



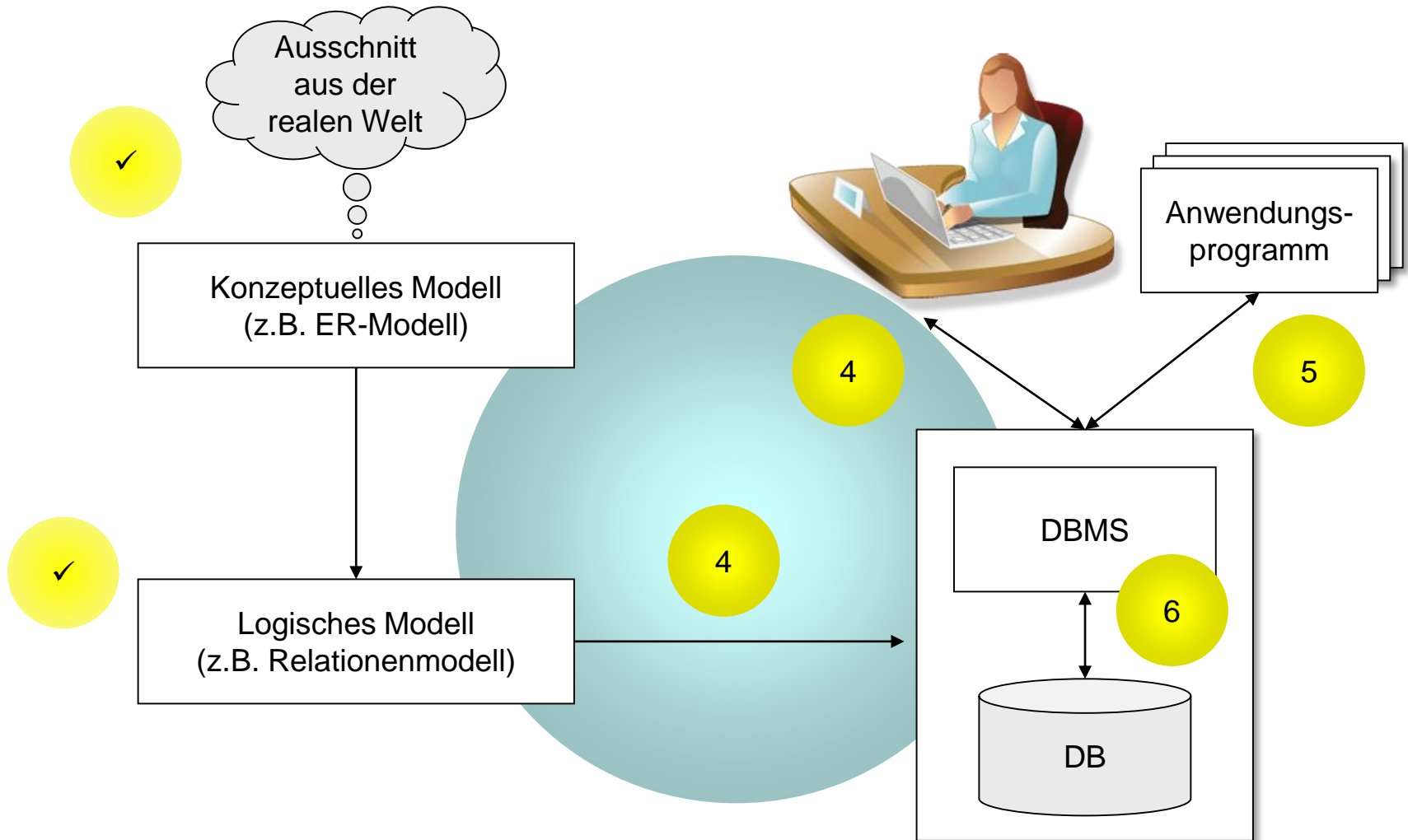
Datenbanken 1

– Kapitel 4: Datendefinition und -manipulation –

```
SELECT knr, count(*)  
FROM bestellung  
WHERE datum BETWEEN '01.01.2014' and '31.12.2014'  
GROUP BY knr  
HAVING count(*) > 1;
```



Vorlesung Datenbanken 1





Datendefinition und -manipulation

Inhalt des Kapitels

- Datendefinition
 - SQL DDL
- Datenmanipulation und –abfragen
 - Grundlagen von Anfragen
 - Relationenalgebra
 - SQL DML
 - Andere Datenbankabfragesprachen

Lernziele

- Kennenlernen der Möglichkeiten zur Datendefinition, -manipulation und -abfrage mit SQL
- Verstehen und Anwenden der Relationenalgebra
- Beherrschen und Anwenden der wichtigsten SQL-Kommandos



Structured Query Language (SQL)

Ziel: Entwicklung einer Datenbanksprache für alle Aufgaben der Datenbank-Verwaltung

Entwicklung

- 1974 viele Sprachentwürfe, u.a. SEQUEL (Structured English Query Language)
→ Weiterentwicklung zu SQL (Structured Query Language)
- 1975/76 System R (IBM) als Prototyp für SQL
- 1986/87 SQL als ANSI/ISO-Standard
- 1989 SQL-89
- 1992 SQL-92 mit 3 Leveln (Entry Level, Intermediate Level, Full Level)
- 1999 SQL:1999 u.a. objekt-relationale Erweiterungen
- ...
- 2006 SQL:2006 u.a. (weitere) XML-Erweiterungen (Integration XQuery)
- ...
- 2011 SQL:2011 u.a. temporale Erweiterungen
 - SQL ist heute „de-facto“ Standard in der relationalen Welt
 - Produkte unterstützen meist (nur) SQL-92/Intermediate Level und Teile von SQL:1999 / SQL:20**



SQL – Grundlagen

Was wird innerhalb einer Datenbanksprache benötigt?

- Definition der Struktur einer Datenbank:
DDL - Data Definition Language
- Lesen der Daten:
Data Query Language
- Verändern der Daten:
DML - Data Manipulation Language
- Kommandos zur Administration der Daten (Sichern, Reorganisieren etc.): diese umfasst SQL (leider!) nicht

Wichtige Anmerkung:

- In der Vorlesung werden weder alle SQL-Kommandos vorgestellt, noch die vorgestellten notwendigerweise vollständig behandelt; sondern jeweils nur die wichtigsten Features.



SQL – DDL

- Zur Verwaltung von Strukturelementen einer Datenbank stehen in der DDL von SQL die folgenden Operatoren zur Verfügung:

CREATE	Strukturelement	(* Anlegen eines Strukturelements *)
ALTER	Strukturelement	(* Ändern eines Strukturelements *)
DROP	Strukturelement	(* Löschen eines Strukturelements *)



Systemkatalog

- Strukturinformation einer relationalen Datenbank wird im sog. **Systemkatalog** (Datenbankkatalog, *Data Dictionary*, *Information Schema*) verwaltet
- Der Systemkatalog selbst hat wieder die Struktur einer relationalen Datenbank, z.B.
 - Systemtabelle, in welcher die Informationen enthalten ist, welche Tabellen die Datenbank hat, Systemtabelle mit Informationen zu allen Spalten etc.
 - TABLES
 - COLUMNS
 - DOMAINS
 - CHECK_CONSTRAINTS
 - ...
- *Information Schema*: Standardisiert seit SQL:92;
im SQL:2003-Standard: 132(!) Tabellen bzw. Views



Definition von Tabellen

- Relationenschemata werden in Form von Tabellen abgebildet:

```
CREATE TABLE tabellen-name (  
    spalten-name1 wertebereich1 [spalten-constraint-def1],  
    ... ,  
    spalten-namen wertebereichn [spalten-constraint-defn],  
);
```

- Wertebereich kann entweder ein vordefinierter atomarerer Datentyp oder eine benutzerdefinierte Einschränkung auf einem vordefinierten Datentyp sein (DOMAIN – hier nicht behandelt)

- Beispiel

```
CREATE TABLE Mitarbeiter (  
    Personalnummer integer,  
    Name varchar(20),  
    Geburtsdatum date  
);
```




Datentypen in SQL

- **integer** (oder auch **integer4**, **int**),
- **smallint** (oder auch **integer2**),
- **float**(p) (oder auch kurz **float**),
- **decimal**(p,q) und **numeric**(p,q) mit jeweils q Nachkommastellen,
- **character**(n) (oder kurz **char**(n), bei $n = 1$ auch **char**) für Zeichenketten (Strings) fester Länge n ,
- **character varying**(n) (oder kurz **varchar**(n) für Strings variabler Länge bis zur Maximallänge n ,
- **bit**(n) oder **bit varying**(n) analog für Bitfolgen, und
- **date**, **time** bzw. **timestamp** für Datums-, Zeit- und kombinierte Datums-Zeit-Angaben
- **blob** (**binary large object**) für sehr große binäre Daten
- **clob** (**character large object**) für sehr große Strings

Achtung: Gelegentlich verwenden Hersteller andere Bezeichnungen (beispielsweise bei Neuimplementierung von Datentypen – Oracle beispielsweise **varchar2**(n))



Integritätsbedingungen (constraints) – 1(4)

- Statische Integritätsbedingungen auf Tabellen (d.h. bezüglich Attributen oder Attributkombinationen):
 - Verbot von Nullwerte (NOT NULL)
 - Default-Werte (DEFAULT)
 - Eindeutigkeit (UNIQUE bzw. PRIMARY KEY)
 - Fremdschlüssel (FOREIGN KEY)
 - CHECK-Bedingungen
- Fall 1: Constraint bezieht sich nur auf ein Attribut
 - Angabe direkt hinter der Spaltendefinition oder
 - Angabe nach den Spaltendefinitionen
- Fall 2: Constraint bezieht sich auf mehr als ein Attribut
 - Angabe nach den Spaltendefinitionen - dabei müssen die Attribute, auf welcher sich der Constraint bezieht, mit angegeben werden.



