Technische Hochschule Rosenheim

Kapitel 7 – Advanced SQL

Vorlesung Datenbanken

Dr. Kai Höfig

Überblick

Kapitel 7: Advanced SQL

- Aggregation und Gruppierung
- Joins
- Sortierung, Top-k-Anfragen und Nullwerte
- Benannte und rekursive Ausdrücke
- Division
- Integrität
- Trigger
- Sichten
- Zugriffskontrolle
- PSM

SQL-Kern

- **`select`** Projektionsliste
  arithmetische Operationen & Aggregatfunktionen
- **`from`** zu verwendende Relationen, evtl. Umbenennungen
- **`where`** Selektions-, Verbundbedingungen
  geschachtelte Anfragen (wieder ein SFW-Block)
- **`group by`** Gruppierung für Aggregatfunktionen
- **`having`** Selektionsbedingungen an Gruppen
- **`order by`** Ausgabereihenfolge
- Auswertungsreihenfolge:
  `from`, **`where`**, **`group by`**, **`having`**, **`select`**, **`order by`**

Aggregatfunktionen und Gruppierung

- Aggregatfunktionen berechnen neue Werte für eine gesamte Spalte, etwa die Summe oder den Durchschnitt der Werte einer Spalte

- Beispiele:
  - 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

- Einfaches Beispiel: Berechnung der Gesamt-Anzahl der Weine

```
select count(*) as Anzahl
from WEINE
```

Ergebnis:

Anzahl
7

Aggregatfunktionen in Standard-SQL (1)

- **`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
- Argumente einer Aggregatfunktion
  - ein Attribut der durch die **`from`**-Klausel spezifizierten Relation,
  - ein gültiger skalarer Ausdruck oder
  - im Falle der **`count`**-Funktion auch das Symbol **`*`**

Aggregatfunktionen in Standard-SQL (1)

- 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
  - **`all`**: Duplikate gehen mit in die Berechnung ein (Default-Voreinstellung)
  - Nullwerte werden vor Anwendung der Funktion aus der Wertemenge eliminiert. Ausnahme: **`count(*)`**

Aggregatfunktionen – Beispiele

- Anzahl der verschiedenen Weinregionen:

```
select count(distinct Region)
from   ERZEUGER
```

- Weine, die älter als der Durchschnitt sind:

```
select Name, Jahrgang
from   WEINE
where  Jahrgang < ( select avg(Jahrgang) from WEINE )
```

Aggregatfunktionen - Schachtelung

- Schachtelung von Aggregatfunktionen nicht erlaubt
- **Falsches** Beispiel:

```
select max(avg(A)) as Ergebnis
from R …
```

- Mögliche richtige Formulierung:

```
select max(Temp) as Ergebnis
from   ( select avg(A) as Temp from R ...)
```

Aggregatfunktionen in **`where`**-Klausel

- Aggregatfunktionen liefern nur einen Wert
  → Einsatz in Konstanten-Selektionen der **`where`**-Klausel möglich

- Beispiel: alle Weingüter, die nur einen Wein liefern:

```
select *
from   ERZEUGER e
where  1 = (
     select count(*)
     from   WEINE w
     where  w.Weingut = e.Weingut)
```

Gruppierung mittels `group by`

- Gruppierung mittels **`group by`** erlaubt das Zusammenfassen von Tupeln

- Beispiel: Anzahl der Weine nach Farbe

```
select   Farbe, count(*) as Anzahl
from     WEINE
group by Farbe
```

Ergebnis:

Farbe	Anzahl
Rot	5
Weiß	2

Selektion der Gruppen mittels **having**

- Selektion auf der Gruppierung mittels **having** möglich
  - in **where** nicht möglich wegen Auswertungsreihenfolge

- Beispiel: Regionen mit mehr als einem Wein

```
select   Region, count(*) as Anzahl
from     ERZEUGER natural join WEINE
group by Region
having   count(*) > 1
```

Ergebnis:

Region	Anzahl
South Australia	2
Kalifornien	3

Gruppierung: Schematische Ablauf (1)

- Relation

REL

A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	5
3	3	6	7
4	3	4	1
5	4	4	3
...	...	...	...

