name))
```

```
create table Projekte(...
  check(all x, all y: ...
  (
    (x.Name <> y.Name or x = y)
  )
```

(x.Name = y.Name) → (x = y)

Eine Tabelle kann mehrere Schlüsselkandidaten besitzen, die durch separate unique-Klauseln beschrieben werden.

Tabellenintegrität (3)

2. **Primärschlüsselintegrität**: Ein Schlüsselkandidat, in dessen Spalten keine Nullwerte auftreten dürfen, kann als Primärschlüssel ausgezeichnet werden. Eine Tabelle kann nur einen Primärschlüssel besitzen.

Beispiel zweier semantisch äquivalenter Zusicherungen:

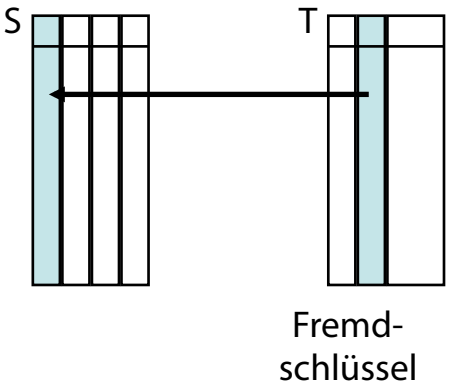
```
create table Projekte (...
  primary key(Nr))
```

```
create table Projekte(...
  unique Nr
  check(Nr is not null))
```

3. **Referentielle Integrität** (Fremdschlüsselintegrität): Diese Zusicherung bezieht sich auf den Zustand zweier Tabellen (s. nächste Folien)

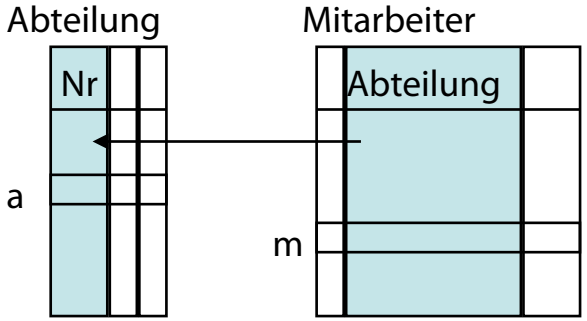
Referentielle Integrität (1)

Referentielle Integrität ist eine Zusicherung über den Zustand zweier Tabellen, die dann erfüllt ist, wenn zu jeder Reihe in Tabelle *T* eine zugehörige Reihe in Tabelle *S* existiert, die den Fremdschlüsselwert von *T* als Wert ihres Schlüsselkandidaten besitzt.



Beispiel zweier semantisch äquivalenter Zusicherungen:

```
create table Mitarbeiter
  (...
   constraint MitarbeiterHatAbteilung
   foreign key(Abteilung)
     references Abteilung(Nr))
```




```
create assertion MitarbeiterHatAbteilung
  check(not exists(select * from Mitarbeiter m where
       not exists(select * from Abteilung a where m.Abteilung = a.Nr)))
```

$\forall$ m $\in$ Mitarbeiter : $\exists$ a $\in$ Abteilung: m.Abteilung=a.Nr

Referentielle Integrität (2)

Im allgemeinen besteht ein Fremdschlüssel einer Tabelle `T` aus einer Liste von Spalten, der eine typkompatible Liste von Spalten in `S` entspricht:

```
create table T
  (...
   constraint Name
     foreign key(A1, A2, ..., An) references (S(B1, B2, ..., Bn))
```

Sind $B_1, B_2, \ldots, B_n$ die Primärschlüsselspalten von `S`, kann ihre Angabe entfallen.

Beachte: Rekursive Beziehungen (z.B. Abteilung : Oberabteilung) führen zu reflexiven Fremdschlüsseldeklarationen (`S` = `T`).

Behandlung von Integritätsverletzungen (1)

- Ohne spezielle Maßnahmen wird eine SQL-Anweisung, die eine Zusicherung verletzt, vom DBMS ignoriert. Eine Statusvariable signalisiert, welche Zusicherung verletzt wurde.
- Bei der commit-Anweisung wird im Falle einer gescheiterten verzögerten Integritätsbedingung ein Transaktionsabbruch (**`rollback`**) ausgelöst. Es kann daher sinnvoll sein, vor dem Transaktionsende mit der Anweisung **`set constraints all immediate`** eine unmittelbare Überprüfung aller verzögerbaren Zusicherungen zu erzwingen und Integritätsbedingungen explizit programmgesteuert zu behandeln.
- Fremdschlüsselintegrität zwischen zwei Tabellen *S* und *T* kann durch vier Operationen verletzt werden:
  1. **`insert into`** `T`
  2. **`update`** `T` **`set ...`**
  3. **`delete from`** `S`
  4. **`update`** `S` **`set ...`**

Behandlung von Integritätsverletzungen (2)

- Im Fall 1 und 2 führt der Versuch in *T* einen Fremdschlüsselwert einzufügen, der nicht in *S* definiert ist dazu, daß die Anweisung ignoriert wird. Die Verletzung wird über eine Statusvariable oder eine Fehlermeldung angezeigt.
- Wird im Falle 3 oder 4 versucht, eine Reihe zu löschen, deren Schlüsselwert noch als Fremdschlüsselwert in einer oder mehrerer Reihen der Tabelle *T* auftritt, wird eine der folgenden Aktion ausgeführt, die am Ende der references-Klausel spezifiziert werden kann; folgende Aktionsalternativen sind möglich:
  - **`set null`**: Der Fremdschlüsselwert aller betroffener Reihen von *T* wird durch **`null`** erstetzt.
  - **`set default`**: Der Fremdschlüsselwert aller betroffener Reihen von *T* wird durch den Standardwert der Fremdschlüsselspalte ersetzt.
  - **`cascade`**: Im Fall 3 (**`delete`**) werden alle betroffenen Reihen von *T* gelöscht. Im Falle 4 (**`update`**) werden die Fremdschlüsselwerte aller betroffener Reihen von *T* durch die neuen Schlüsselwerte der korrespondierenden Reihen ersetzt.
  - **`no action`**: Es wird keine Folgeaktion ausgelöst, die Anweisung wird ignoriert.

Zeitpunkt der Integritätsprüfung

Bezieht sich eine Zusicherung auf mehrere Zustandsvariablen, muß der Zeitpunkt der Integritätsprüfung nach Änderungsoperationen genau spezifiziert werden.

Dazu existieren zwei Modi der Integritätsprüfung:

- **`not deferrable`** kennzeichnet eine nicht verzögerbare Zusicherung, die unmittelbar nach jeder SQL-Anweisung überprüft wird.
- **`deferrable`** kennzeichnet eine verzögerbare Zusicherung. Man kann ein Flag auf den Wert
  - **`immediate`** setzen, wenn nach der nächsten SQL-Anweisung geprüft werden soll oder auf den Wert
  - **`deferred`**, wenn die Prüfung bis zum Transaktionsende aufgeschoben werden soll.
  - Zusätzlich wird bei jedem Umschalten auf den Wert **`immediate`** und am Transaktionsende überprüft.

Optimierungen können den tatsächlichen Zeitpunkt beeinflussen.

Zusammenfassung, Kernpunkte

Grundlagen von Datenbanksystemen

- Anfragesprache SQL