Integritätsbedingungen (*constraints*) – 2(4)

```
spalten-constraint-def ::=  
[ CONSTRAINT constraint-name ]  
    [ DEFAULT default-wert | NOT NULL ]  
    [ PRIMARY KEY | UNIQUE ]  
    [ references-def ]  
    [ CHECK (cond-def) ]
```

```
CREATE TABLE Mitarbeiter (  
    Personalnummer integer          PRIMARY KEY,  
    Name           varchar(20)      NOT NULL,  
    Geburtsdatum   date,  
    Gehalt         decimal(8,2)      DEFAULT 0,00 CHECK (Gehalt < 190.000,00)  
);
```

```
CREATE TABLE Buch_Versionen (  
    ISBN           char(10),  
    Auflage        smallint CHECK (Auflage > 0),  
    Jahr           integer  CHECK (Jahr BETWEEN 1800 AND 2060),  
    PRIMARY KEY (ISBN, Auflage)  
);
```



Integritätsbedingungen (*constraints*) – 3(4)

Bemerkung zu NULL-Werten

- NOT NULL schließt in den entsprechenden Spalten *Nullwerte* als Attributwerte aus
- NULL repräsentiert die Bedeutung „Wert unbekannt“, „Wert nicht anwendbar“ oder „Wert existiert nicht“, gehört aber zu keinem Wertebereich, insbesondere gilt: $\text{NULL} \neq \text{NULL}$
- NULL kann in allen Spalten auftauchen, außer in Schlüsselattributen und den mit NOT NULL gekennzeichneten



Integritätsbedingungen (*constraints*) – 4(4)

Definition von Fremdschlüsseln

references-def ::=

```
FOREIGN KEY (spalten-name1, ..., spalten-namem)  
  REFERENCES tabellen-name [ (spalten-name1, ..., spalten-namem) ]  
  [ON DELETE ref-action]  
  [ON UPDATE ref-action]
```

ref-action ::= **NO ACTION** | **CASCADE** | **SET DEFAULT** | **SET NULL** | **RESTRICT**

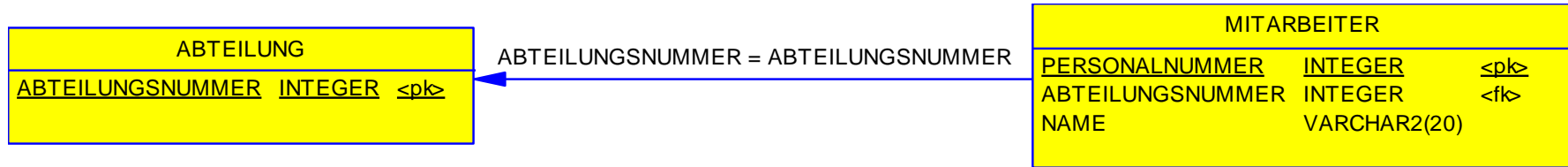
- Referentielle Aktionen (ref-action) werden später erläutert
- Beispiel:

```
CREATE TABLE Mitarbeiter (  
  Personalnummer integer           PRIMARY KEY,  
  Name           varchar(20)       NOT NULL,  
  ...  
  AbtNr          integer           NOT NULL,  
  FOREIGN KEY (AbtNr) REFERENCES Abteilung (Abteilungsnummer)  
);
```



Referentielle Integrität bei Änderungen

- Was ist das Problem?



- Verhalten bei Veränderung bzw. Löschen des referenzierten Primärschlüssels (z.B. Abteilung wird umbenannt oder Abteilung wird aufgelöst)
- Varianten?



Referentielle Integrität bei Änderungen

references-def ::=

```
FOREIGN KEY (spalten-nameh, ..., spalten-namem)  
  REFERENCES tabellen-name [ (spalten-namei, ..., spalten-namen) ]  
  [ON DELETE ref-action]  
  [ON UPDATE ref-action]
```

ref-action ::= **NO ACTION** | **CASCADE** | **SET DEFAULT** | **SET NULL** | **RESTRICT**

CASCADE	Operation „kaskadiert“ zu allen zugehörigen Datensätzen
SET DEFAULT	Fremdschlüssel wird in allen zugehörigen Sätzen auf benutzerdefinierten Default-Wert gesetzt
SET NULL	Fremdschlüssel wird in allen zugehörigen Sätzen auf „NULL“ gesetzt
RESTRICT	Operation wird nur ausgeführt, wenn keine zugehörigen Sätze (Fremdschlüssel) vorhanden sind
NO ACTION	(im Prinzip) wie RESTRICT = default-Einstellung



Management von *Constraints*

Constraints können (sollten!) einen (benutzerdefinierten) Namen erhalten

⇒ können über diesen Namen identifiziert werden und später gelöscht / geändert werden

```
CREATE TABLE Buch_Versionen (  
  ISBN          char(10),  
  Auflage       smallint CONSTRAINT C_Auflage CHECK (Auflage > 0),  
  Jahr         integer CONSTRAINT C_Jahr  
                                CHECK (Jahr BETWEEN 1800 AND 2060),  
  CONSTRAINT PK_BUCH_Versionen PRIMARY KEY (ISBN, Auflage)  
);
```




Ändern von Strukturinformation – 1(2)

Relationenschemata können (in gewissem Maße) geändert werden

- Hinzufügen von Spalten

```
ALTER TABLE tabellen-name ADD spalten-name wertebereich  
    [ DEFAULT default-wert ]  
    [ CHECK (cond-def) ];
```

→ Anpassung im Systemkatalog und

→ in der Tabelle, d.h. jedem Tupel wird ein neues Attribut
(mit NULL oder dem Default-Wert belegt) hinzugefügt

```
ALTER TABLE Mitarbeiter ADD Kinder integer;
```

```
ALTER TABLE Mitarbeiter ADD Bonus integer DEFAULT 1000;
```

- Löschen von Spalten

```
ALTER TABLE tabellen-name DROP spalten-name  
    { RESTRICT | CASCADE };
```



Ändern von Strukturinformation – 2(2)

- Hinzufügen/Löschen von Constraints zu einer Tabelle

```
ALTER TABLE tabellen-name ADD constraint-def;
```

```
ALTER TABLE tabellen-name DROP CONSTRAINT constraint-name;
```

```
ALTER TABLE Buch_Versionen DROP CONSTRAINT C_Jahr;
```

```
ALTER TABLE Buch_Versionen ADD  
    CONSTRAINT C_Jahr CHECK (Jahr BETWEEN 1800 AND 2090);
```

Ein Grund,
warum Namen
für Constraints
sinnvoll sind!

- Löschen einer ganzen Tabelle

```
DROP TABLE tabellen-name { RESTRICT | CASCADE };
```



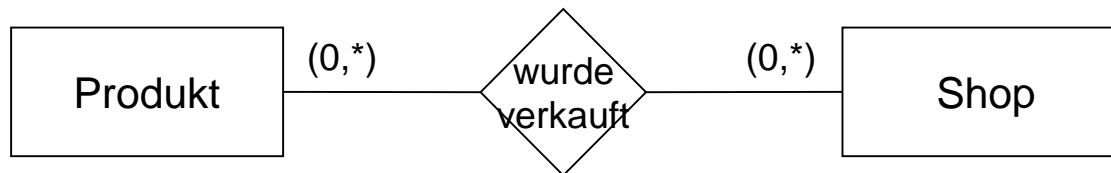
Systemkatalog: Oracle

- Systemtabellen sind bei Oracle nach folgendem Schema aufgebaut:
<Präfix>_<Schemaobjekt>
- Das Präfix schränkt die anzuzeigenden Objekte ein:
 - USER_ alle Schemaobjekte des aktuellen Users, also des entsprechenden Schemas
 - ALL_ alle Schemaobjekte, auf die der User Zugriffsrechte hat
 - DBA_ alle Schemaobjekte; nur ein Datenbankadministrator (User DBA) darf hierauf zugreifen
- Auswahl wichtiger Schemaobjekte:
 - TABLES, TAB_COLUMNS, CONSTRAINTS, INDEXES, TABLESPACES, TRIGGERS, VIEWS, ...