Auswertungsreihenfolge:
from, **where**, **group by**, **having**, **select**, **order by**

- Anfrage:

```
select    A, sum(D) as D_GESAMT
from      REL
where     A<=4
group by  A, B
having    sum(D)<10 and max(C)=4
```

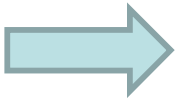
Gruppierung: Schematische Ablauf (2)

Gruppierung: Schritt 1

- **`from`** und **`where (from REL where A<=4)`**

REL

A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	5
3	3	6	7
4	3	4	1
5	4	4	3
...	...	...	...



Interne Tabelle im DBMS, nicht nach außen sichtbar!

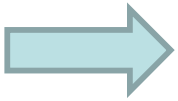
A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	5
3	3	6	7
4	3	4	1

Gruppierung: Schematische Ablauf (3)

Gruppierung: Schritt 2

- **`group by`** `A, B`

A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	5
3	3	6	7
4	3	4	1



A	B	N	
		C	D
1	2	3	4
		4	5
2	3	3	4
3	3	4	5
		6	7
4	3	4	1

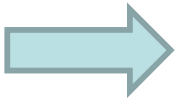
Interne Tabellen im DBMS, nicht nach außen sichtbar!

Gruppierung: Schematische Ablauf (4)

Gruppierung: Schritt 3

- **`having sum(`**D**`)<10 and max(`**C**`)=4`**

A	B	N	
		C	D
1	2	3	4
		4	5
2	3	3	4
3	3	4	5
		6	7
4	3	4	1



A	B	N	
		C	D
1	2	3	4
		4	5
4	3	4	1

Interne Tabellen im DBMS, nicht nach außen sichtbar!

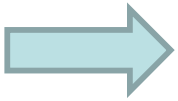
Gruppierung: Schematische Ablauf (5)

Gruppierung: Schritt 4

- **`select`** A, **`sum`**(D) **`as`** D_GESAMT

A	B	N	
		C	D
1	2	3	4
		4	5
4	3	4	1

Interne Tabelle im DBMS, nicht nach außen sichtbar!



A	D_GESAMT
1	9
4	1

Ergebnistabelle

Gruppierung

- Gruppierungsoperator γ:

$$\gamma_{f_1(x_1),f_2(x_2),\ldots f_n(x_n);A}(r(R))$$

- erweitert Attributschema von $r(R)$ um neue Attribute, die mit den Funktionsanwendungen $f_1(x_1), f_2(x_2), \ldots, f_n(x_n)$ korrespondieren
- Anwendung der Funktionen $f_i(x_i)$ auf die Teilmenge derjenigen Tupel von $r(R)$ die gleiche Attributwerte für die Attribute A haben
- In SQL:

```
select f1(x1), f2(x2), ..., fn(xn), A
from R
group by A
```

- Formale Semantik: siehe Literatur

Attribute für Aggregation bzw. `having`

- Zulässige Attribute hinter `select` bei Gruppierung auf Relation mit Schema R
  - Gruppierungsattribute G
  - Aggregationen auf Nicht-Gruppierungsattributen R - G
- Zulässige Attribute hinter `having`
  - Gruppierungsattribute G
  - Aggregationen auf Nicht-Gruppierungsattributen R - G

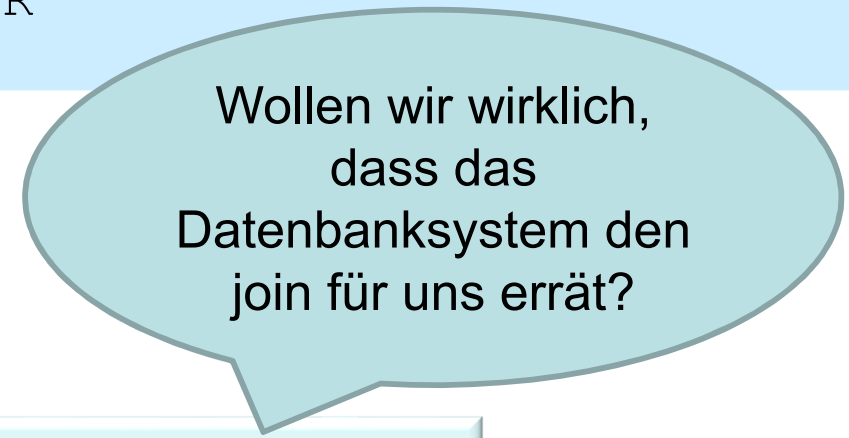
➔ **Typischer Fehler obwohl eigentlich absolut logisch, sonst funktioniert ja das zusammenschieben nicht mehr. Sollen weitere Einschränkungen vorgenommen werden, siehe where-Teil.**