Übung

- Überführen Sie das folgende E/R-Modell in ein relationales Modell und geben Sie die entsprechenden SQL-Statements zur Tabellendefinitionen an:



Produkt (ProdNr, Bezeichnung, Preis)

Shop (ShopID, Name)

wurde verkauft (Anzahl)



Datendefinition und -manipulation

- ✓ Datendefinition
 - ✓ SQL DDL
- Datenmanipulation und –abfragen
 - Grundlagen von Anfragen
 - Relationenalgebra
 - SQL DML
 - Andere Datenbankabfragesprachen



Kriterien für Anfragesprachen (Auswahl)

- **Ad-Hoc-Formulierung:** Benutzer soll eine Anfrage formulieren können, ohne ein vollständiges Programm schreiben zu müssen
- **Deskriptivität:** Benutzer soll formulieren „Was will ich haben?“ und nicht „Wie komme ich an das, was ich haben will?“
- **Mengenorientiertheit:** jede Operation soll auf Mengen von Daten gleichzeitig arbeiten, nicht navigierend nur auf einzelnen Elementen („one-tuple-at-a-time“)
- **Abgeschlossenheit:** Ergebnis ist wieder eine Relation und kann wieder als Eingabe für die nächste Anfrage verwendet werden
- **Orthogonalität:** Sprachkonstrukte sind in ähnlichen Situationen auch ähnlich anwendbar
- **Optimierbarkeit:** Sprache besteht aus wenigen Operationen, für die es Optimierungsregeln gibt
- **Sicherheit:** keine Anfrage, die syntaktisch korrekt ist, darf in eine Endlosschleife geraten oder ein unendliches Ergebnis liefern



Relationenalgebra – Grundlagen

- Basisoperationen auf Tabellen, die die Berechnung von neuen Ergebnistabellen aus gespeicherten Datenbanktabellen erlauben
 - Operationen werden zur so genannten *Relationenalgebra* zusammengefasst (Mathematik: Algebra ist definiert durch Wertebereich sowie darauf definierten Operationen)
- ⇒ für Datenbankabfragen entsprechen die Inhalte der Datenbank den Werten, Operationen sind dagegen Funktionen zum Berechnen der Abfrageergebnisse
- ⇒ Abfrageoperationen sind beliebig kombinierbar und bilden eine Algebra zum „Rechnen mit Tabellen“ – die so genannte relationale Algebra oder auch Relationenalgebra
- ⇒ SQL ist „lediglich“ eine Sprache, welche die Relationenalgebra praktisch umsetzt



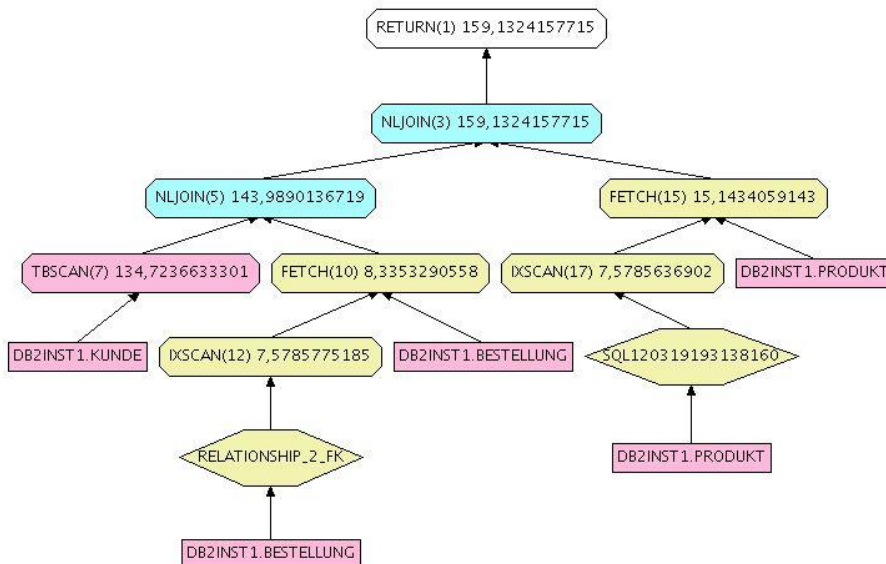
Relationenalgebra – Anwendung

- Die Relationenalgebra wird intern vom Datenbankmanagementsystem verwendet, um die SQL-Anfragen zu optimieren und auszuführen

- Beispiel:

```
SELECT produktname
FROM   Produkt, Bestellung, Kunde
WHERE  Produkt.pid = Bestellung.pid AND
       Bestellung.knr = Kunde.knr AND
       Kunde.kname = 'Meier'
```

⇒ Optimierter
Anfragebaum (DB2)





Relationenalgebra – Operationen

- **Spalten ausblenden:** Projektion π
- **Zeilen heraussuchen:** Selektion σ
- **Tabellen verknüpfen:** Verbund (Join) $\triangleright \triangleleft$
- **Tabellen vereinigen:** Vereinigung \cup
- **Tabellen voneinander abziehen:** Differenz $-$
- **Spalten umbenennen:** Umbenennung β
(wichtig für $\triangleright \triangleleft$, \cup , $-$)



Selektion σ

PERS:	<u>PNR</u>	NAME	ALTER	GEHALT	ANR	MNR
	406	Coy	47	50700	K55	\perp
	123	Müller	32	43500	K51	187
	829	Schmid	36	45200	K53	177
	574	Abel	28	36000	K55	406

- *Selektion*: Auswahl von Zeilen einer Tabelle anhand eines Selektionsprädikats
- Syntax: $\sigma_{\text{Bedingung}}(\text{Relation})$
- Beispiel: $\sigma_{\text{ANR}='K55'}(\text{PERS})$

ERGEBNIS:	PNR	NAME	ALTER	GEHALT	ANR	MNR
	406	Coy	47	50700	K55	\perp
	574	Abel	28	36000	K55	406



Selektion σ

- *Definition:* $\sigma_P(R) := \{ t \mid t \in R \wedge P(t) \}$
- *Konstantenselektion:* Attribut Θ Konstante
 - boolesches Prädikat Θ ist = oder \neq , bei linear geordneten Wertebereichen auch \leq , $<$, \geq oder $>$
- *Attributselektion:* Attribut₁ Θ Attribut₂
- logische Verknüpfung mehrerer Konstanten- oder Attribut-Selektionen mit \wedge , \vee oder \neg
- Beispiel: $\sigma_{ANR='K55' \wedge GEHALT > 50000}(PERS)$
- Kommutativität: $\sigma_P(\sigma_Q(R)) =$



Projektion π

- *Projektion*: Auswahl von Spalten durch Angabe einer Attributliste

- Syntax: $\pi_{\text{Attributmenge}}(\text{Relation})$

- Definition: $\pi_A(R) := \{ t(A) \mid t \in R \}$ mit Attributmenge $A \subseteq R$
- Achtung: Die Projektion entfernt Duplikate (Mengensemantik)

- Beispiele: $\pi_{PNR, ANR}(PERS)$

PNR	ANR
406	K55
123	K51
829	K53
574	K55

- $\pi_{ANR}(PERS)$

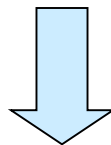
ANR
K55
K51
K53



Kombination der Operatoren

- $\pi_A(\sigma_P(R)) = \sigma_P(\pi_A(R))$
falls die im Prädikat P betrachteten Attribute eine Teilmenge von A sind
- Beispiel: $\pi_{NAME, ANR}(\sigma_{ANR='K55'}(PERS)) = \sigma_{ANR='K55'}(\pi_{NAME, ANR}(PERS))$

PERS:	<u>PNR</u>	NAME	ALTER	GEHALT	ANR	MNR
	406	Coy	47	50700	K55	⊥
	123	Müller	32	43500	K51	187
	829	Schmid	36	45200	K53	177
	574	Abel	28	36000	K55	406





Übung

PERS:	<u>PNR</u>	NAME	ALTER	GEHALT	ANR	MNR
	406	Coy	47	50700	K55	⊥
	123	Müller	32	43500	K51	187
	829	Schmid	36	45200	K53	177
	574	Abel	28	36000	K55	406

ABT:	<u>ANR</u>	ANAME	AORT
	K51	Planung	Darmstadt
	K53	Einkauf	Frankfurt
	K55	Vertrieb	Frankfurt
	K56	Finanzen	München

- Finde alle Abteilungsorte.
- Finde alle Angestellten (PNR, NAME), deren Manager die Personalnummer 406 hat (MNR).
- Finde alle Angestellten (PNR, ALTER, NAME), die in einer Abteilung in Frankfurt arbeiten und zwischen 30 und 36 Jahren alt sind.



Kartesisches Produkt

- $K = R \times S := \{ k \mid \exists x \in R, y \in S: (k = \langle x_1, x_2, \dots, y_1, y_2, \dots, y_s \rangle) \}$

ABTxPERS:	ANR	ANAME	AORT	PNR	NAME	ALTER	GEHALT	ANR'	MNR
	K51	Planung	Darmstadt	406	Coy	47	50700	K55	⊥
	K51	Planung	Darmstadt	123	Müller	32	43500	K51	187
	K51	Planung	Darmstadt	829	Schmid	36	45200	K53	177
	K51	Planung	Darmstadt	574	Abel	28	36000	K55	406
	K53	Einkauf	Frankfurt	406	Coy	47	50700	K55	⊥
	K53	Einkauf	Frankfurt	123	Müller	32	43500	K51	187
	K53	Einkauf	Frankfurt	829	Schmid	36	45200	K53	177
	K53	Einkauf	Frankfurt	574	Abel	28	36000	K55	406
	K55	Vertrieb	Frankfurt	406	Coy	47	50700	K55	⊥
	K55	Vertrieb	Frankfurt	123	Müller	32	43500	K51	187



Verbund (*Join*, Θ -*Join*)

- Kartesisches Produkt zwischen zwei Relationen R und S eingeschränkt durch Θ -Beziehung zwischen Attribut A von R und Attribut B von S mit $\Theta \in \{<, =, >, \leq, \neq, \geq\}$
- $R \bowtie_{A\Theta B} S = \sigma_{A\Theta B}(R \times S)$
- Wichtigster Spezialfall: $\Theta = '=' (Gleichverbund)$



Gleichverbund (*Equi Join*)

- Beispiel: $ABT \bowtie PERS$
ANR=ANR

ABTxPERS:	ANR	ANAME	AORT	PNR	NAME	ALTER	GEHALT	ANR'	MNR
	K51	Planung	Darmstadt	123	Müller	32	43500	K51	187
	K53	Einkauf	Frankfurt	829	Schmid	36	45200	K53	177
	K55	Vertrieb	Frankfurt	406	Coy	47	50700	K55	⊥
	K55	Vertrieb	Frankfurt	574	Abel	28	36000	K55	406

- **Verlustbehafteter Gleichverbund**
 - wenn Tupel in ABT oder PERS keinen Verbundpartner finden (*dangling tuples*), z.B. (K56, Finanzen, München)
 - ⇒ π als Umkehroperation führt in diesem Fall nicht wieder zu den Ausgangsrelationen
- **Verlustfreier Gleichverbund (*lossless join*)**
 - Ein Gleichverbund zwischen R und S heisst verlustfrei, wenn alle Tupel von R und S am Verbund teilnehmen.
 - Die inverse Operation π erzeugt dann wieder R und S



Übung (Fortsetzung)

PERS:	<u>PNR</u>	NAME	ALTER	GEHALT	ANR	MNR
	406	Coy	47	50700	K55	⊥
	123	Müller	32	43500	K51	187
	829	Schmid	36	45200	K53	177
	574	Abel	28	36000	K55	406

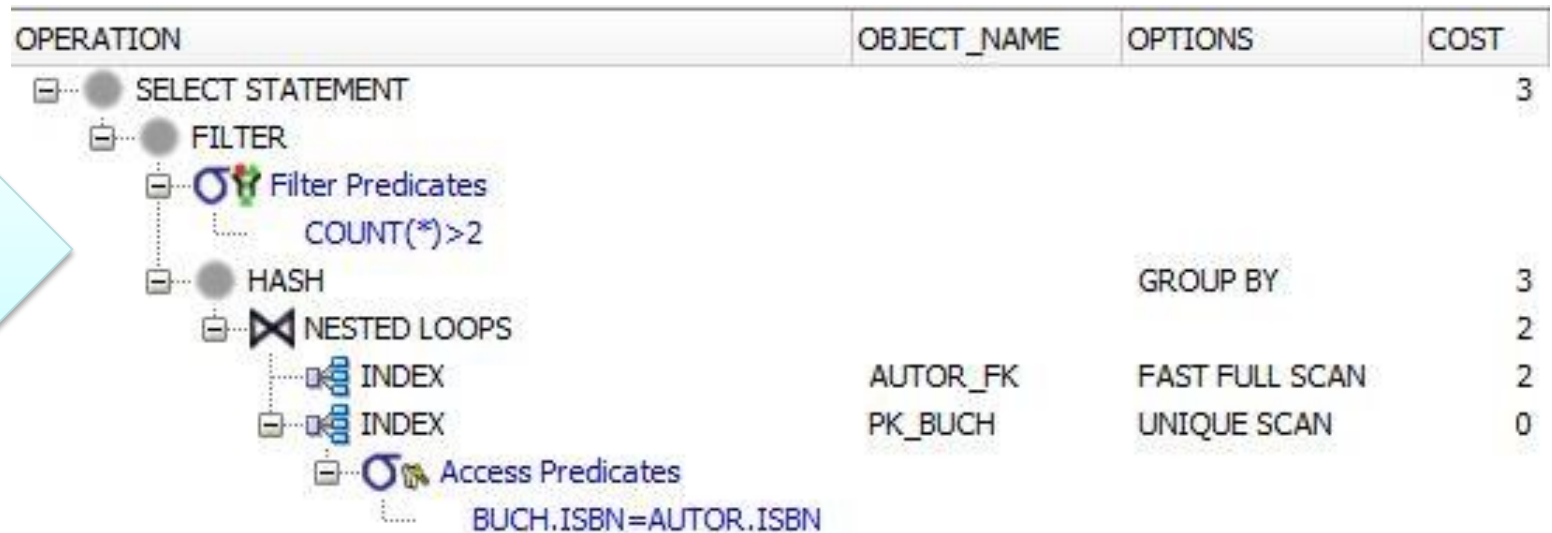
ABT:	<u>ANR</u>	ANAME	AORT
	K51	Planung	Darmstadt
	K53	Einkauf	Frankfurt
	K55	Vertrieb	Frankfurt
	K56	Finanzen	München

- Finde alle Angestellten (PNR, ALTER, NAME), die in einer Abteilung in Frankfurt arbeiten und zwischen 30 und 36 Jahren alt sind.



Relationenalgebra – Anwendung

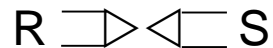
```
SELECT buch.isbn, count(*)
FROM   buch, autor
WHERE  buch.ISBN = autor.ISBN
GROUP BY buch.isbn
HAVING COUNT(*) > 2;
```





Verbundvarianten

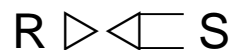
- Ziel: Verlustfreier Verbund soll erzwungen werden
- Übernahme von „*dangling tuples*“ in das Ergebnis und Auffüllen mit Nullwerten
- **voller äußerer Verbund** (*full outer join*) übernimmt alle Tupel beider Operanden



- **linker äußerer Verbund** (*left outer join*) übernimmt alle Tupel des linken Operanden



- **rechter äußerer Verbund** (*right outer join*) übernimmt alle Tupel des rechten Operanden






Verbundvarianten – Beispiele

Firma

<u>Name</u>	LC
BMW	DE
VW	DE
Jaguar	UK

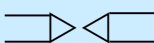
Zulieferer

<u>LName</u>	LC
Michelin	FR
Continental	DE



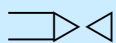
 LC=LC

Name	LC	LName
BMW	DE	Continental
VW	DE	Continental



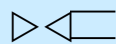
 LC=LC

Name	LC	LName
⊥	FR	Michelin
BMW	DE	Continental
VW	DE	Continental
Jaguar	UK	⊥



 LC=LC

Name	LC	LName
BMW	DE	Continental
VW	DE	Continental
Jaguar	UK	⊥



 LC=LC

Name	LC	LName
⊥	FR	Michelin
BMW	DE	Continental
VW	DE	Continental



Umbenennung β

- Anpassen von Attributnamen mittels Umbenennung:
Syntax: $\beta_{neu \leftarrow alt}(Relation)$

- Beispiel:

KUNDE:	VORNAME	NAME
	Ralf	Schaarschmidt
	Jens	Lufter

INTERESSENT:	VORNAME	NACHNAME
	Jens	Lufter
	Jan	Nowitzky
	Christoph	Gollmick

Angleichen durch $\beta_{Name \leftarrow Nachname}(Interessent)$

- durch Umbenennung nun möglich:
 - Mengenoperationen
 - (Natural Join statt Equi Join)



Mengenoperationen: Vereinigung

- Vereinigung: $R \cup S := \{ t \mid t \in R \vee t \in S \}$
- Vereinigung sammelt die Tupelmengen zweier Relationen unter einem gemeinsamen Schema auf
- Attributmengen beider Relationen müssen identisch (bzw. vereinigungsverträglich) sein – gilt für alle nachfolgenden Mengenoperationen
- Beispiel:
alle Kunden und Interessenten: $Kunde \cup \beta_{Name \leftarrow Nachname}(Interessent)$