Der Natural Join

- Natural join (dt. Natürlicher Verbund): Gleichheitsbedingung über alle gleichnamigen Attribute

- Beispiel:

```
select    Name, Angebaugebiet
from      WEINE natural join ERZEUGER
```



Der Equi Join

- Equi join (dt. Gleichverbund): Gleichheitsbedingung über explizit angegebene und evtl. verschiedene Attribute

In MySQL, in TSQL

- Beispiele:

```
select    Name, Angebaugebiet
from      WEINE join ERZEUGER
                on (WEINE.Weingut = ERZEUGER.Weingut)
```

```
select    KUNDEN.Name as Kundenname, WEINE.Angebaugebiet,
          WEINE.Name as Weinname
from      WEINE join KUNDEN
                on (WEINE.Anbaugebiet = KUNDEN.Lieblingsgebiet)
```

Der Theta Join

- Theta join (θ-join, dt. Thetaverbund): beliebige Verbundbedingung

- Beispiel:

In MySQL, in TSQL

```
select    KUNDEN.Name as Kundenname, WEIN.Name as Weinname,
          Preis
from      WEINE join KUNDEN
               on (WEINE.Preis <= KUNDEN.MaxPreis)
```

Kundenname	Weinname	Preis
Hans Huber	Zinfandel	3,99
Hans Huber	Pinot Noir	5,99
Hans Huber	Pinot Noir	9,99
Hans Huber	Chardonnay	1,99
Erwin Ehrlich	Zinfandel	3,99
Erwin Ehrlich	Chardonnay	1,99
Renate Rich	Creek Shiraz	23,90
...		

Der Semi Join

- Semi join (dt. Semijoin): nur Attribute eines Operanden erscheinen im Ergebnis
  - Sinn: Elimination der dangling Tupel
  - Left Semi Join: nur Attribute des linken Operanden erscheinen im Ergebnis
  - Right Semi Join: nur Attribute des rechten Operanden erscheinen im Ergebnis
  - Keine explizite Umsetzung in SQL, aber sehr einfach mittels Angabe der Attribute hinter SELECT DISTINCT zu implementieren

- Beispiel für einen Left Semi Join:

```
select distinct ERZEUGER.*
from   ERZEUGER e join WEINE w on (e.Weingut = w.WeinGut)
```

Weingut	Anbaugebiet	Region
Creek	Barossa Valley	South Australia
Helena	Napa Valley	Kalifornien
Chateau La Rose	Saint-Emilion	Bordeaux
Müller	Rheingau	Hessen
Bighorn	Napa Valley	Kalifornien

In MySQL, in TSQL

Die einfachen joins in TSQL

- Natural join durch explizite Angebe der Verbundbedingung, keine explizite Implementierung vorhanden.
- Equi join ebenfalls
- Theta join ebenfalls
- Semi join ebenfalls

Beispielrelationen für Bestellungen

id	name	email
1	Michaela	123@456.de
2	Deike	123er@456.de
3	Klaus	12w3@456.de
4	Matze	12sss3@456.de
5	Herbert	1wwdc23@456.de
6	Carolin	1wewd23@456.de

id	datum	lieferadresse	kundennummer
1	2018-04-21	Marx Straße 4	1
2	2018-03-11	Lauterweg 12	1
3	2018-04-21	Marx Straße 8	1
4	2018-05-11	Bananengasse 189	2
5	2018-06-03	Lauterweg 12	2
6	2018-07-04	Wie auch immer Straße 9	5
7	2018-02-05	Marx Straße 4	5
8	2018-03-06	Marx Straße 4	NULL