ERGEBNIS:	VORNAME	NAME
	Ralf	Schaarschmidt
	Jens	Luffer
	Jan	Nowitzky
	Christoph	Gollmick



Mengenoperationen: Differenz

- Differenz: $R - S := \{ t \mid t \in R \wedge t \notin S \}$
- Differenz eliminiert die Tupel aus der ersten Relation, die auch in der zweiten Relation vorkommen
- Beispiel:
Kunden, die keine Interessenten sind:
 $Kunde - \beta_{Name \leftarrow Nachname}(Interessent)$

ERGEBNIS:	VORNAME	NAME
	Ralf	Schaarschmidt



Mengenoperationen: Durchschnitt

- *Durchschnitt:* $R \cap S := \{ t \mid t \in R \wedge t \in S \}$
- Durchschnitt ergibt die Tupel, die in beiden Relationen gemeinsam vorkommen
- Beispiel:
alle Kunden, die auch Interessenten sind:
 $Kunde \cap \beta_{Name \leftarrow Nachname}(Interessent)$

ERGEBNIS:	VORNAME	NAME
	Jens	Lufner

- Bemerkung: Durchschnittsoperation ist eigentlich überflüssig, da:
 $R \cap S = R - (R - S)$



Relationenalgebra

Zusammenfassung

- Formales Modell für Anfragen in relationalen Datenbanksystemen
- Operationaler Ansatz
- Anfrage als Schachtelung von Operatoren auf Relationen
- Relationenoperationen auf einer Relation
 - Projektion
 - Selektion
- Relationenoperationen auf zwei Relationen
 - Kartesisches Produkt
 - Unterschiedliche Verbundoperationen
- Klassische Mengenoperationen (auf gleich strukturierten Relationen)
 - Vereinigung, Differenz, Durchschnitt

- Anderer Ansatz: Relationenkalkül (logikbasierter Ansatz: Anfrage als abgeleitete Prädikate)



Datendefinition und -manipulation

- ✓ Datendefinition
 - ✓ SQL DDL

- ✓ Datenmanipulation und –abfragen
 - ✓ Grundlagen von Anfragen
 - ✓ Relationenalgebra
 - **SQL DML**
 - Elementare Datenmanipulation
 - SQL-Kern: SFW-Block
 - Erweiterungen des SFW-Blocks
 - Aggregatfunktionen und Gruppierungen
 - Veränderungen am Datenbestand

 - Andere Datenbankabfragesprachen



Beispiel-Tabellen

PRODUKT:	<u>PRODID</u>	BEZEICHNUNG	PREIS	BESTAND	HERSTID
	201	Skyscraper	99.0	12	901
	202	Himmelsstürmer	129.0	4	901
	203	Rainbow Hopper	45.0	20	902
	204	TumbleAround	21.0	30	901
	205	2Hi4U	129.0	1	902

KUNDE:	<u>KNR</u>	NAME	PLZ	ORT
	101	Schaarschmidt	60528	Frankfurt
	102	Lufter	65196	Wiesbaden
	103	Nowitzky	60431	Frankfurt
	104	Gollmick	69190	Walldorf

HERSTELLER:	<u>HERSTID</u>	NAME
	901	Flattermann GmbH
	902	Dragon.com
	903	KiteSports

BESTELLUNG:	<u>BID</u>	DATUM	KNR	PRODID
	1	01.03.2014	103	201
	2	11.03.2014	101	202
	3	05.05.2014	104	202
	4	10.12.2014	103	205
	5	03.01.2015	101	205



Einfügen von Tupeln: **insert**

Einfachste Form:

```
INSERT INTO relationen-name [ (attribut1, ..., attributn) ]  
VALUES (konstante1, ..., konstanten)
```

- Optionale Attribute erlaubt das Einfügen von unvollständigen Tupeln
⇒ Wert der nicht angegebenen Attribute werden NULL (falls für diese Attribute erlaubt!)

Beispiele:

```
INSERT INTO Produkt  
VALUES (206, 'AirCrusher', 69.0, 20, 902)
```

```
INSERT INTO Produkt (Prodid, HerstID, Bezeichnung)  
VALUES (206, 902, 'AirCrusher')
```

- Andere Formulierungsmöglichkeit?

Später: weitere Varianten zum Einfügen von Tupeln (nach Einführung SFW)



SQL-Kern (SFW-Block)

- **SELECT**
 - Projektionsliste
 - arithmetische Operationen und Aggregatfunktionen
- **FROM**
 - zu verwendende Relationen, evtl. Umbenennungen
- **WHERE**
 - Selektions- und Verbundbedingungen
 - Geschachtelte Anfragen (wieder ein SFW-Block)

```
SELECT [ ALL | DISTINCT ] select-item-commalist  
FROM table-ref-commalist  
[ WHERE cond-exp ]  
[ GROUP BY column-ref-commalist ]  
[ HAVING cond-exp ]  
[ ORDER BY order-item-commalist ]
```



Auswahl von Tabellen: die **from**-Klausel

- Einfachste Form:

```
SELECT *  
FROM Relationenname
```

- Beispiel:

```
SELECT *  
FROM Hersteller
```

- Es werden alle Tupel der angegebenen Relation ausgewählt



Kartesisches Produkt

- Bei Angabe mehr als einer Relation wird das kartesische Produkt gebildet:

```
SELECT *  
FROM Produkt, Hersteller
```

Bemerkungen

- alle möglichen Kombinationen werden ausgegeben!
- es können mehrere (beliebige viele Relationen) angegeben werden

PRODID	BEZEICHNUNG	PREIS	BESTAND	HERSTID	HERSTID	NAME
201	Skyscraper	99.0	12	901	901	Flattermann GmbH
201	Skyscraper	99.0	12	901	902	Dragon.com
201	Skyscraper	99.0	12	901	903	KiteSports
202	Himmelsstürmer	129.0	4	901	901	Flattermann GmbH
202	Himmelsstürmer	129.0	4	901	902	Dragon.com
202	Himmelsstürmer	129.0	4	901	903	KiteSports
...



Verbund

- frühe SQL-Versionen
 - üblicherweise realisierter Standard in aktuellen Systemen
 - kennen nur Kreuzprodukt, keinen expliziten Verbundoperator
 - Verbund durch Prädikat hinter **where** realisieren
- Beispiel für (natürlichen) Verbund:

```
SELECT *  
FROM Produkt, Hersteller  
WHERE Produkt.HerstID = Hersteller.HerstID
```

PRODID	BEZEICHNUNG	PREIS	BESTAND	HERSTID	HERSTID	NAME
201	Skyscraper	99.0	12	901	901	Flattermann GmbH
202	Himmelsstürmer	129.0	4	901	901	Flattermann GmbH
203	Rainbow Hopper	45.0	20	902	902	Dragon.com
204	TumbleAround	21.0	30	901	901	Flattermann GmbH
205	2Hi4U	129.0	1	902	902	Dragon.com
206	AirCrusher	69.0	20	902	902	Dragon.com



select-Klausel

- Festlegung der Projektionsattribute

```
SELECT [ DISTINCT ] { attribut |  
                        arithmetischer-ausdruck  
                        aggregat-funktion }+  
FROM ...
```

- Attribute der hinter **from** stehenden Relationen
(optional mit einem Präfix, der den Relationennamen oder den Namen der Tupelvariablen angibt)
- arithmetische Ausdrücke über Attributen dieser Relationen und passenden Konstanten
- Aggregatfunktionen über Attribute dieser Relationen



select-Klausel mit **distinct**-Operator

```
SELECT Ort  
FROM Kunde
```

- liefert die Ergebnisrelation als Multimenge:

ORT
Frankfurt
Wiesbaden
Frankfurt
Walldorf

```
SELECT DISTINCT Ort  
FROM Kunde
```

- ergibt Projektion aus der Relationenalgebra:

ORT
Frankfurt
Wiesbaden
Walldorf



Tupelvariablen und Relationennamen

- Anfrage

```
SELECT Bezeichnung  
FROM Produkt
```

ist äquivalent zu

```
SELECT Produkt.Bezeichnung  
FROM Produkt
```

ist äquivalent zu

```
SELECT p.Bezeichnung  
FROM Produkt p
```



Präfixe für Eindeutigkeit

SELECT Herstld, Name, Prodlid, Bezeichnung, Preis
FROM Produkt, Hersteller
WHERE Produkt.Herstld = Hersteller.Herstld

- Attribut Herstld existiert sowohl in der Tabelle Produkt als auch in Hersteller!

⇒ richtig mit Präfix:

SELECT Hersteller.Herstld, Name, Prodlid, Bezeichnung, Preis
FROM Produkt, Hersteller
WHERE Produkt.Herstld = Hersteller.Herstld

⇒ Als Präfix entweder Relationenname oder Tupelvariable verwenden:

SELECT h.Herstld, Name, Prodlid, Bezeichnung, Preis
FROM Produkt p, Hersteller h
WHERE p.Herstld = h.Herstld



Tupelvariablen für mehrfachen Zugriff

- Einführung von Tupelvariablen erlaubt mehrfachen Zugriff auf eine Relation (z.B. bei rekursiven Beziehungen):

PRODUKT:	<u>PRODID</u>	BEZEICHNUNG	PREIS	BESTAND	HERSTID	PARENTID → PRODID
----------	---------------	-------------	-------	---------	---------	-------------------

- Spalten lauten dann:

Spalten lauten dann:
eins.ProdID, *eins*.Bezeichnung, ...
zwei.ProdID, *zwei*.Bezeichnung, ...

- ⇒ bei der Verwendung von Tupelvariablen, kann der Name einer Tupelvariablen zur Qualifizierung eines Attributs benutzt werden:

```
SELECT eins.Bezeichnung, zwei.Bezeichnung  
FROM Produkt eins, Produkt zwei  
WHERE eins.ProdID = zwei.ParentID
```



Die **where**-Klausel

```
SELECT ...  
FROM ...  
WHERE bedingung
```

- Formen der Bedingung
 - Vergleich eines Attributs mit einer Konstanten: *attribut* Θ *konstante*
 - boolesches Prädikat Θ ist = oder \neq , bei linear geordneten Wertebereichen auch $<$, \leq , \geq oder $>$
 - Vergleich zwischen zwei Attributen mit kompatiblen Wertebereichen: *attribut*₁ Θ *attribut*₂
 - logische Verknüpfung mehrerer Bedingungen mit **OR**, **AND** und **NOT**



where-Klausel – Beispiele

- Verbundbedingung (Attributvergleich)

```
SELECT *  
FROM Produkt, Hersteller  
WHERE Produkt.HerstID = Hersteller.HerstID
```

- Vergleich Attribut und Konstante

```
SELECT *  
FROM Produkt  
WHERE Preis > 100
```

- Kombination mehrerer Bedingungen

```
SELECT *  
FROM Produkt, Hersteller  
WHERE Produkt.HerstID = Hersteller.HerstID AND Preis > 100
```




Übung (Relationenalgebra + SQL)

PRODUKT:	<u>PRODID</u>	BEZEICHNUNG	PREIS	BESTAND	HERSTID
	201	Skyscraper	99.0	12	901
	202	Himmelsstürmer	129.0	4	901
	203	Rainbow Hopper	45.0	20	902
	204	TumbleAround	21.0	30	901
	205	2Hi4U	129.0	1	902

KUNDE:	<u>KNR</u>	NAME	PLZ	ORT
	101	Schaarschmidt	60528	Frankfurt
	102	Lufter	65196	Wiesbaden
	103	Nowitzky	60431	Frankfurt
	104	Gollmick	69190	Walldorf

HERSTELLER:	<u>HERSTID</u>	NAME
	901	Flattermann GmbH
	902	Dragon.com
	903	KiteSports

BESTELLUNG:	<u>BID</u>	DATUM	KNR	PRODID
	1	01.03.2014	103	201
	2	11.03.2014	101	202
	3	05.05.2014	104	202
	4	10.12.2014	103	205
	5	03.01.2015	101	205

- Geben Sie für alle Produkte, von denen weniger als 10 Produkte im Bestand sind, die Produktbezeichnung sowie den Herstellernamen an.
- Geben Sie für jeden Kunden seinen Namen und die Bezeichnung der von ihm bestellten Produkte an.



Mengenoperationen

Anfragen wie

- Alle Kunden, die jemals etwas bestellt haben ...
- Alle Kunden, die noch nie etwas bestellt haben ...
- Produkte, die noch nie bestellt wurden ...

Umsetzung?



in-Prädikat und geschachtelte Anfragen

- Notation:

Bedingung **IN** (SFW-block)

- Beispiel

```
SELECT Name FROM Kunde  
WHERE KNr IN  
  ( SELECT KNr FROM Bestellung)
```

- ⇒ **not in** kann für Bildung von Differenzen verwendet werden, z.B. für die Anfrage „Kunden ohne Bestellungen“:

```
SELECT Name FROM Kunde  
WHERE KNr NOT IN  
  ( SELECT KNr FROM Bestellung)
```

- Achtung: Bei Verwendung von **in** bzw. **not in** müssen Spaltennamen übereinstimmen – sonst Anpassung mit **as**

```
SELECT attribut-name AS new-name FROM ...
```



Mengenoperationen: **union**-Operator

KUNDE:	<u>KNR</u>	NAME	PLZ	ORT
	101	Schaarschmidt	60528	Frankfurt
	102	Lufter	65196	Wiesbaden
	103	Nowitzky	60431	Frankfurt
	104	Gollmick	69190	Walldorf

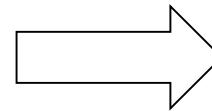
INTERESSENT:	<u>KNR</u>	NAME
	901	Küspert		
	902	Lufter		

- Namen aller Kunden und Interessenten?
- Expliziter Vereinigungsoperator: **union**

```
SELECT A, B, C FROM R  
UNION  
SELECT A, B, C FROM S
```

- Beispiel

```
SELECT Name FROM Interessent  
UNION  
SELECT Name FROM Kunde
```



ERGEBNIS:	NAME
	Gollmick
	Lufter
	Küspert
	Nowitzky
	Schaarschmidt



Mengenoperationen

- Mengenoperationen erfordern kompatible Wertebereiche für Paare korrespondierender Attribute:
 - beide Wertebereiche sind gleich oder
 - beide sind auf character basierende Wertebereiche (unabhängig von der Länge der Strings) oder
 - beide sind numerische Wertebereiche (unabhängig vom genauen Typ) wie integer oder float
- Expliziter Vereinigungsoperator: **union**
- Differenz und Durchschnitt können durch geschachtelte Anfragen ausgedrückt werden, oder:
- ab SQL-92 existieren explizite Differenz- und Durchschnittsoperatoren: **intersect** und **except** (werden aber häufig von Systemen noch nicht unterstützt!)



Mächtigkeit des SQL-Kern

Relationenalgebra	SQL
Projektion	select distinct
Selektion	where ohne Schachtelung
Verbund	from, where from mit join oder natural join (folgt später)
Umbenennung	from mit Tupelvariable as in select-Klausel
Differenz	where mit Schachtelung except
Durchschnitt	where mit Schachtelung intersect
Vereinigung	union



Datendefinition und -manipulation

- ✓ Datendefinition
 - ✓ SQL DDL

- ✓ Datenmanipulation und –abfragen
 - ✓ Grundlagen von Anfragen
 - ✓ Relationenalgebra
 - **SQL DML**
 - ✓ Elementare Datenmanipulation
 - ✓ SQL-Kern: SFW-Block
 - Erweiterungen des SFW-Blocks
 - Aggregatfunktionen und Gruppierungen
 - Veränderungen am Datenbestand
 - Andere Datenbankabfragesprachen



Erweiterungen des SFW-Block

Erweiterungen des SFW-Blocks

- innerhalb der **from**-Klausel explizite Verbundoperationen (natural join, äußere Verbunde etc.)
- innerhalb der **where**-Klausel weitere Arten von Bedingungen, z.B. Bereichsanfragen und Bedingungen mit „Wildcards“
- innerhalb der **select**-Klausel die Anwendung von skalaren Operationen und Aggregatfunktionen
- zusätzliche Klauseln **group by** und **having** und
- Sortierung mit **order by**



Verbunde als explizite Operatoren: JOIN

- neuere SQL-Versionen (ab SQL-92)
 - kennen mehrere explizite Verbundoperatoren (engl. *join*)
 - als Abkürzung für die ausführliche Anfrage mit Kreuzprodukt aufzufassen
- **JOIN**
 - Verbund mit Prädikatangabe:

```
SELECT *  
FROM Produkt JOIN Hersteller ON Produkt.HerstId = Hersteller.HerstId
```

- **NATURAL JOIN:**
 - Gleichverbund über alle Attribute gleichen Namens
- ```
SELECT *
FROM Produkt NATURAL JOIN Hersteller
```
- „Join für Tippfaule“ ⇨ Achtung: Fehlerquelle!