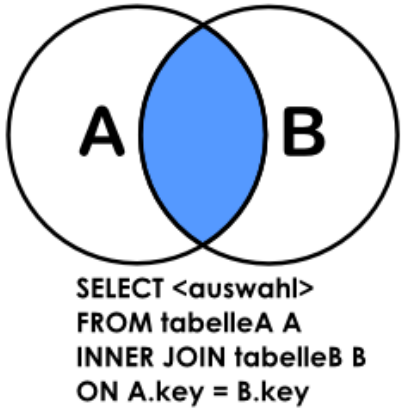
Inner Join – Die Schnittmenge

- **Nur die A** und **nur die B** zu denen **in beiden** Tabellen Informationen vorhanden sind.
- In diesem Beispiel sind also weder Kunden ohne Bestellung, noch Bestellungen ohne Kunden.

```
SELECT  kunde.name, bestellung.id, bestellung.datum
FROM    kunde, bestellung
WHERE   kunde.id = bestellung.kundennummer
```

oder

```
SELECT      kunde.name, bestellung.id, bestellung.datum
FROM        kunde
INNER JOIN  bestellung
ON          kunde.id = bestellung.kundennummer
```

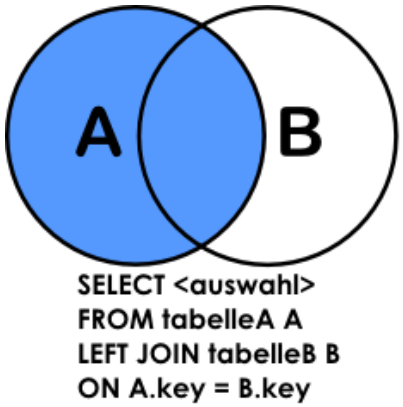


name	id	datum
Michaela	1	2018-04-21
Michaela	2	2018-03-11
Michaela	3	2018-04-21
Deike	4	2018-05-11
Deike	5	2018-06-03
Herbert	6	2018-07-04
Herbert	7	2018-02-05

Left Join – Alle

- **Alle von A** mit den Informationen von B (wenn vorhanden)
- In dem Beispilel sind das alle Kunden, auch die die keine Bestellung haben mit den Informationen aus der Tabelle *Bestellung*.

```
SELECT      kunde.name, bestellung.id,
bestellung.datum
FROM        kunde
LEFT JOIN   bestellung
ON          kunde.id = bestellung.kundennummer
```



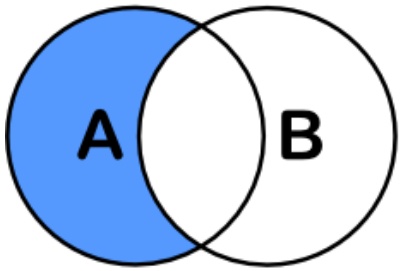
name	id	datum
Michaela	1	2018-04-21
Michaela	2	2018-03-11
Michaela	3	2018-04-21
Deike	4	2018-05-11
Deike	5	2018-06-03
Klaus	NULL	NULL
Matze	NULL	NULL
Herbert	6	2018-07-04
Herbert	7	2018-02-05
Carolin	NULL	NULL

Left Join – Nur das Komplement

- **Nur die A**, die **keine in B** haben
- In dem Beispiel sind das alle Kunden, die noch keine Bestellung haben

```
SELECT      kunde.name, bestellung.id,
bestellung.datum
FROM        kunde
LEFT JOIN    bestellung
ON          kunde.id = bestellung.kundennummer
WHERE       bestellung.kundennummer IS NULL
```

name	id	datum
Klaus	NULL	NULL
Matze	NULL	NULL
Carolin	NULL	NULL

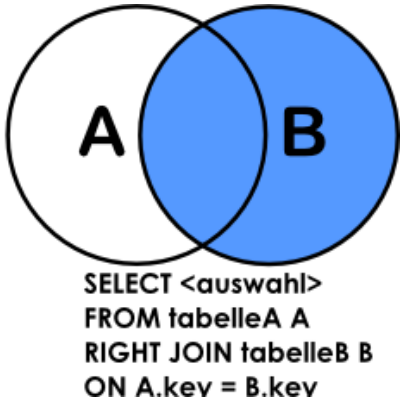



```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL
```

Right Join – Alle

- **Alle von B** mit den Informationen von A (wenn vorhanden)
- In dem Beispilel sind das alle Bestellungen, auch die die keinen Kunden haben mit den Informationen aus der Tabelle *Bestellung*.

```
SELECT      kunde.name, bestellung.id,
bestellung.datum
FROM        kunde
RIGHT JOIN  bestellung
ON          kunde.id = bestellung.kundennummer
```



name	id	datum
Michaela	1	2018-04-21
Michaela	2	2018-03-11
Michaela	3	2018-04-21
Deike	4	2018-05-11
Deike	5	2018-06-03
Herbert	6	2018-07-04
Herbert	7	2018-02-05
NULL	8	2018-03-06

Right Join – Alle

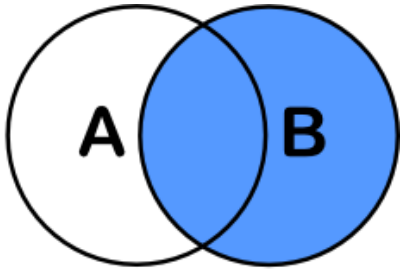
- **Alle von B** mit den Informationen von A (wenn vorhanden)
- In dem Beispilel sind das alle Bestellungen, auch die die keinen Kunden haben mit den Informationen aus der Tabelle *Bestellung*.

```
SELECT      kunde.name, bestellung.id,
bestellung.datum
FROM        kunde
RIGHT JOIN  bestellung
ON          kunde.id = bestellung.kundennummer
```

oder

```
SELECT      kunde.name, bestellung.id,
bestellung.datum
FROM        bestellung
LEFT JOIN   kunde
ON          kunde.id = bestellung.kundennummer
```

name	id	datum
Michaela	1	2018-04-21
Michaela	2	2018-03-11
Michaela	3	2018-04-21
Deike	4	2018-05-11
Deike	5	2018-06-03
Herbert	6	2018-07-04
Herbert	7	2018-02-05
NULL	8	2018-03-06



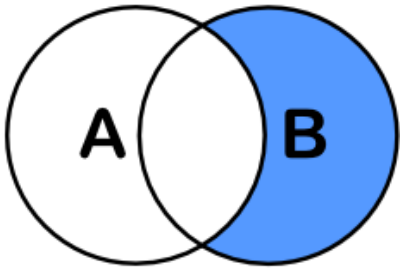

```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
```

Jeder Left Join lässt sich auch als Right Join darstellen und umgekehrt

Right Join – Nur das Komplement

- **Nur die B**, die **keine in A** haben
- In dem Beispiel sind das alle Bestellungen, die keinen Kunden haben

```
SELECT      kunde.name, bestellung.id,
bestellung.datum
FROM        kunde
RIGHT JOIN   bestellung
ON          kunde.id = bestellung.kundennummer
WHERE       bestellung.kundennummer IS NULL
```




```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
```

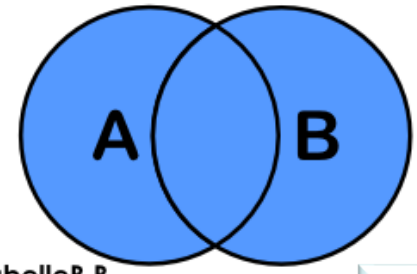
name	id	datum
NULL	8	2018-03-06

Full Outer Join

- Alle von A und alle von B jeweils mit den Informationen aus A und B, wenn vorhanden.
- In dem Beispile sind das alle Kunden, auch die, die keine Bestellung haben und alle Bestellungen, auch die, die keinen Kunden haben.

```
SELECT      kunde.name, bestellung.id, bestellung.datum
FROM        kunde
FULL OUTER JOIN         bestellung
ON          kunde.id = bestellung.kundennummer
```

name	id	datum
Michaela	1	2018-04-21
Michaela	2	2018-03-11
Michaela	3	2018-04-21
Deike	4	2018-05-11
Deike	5	2018-06-03
Klaus	NULL	NULL
Matze	NULL	NULL
Herbert	6	2018-07-04
Herbert	7	2018-02-05
Carolin	NULL	NULL
NULL	8	2018-03-06




```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
```

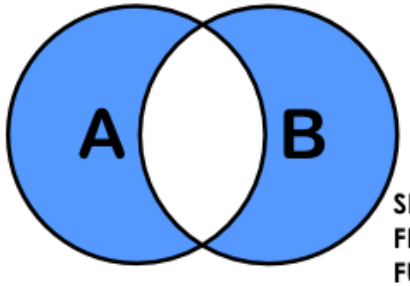
Nicht in MySQL, in TSQL

Full Outer Join

- Alle von A und alle von B jeweils mit den Informationen aus A und B, wenn vorhanden.
- In dem Beispile sind das alle Kunden, auch die, die keine Bestellung haben und alle Bestellungen, auch die, die keinen Kunden haben.

```
SELECT          kunde.name, bestellung.id, bestellung.datum
FROM            kunde
FULL OUTER JOIN         bestellung
ON              kunde.id = bestellung.kundennummer
WHERE           kunde.id IS NULL OR bestellung.id IS NULL
```

name	id	datum
Klaus	NULL	NULL
Matze	NULL	NULL
Carolin	NULL	NULL
NULL	8	2018-03-06