# Äußere Verbunde

---

- zusätzlich zu klassischem Verbund (**inner join**): in SQL-92 auch äußerer Verbund
  - ⇒ Übernahme von „dangling tuples“ in das Ergebnis und Auffüllen mit Nullwerten
- **outer join** übernimmt alle Tupel beider Operanden (Langfassung: **full outer join**)
- **left outer join** bzw. **right outer join** übernimmt alle Tupel des linken bzw. des rechten Operanden
- äußerer natürlicher Verbund jeweils mit Schlüsselwort **natural**, also z.B. **natural left outer join**



# Weitere Selektionen in SQL

---

- Bereichsselektion
- Ungewissheitsselektion
- Nullwerte



# Bereichsselektion

- Notation:

attribut **BETWEEN** konstante<sub>1</sub> **AND** konstante<sub>2</sub>

- ist Abkürzung für

attribut  $\geq$  konstante<sub>1</sub> **AND** attribut  $\leq$  konstante<sub>2</sub>

- schränkt damit Attributwerte auf das abgeschlossene Intervall [konstante<sub>1</sub>, konstante<sub>2</sub>] ein

- Beispiel:

```
SELECT Bezeichnung
FROM Produkt
WHERE Preis BETWEEN 50 AND 100
```



# Ungewissheitsselektion

- Notation:

attribut **LIKE** spezialkonstante

- Mustererkennung in Strings (Suche nach mehreren Teilzeichenketten)
- Spezialkonstante kann die Sondersymbole % und \_ beinhalten
  - % steht für kein oder beliebig viele Zeichen
  - \_ steht für genau ein Zeichen
- Beispiel

```
SELECT *
FROM Produkt
WHERE Bezeichnung LIKE 'Rainbow%opper'
```



# Selektion nach Nullwerten

---

- *Null-Selektion* wählt Tupel aus, die bei einem bestimmten Attribut Nullwerte enthalten

- Notation:

attribut **IS NULL**

- Beispiel:

```
SELECT *
FROM Produkt
WHERE Preis IS NULL
```

```
SELECT *
FROM Produkt
WHERE Preis IS NOT NULL
```



# Erweiterungen des SFW-Block

---

## Erweiterungen des Select-From-Where-(SFW)-Blocks

- ✓ innerhalb der **from**-Klausel explizite Verbundoperationen (natural join, äußere Verbunde etc.)
- ✓ innerhalb der **where**-Klausel weitere Arten von Bedingungen, z.B. Bereichsanfragen und Bedingungen mit „Wildcards“
- innerhalb der **select**-Klausel die Anwendung von skalaren Operationen und Aggregatfunktionen
- zusätzliche Klauseln **group by** und **having** und
- Sortierung mit **order by**



# Skalare Ausdrücke

- skalare Operationen auf
  - numerischen Wertebereichen: etwa +, −, und /
  - Strings: Operationen wie **char\_length** (aktuelle Länge eines Strings), Konkatenation || und **substring** (Suchen einer Teilzeichenkette an bestimmten Positionen des Strings),
  - Datumstypen und Zeitintervallen: Operationen wie **current\_date** (aktuelles Datum), **current\_time** (aktuelle Zeit), +, − und \*
- Beispiel: Umwandlung des DM-Preises in den aktuellen Euro-Preis:

```
SELECT Prodlid, Bezeichnung, Preis / 1.95583 AS EuroPreis
FROM Produkt
```

| ERGEBNIS: | PRODID | BEZEICHNUNG    | EUROPREIS |
|-----------|--------|----------------|-----------|
|           | 201    | Skyscraper     | 50.61     |
|           | 202    | Himmelsstürmer | 65.96     |
|           | 203    | Rainbow Hopper | 23.01     |
|           | 204    | TumbleAround   | 10.74     |
|           | 205    | 2Hi4U          | 65.69     |





# Aggregatfunktionen und Gruppierung

---

- Aggregatfunktionen berechnen neue Werte für eine gesamte Spalte, etwa die Summe oder den Durchschnitt der Werte einer Spalte
  - Beispiel: Ermittlung des Durchschnittspreises aller Artikel oder des Gesamtumsatzes über alle verkauften Produkte
- bei zusätzlicher Anwendung von Gruppierung: Berechnung der Funktionen pro Gruppe, z.B. der Durchschnittspreis pro Warengruppe oder der Gesamtumsatz pro Kunde



# Aggregatfunktionen

---

## Aggregatfunktionen in Standard-SQL:

- **count**: berechnet Anzahl der Werte einer Spalte oder alternativ (im Spezialfall **count(\*)**) die Anzahl der Tupel einer Relation
- **sum**: berechnet die Summe der Werte einer Spalte (nur bei numerischen Wertebereichen)
- **avg**: berechnet den arithmetischen Mittelwert der Werte einer Spalte (nur bei numerischen Wertebereichen)
- **max** bzw. **min**: berechnen den größten bzw. kleinsten Wert einer Spalte

vor dem Argument (außer im Fall von **count(\*)**) optional auch die Schlüsselwörter **distinct** oder **all**:

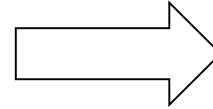
- **distinct**: vor Anwendung der Aggregatfunktion werden doppelte Werte aus der Menge von Werten, auf die die Funktion angewendet wird, eliminiert
- **all**: Duplikate gehen mit in die Berechnung ein (Default-Voreinstellung)
- Nullwerte werden in jedem Fall vor Anwendung der Funktion aus der Wertemenge eliminiert (außer im Fall von **count(\*)**)



# Aggregatfunktionen – Beispiele

- Gesamtbestand aller Produkte:

```
SELECT SUM (Bestand) FROM Produkt
```



| SUM(BESTAND) |
|--------------|
| 67           |

- Anzahl der Bestellungen:

```
SELECT COUNT (*) FROM Bestellung
```

- Anzahl der Kunden, für die derzeit eine Bestellung vorliegt:

```
SELECT COUNT (DISTINCT Knr) FROM Bestellung
```

- Durchschnittspreis aller Produkte:

```
SELECT AVG (Preis) FROM Produkt
```



# Aggregatfunktionen in **where**-Klausel

---

- Aggregatfunktionen liefern nur einen Wert
- ⇒ Einsatz in Konstanten-Selektionen der **where**-Klausel möglich
- Beispiel: Produkte, deren Bestand unter dem Durchschnitt liegt:

```
SELECT ProdID, Bezeichnung
FROM Produkt
WHERE Bestand <
 (SELECT AVG(Bestand) FROM Produkt)
```

- Generelle Anmerkung: Bei Verwendung von Vergleichsoperatoren wie **=**, **<>**, **<**, **<=**, **>=** oder **>** darf das Ergebnis des Subselect nur ein einzelner Wert sein (nicht mehrere Tupel!)



# Gruppierung

- Gruppierung dient dazu, Berechnung nicht auf der ganzen Tabelle, sondern auf Gruppen durchzuführen:

```
SELECT ...
FROM ...
[WHERE ...]
[GROUP BY attributliste]
[HAVING bedingung]
```

- Beispiel: Anzahl der Bestellungen pro Kunde im Jahr 2014

```
SELECT knr, count(*)
FROM bestellung
WHERE datum BETWEEN '01.01.2014' and '31.12.2014'
GROUP BY knr;
```



# Gruppierung: Abarbeitung – 1(2)

## 1. from-Klausel: **FROM** Bestellung

| BID | DATUM      | KNR | PRODID |
|-----|------------|-----|--------|
| 1   | 01.03.2014 | 103 | 201    |
| 2   | 11.03.2014 | 101 | 202    |
| 3   | 05.05.2014 | 104 | 202    |
| 4   | 10.12.2014 | 103 | 205    |
| 5   | 03.01.2015 | 101 | 205    |

## 2. where-Klausel: **WHERE** datum **BETWEEN** '01.01.2014' and '31.12.2014'

| BID | DATUM      | KNR | PRODID |
|-----|------------|-----|--------|
| 1   | 01.03.2014 | 103 | 201    |
| 2   | 11.03.2014 | 101 | 202    |
| 3   | 05.05.2014 | 104 | 202    |
| 4   | 10.12.2014 | 103 | 205    |

## 3. group by Klausel: **GROUP BY** knr



# Gruppierung: Abarbeitung 2(2)

## 3. group by Klausel: **GROUP BY** knr

| KNR | N   |            |        |
|-----|-----|------------|--------|
|     | BID | DATUM      | PRODID |
| 101 | 2   | 11.03.2014 | 202    |
| 103 | 1   | 01.03.2014 | 201    |
|     | 4   | 10.12.2014 | 205    |
| 104 | 3   | 05.05.2014 | 202    |

Kann so NICHT  
ausgegeben werden!