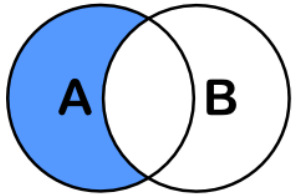



```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

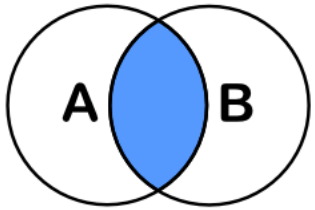
Nicht in MySQL, in TSQL

Zusammenbauen eines Outer joins

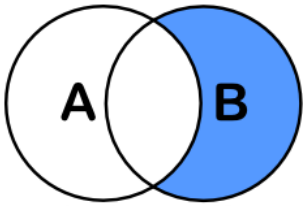
```
SELECT      kunde.name, bestellung.id, bestellung.datum
FROM        kunde
LEFT JOIN    bestellung
ON          kunde.id = bestellung.kundennummer
WHERE       bestellung.kundennummer IS NULL

UNION

SELECT      kunde.name, bestellung.id, bestellung.datum
FROM        kunde
INNER JOIN  bestellung
ON          kunde.id = bestellung.kundennummer

UNION

SELECT      kunde.name, bestellung.id, bestellung.datum
FROM        kunde
RIGHT JOIN    bestellung
ON          kunde.id = bestellung.kundennummer
WHERE       kunde.id IS NULL
```




```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL
```

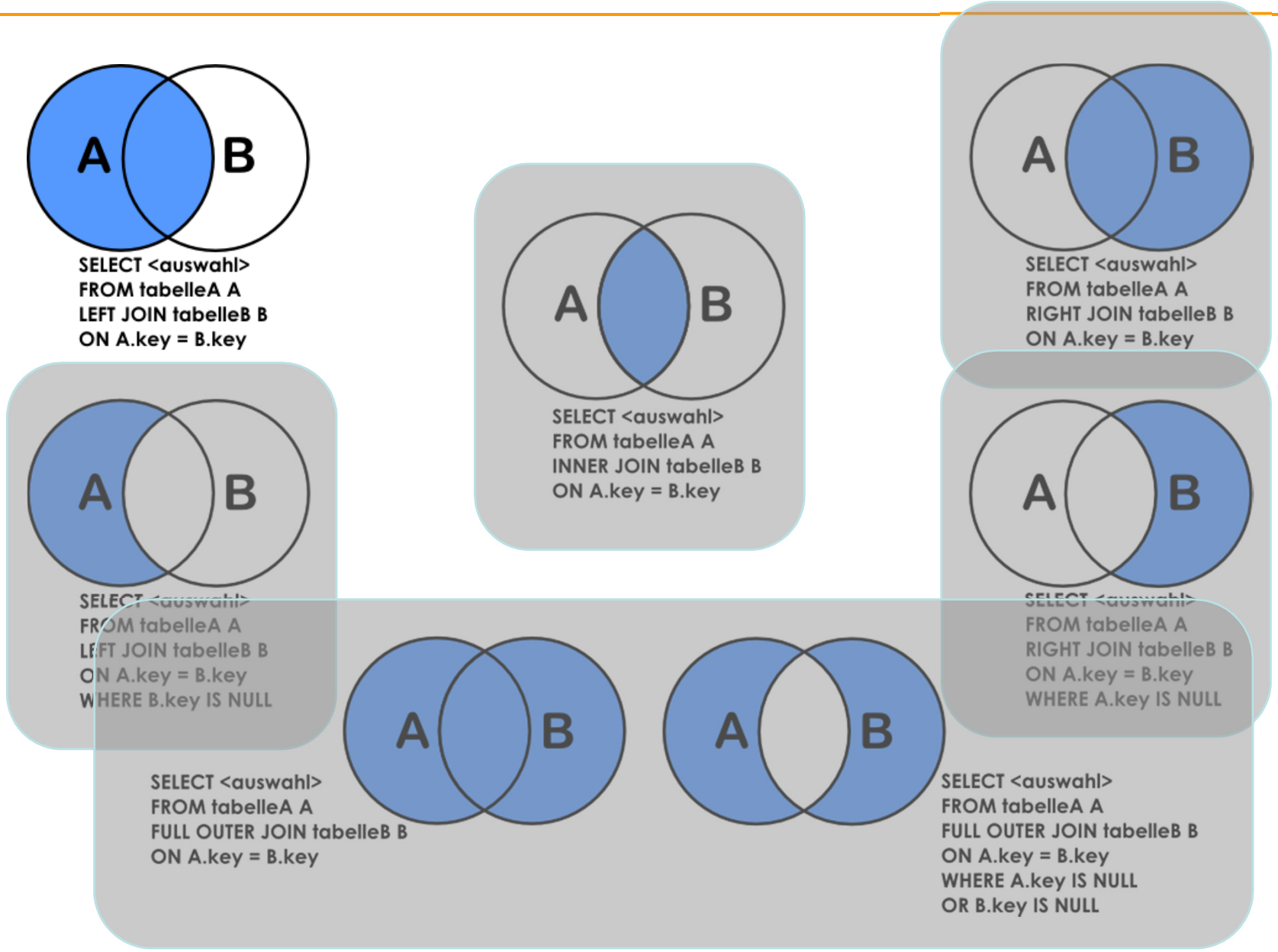