## 4. select-Klausel: **SELECT** knr, count(\*)

| KNR | COUNT(*) |
|-----|----------|
| 101 | 1        |
| 103 | 2        |
| 104 | 1        |



# Selektionsbedingung auf den gruppierten Werten: **having**-Klausel

- Beispiel: Anzahl der Bestellungen pro Kunde im letzten Jahr für diejenigen Kunden, die mehr als 1 mal bestellt haben

```
SELECT knr, count(*)
FROM bestellung
WHERE datum BETWEEN '01.01.2014' and '31.12.2014'
GROUP BY knr
HAVING count(*) > 1;
```

- Abarbeitungsreihenfolge:  
having-Klausel wird nach der Gruppierung (group by) ausgeführt:

1. **from**
2. **where**
3. **group by**
4. **having**
5. **select**





# Selektion auf der Gruppierung

## 3. group by Klausel: **GROUP BY** knr

| KNR | N   |            |        |
|-----|-----|------------|--------|
|     | BID | DATUM      | PRODID |
| 101 | 2   | 11.03.2014 | 202    |
| 103 | 1   | 01.03.2014 | 201    |
|     | 4   | 10.12.2014 | 205    |
| 104 | 3   | 05.05.2014 | 202    |

## 4. having-Klausel: **HAVING** count(\*) > 1

| KNR | N   |            |        |
|-----|-----|------------|--------|
|     | BID | DATUM      | PRODID |
| 103 | 1   | 01.03.2014 | 201    |
|     | 4   | 10.12.2014 | 205    |

## 5. select-Klausel: **SELECT** knr, count(\*)

| KNR | COUNT(*) |
|-----|----------|
| 103 | 2        |



# Übung

---

- Relationales Modell: Geldinstitute mit verschiedenen Filialen  
GELDINSTITUT (BLZ, NAME)  
FILIALE (FID, BLZ → GELDINSTITUT (BLZ), ANZAHLMITARBEITER)

- Aufgabe:  
Ermitteln Sie die BLZ aller Geldinstitute, die in ihren Filialen in Summe mehr als 300 Mitarbeiter haben.

Wie würde sich die Anfrage verändern, wenn ermittelt werden soll, welche Geldinstitute im Durchschnitt pro Filiale mehr als 20 Mitarbeiter haben?

- Zusatzaufgabe:  
Geben Sie auch den zugehörigen Namen des Geldinstituts mit aus.



# Sortierung mit **order by**

---

- Notation:

```
SELECT ...
FROM ...
[WHERE ...]
[GROUP BY ...]
[HAVING ...]
[ORDER BY attributliste [ASC | DESC]]
```

- Beispiel:

```
SELECT Bezeichnung, Preis
FROM Produkt
ORDER BY Preis ASC;
```

- Sortierung aufsteigend (**ASC**) oder absteigend (**DESC**)



# Weitere SQL-Anfragekonstrukte

---

- SQL-Abfragen mit Existenzquantoren (**all**, **any**, **some** and **exists**)
- Rekursive Anfragen
- Benannte Anfragen
- Begrenzung der Anzahl der Anfrageergebnisse (insgesamt oder „portionsweise“)
- ...



# auch so kann eine SQL-Anfrage aussehen ...

- Generierte SQL-Anfrage - durch Tool zur Entscheidungsunterstützung (Online Analytical Processing, OLAP) und GUI-Nutzung erzeugt

```
select distinct a.fn
from T1 a
where a.owf =
 (select min (b.owf)
 from T1 b
 where (1=1) and (b.aid='SAS' and
 b.fc in (select c.cid
 from T2 c
 where c.cn='HKG') and
 b.tc in (select d.cid
 from T2 d
 where e.cn='HLYD') and
 b.fid in (select e.fid
 from T3 e
 where e.did in
 (select f.did
 from T4 f
 where f.dow='saun')) and
 b.fdid in (select g.did
 from T4 g
 where g.dow='saun')))) and
 (1=1) and (a.aid='SAS' and
 a.fc in (select h.cid
 from T2 h
 where h.cn='HKG') and
 a.tc in (select i.cid
 from T2 i
 where i.cn='HLYD') and
 a.did in (select j.fid
 from T3 j
 where j.did in
 (select k.did
 from T4 k
 where k.dow='saun')) and
 a.fdid in (select l.did
 from T4 l
 where l.dow='saun'))
```

Quelle: Härder/Rahm:1999



# Zusammenfassung SQL-Anfragen

---

- Mengenorientierte Spezifikation, verschiedene Typen von Anfragen
- Vielfalt an Suchanfragen
- Auswahlmächtigkeit von SQL ist höher als die der Relationenalgebra (Gruppierung und Aggregation)
- Optimierung der Anfragen durch das DBMS



# Datendefinition und -manipulation

---

- ✓ Datendefinition
  - ✓ SQL DDL
  
- ✓ Datenmanipulation und –abfragen
  - ✓ Grundlagen von Anfragen
  - ✓ Relationenalgebra
  - **SQL DML**
    - ✓ Elementare Datenmanipulation
    - ✓ SQL-Kern: SFW-Block
    - ✓ Erweiterungen des SFW-Blocks
    - ✓ Aggregatfunktionen und Gruppierungen
      - Veränderungen am Datenbestand
  
  - Andere Datenbankabfragesprachen



# Änderungsoperationen in SQL

---

- **insert:**  
Einfügen eines oder mehrerer Tupel in eine Basisrelation oder Sicht
  - **update:**  
Ändern von einem oder mehreren Tupel in einer Basisrelation oder Sicht
  - **delete:**  
Löschen eines oder mehrerer Tupel aus einer Basisrelation oder Sicht
- ⇒ Lokale und globale Integritätsbedingungen müssen bei Änderungsoperationen automatisch vom System überprüft werden!!!





# update-Anweisung

- Syntax:

```
UPDATE basisrelation
SET attribut1 = ausdruck1, ..., attributn = ausdruckn
[WHERE bedingung]
```

- Beispiele: Update eines, mehrerer bzw. aller Tupel

```
UPDATE Produkt
SET Bestand = Bestand - 1
WHERE ProdId = 204
```

```
UPDATE Produkt
SET Preis = Preis * 0.9
WHERE Bestand < 5
```

```
UPDATE Produkt
SET Bestand = 0
```



# delete-Anweisung

---

- Syntax:

```
DELETE
FROM basisrelation
[WHERE bedingung]
```

- Beispiele: Löschen eines, mehrerer bzw. aller Tupel

```
DELETE
FROM Bestellung
WHERE BID = 2
```

```
DELETE
FROM Bestellung
WHERE Knr = 103
```

```
DELETE
FROM Bestellung
```



# Ändern und Löschen von Tupeln

---

- Beim Ändern und Löschen von Tupeln können wiederum die Ergebnisse von Subselects verwendet werden.
- Beispiel:  
Löschen der Bestellungen aller Kunden aus 'Frankfurt':

```
DELETE
FROM Bestellung
WHERE knr IN
 (SELECT knr
 FROM Kunde
 WHERE Ort = 'Frankfurt');
```



# Ergänzungen zur insert-Anweisung

Statt Einfügen von explizit angegebenen Werten:

```
INSERT INTO relationen-name [(attribut1, ..., attributn)]
VALUES (konstante1, ..., konstanten)
```

Einfügen von berechneten(!) Daten:

```
INSERT INTO relationen-name [(attribut1, ..., attributn)]
SQL-Anfrage
```

- Beispiel
  - Annahme: Relation Kunden\_Frankfurt existiert, hat gleiche Struktur wie Kunde und ist leer:

```
INSERT INTO Kunden_Frankfurt
(SELECT * FROM Kunde WHERE Ort = 'Frankfurt');
```

- Anmerkungen
  - Die Datentypen der Zielrelation müssen kompatibel zu denen der Ursprungsrelation sein
  - Die kopierten Tupel sind unabhängig von ihren Ursprungstupeln



# Datendefinition und -manipulation

---

- ✓ Datendefinition
  - ✓ SQL DDL
  
- ✓ Datenmanipulation und –abfragen
  - ✓ Grundlagen von Anfragen
  - ✓ Relationenalgebra
  - ✓ SQL DML
  - Andere Datenbankabfragesprachen



# Die Sprache QBE (Query by Example)

---

- Anfragen in QBE: Einträge in Tabellengerüsten
- Intuition: *Beispieleinträge* in Tabellen
- Vorläufer verschiedener tabellenbasierter Anfrageschnittstellen kommerzieller Systeme
- Theoretische Grundlage: Relationenkalkül



# QBE – Beispiel

- Für welche Vorlesungen mit mehr als 2 Semesterwochenstunden ist 'Datenbanken I' Voraussetzung?*

| Vorl | V_Bez  | SWS | Semester | Studiengang |
|------|--------|-----|----------|-------------|
|      | P. _DB | > 2 |          |             |

| Vorl_Voraus | V_Bez | Voraussetzung |
|-------------|-------|---------------|
|             | _DB   | Datenbanken I |



# QBE in MS Access

- MS-Access: Datenbankprogramm für Windows
  - Basisrelationen mit Schlüsseln
  - Fremdschlüssel über graphische Angabe von Beziehungen
  - Unterstützung von QBE:

Vorl>2SWS\_Vor\_DBI : Auswahlabfrage

| Feld:       | V_Bezeichnung                       | SWS                      | Voraussetzung            |
|-------------|-------------------------------------|--------------------------|--------------------------|
| Tabelle:    | Vorlesungen                         | Vorlesungen              | Vorl_Voraus              |
| Sortierung: |                                     |                          |                          |
| Anzeigen:   | <input checked="" type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Kriterien:  |                                     | >2                       | "Datenbanken I"          |
| oder:       |                                     |                          |                          |





# Vorlesung Datenbanken 1