```
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key
```




```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
```

Outer Joins – Übersicht



Outer Joins – Beispiele (alle natural xxx joins)

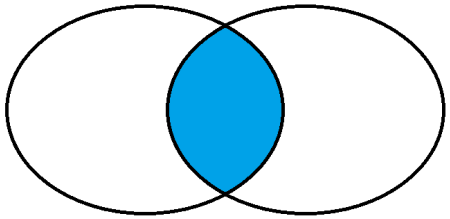
LINKS

A	B
1	2
2	3

RECHTS

B	C
3	4
4	5

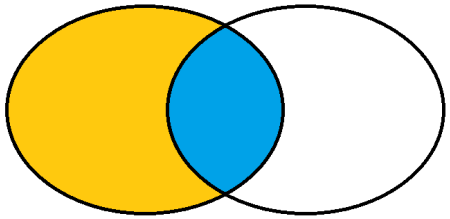
inner join



```
select A, LINKS.B, C
from   LINKS
       natural join
         RECHTS
```

A	B	C
2	3	4

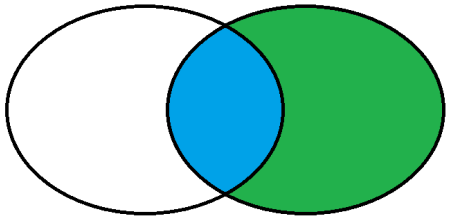
left outer join



```
select A, LINKS.B, C
from   LINKS
  natural left outer join
         RECHTS
```

A	B	C
1	2	⊥
2	3	4

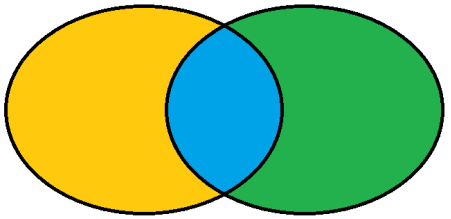
right outer join



```
select A, LINKS.B, C
from   LINKS
  natural right outer join
         RECHTS
```

A	B	C
2	3	4
⊥	4	5

(full) outer join



```
select A, LINKS.B, C
from   LINKS
  natural outer join
         RECHTS
```

A	B	C
1	2	⊥
2	3	4
⊥	4	5

Outer Joins - Ersatz durch Vereinigung

- Outer joins sind praktisch, aber nicht zwingend notwendig
- Beispiel: linker äußerer Verbund

```
select *
from   ERZEUGER natural join WEINE
  union all
select ERZEUGER.*, cast(null as int),
       cast(null as varchar(20)),
       cast(null as varchar(10)), cast(null as int),
       cast(null as varchar(20))
from   ERZEUGER e
where not exists (
                  select *
                  from   WEINE
                  where  WEINE.Weingut = e.Weingut)
```

Verbundvarianten

- Gegeben Relationen: $L(AB)$, $R(BC)$, $S(DE)$
- Equi-join: Gleichheitsbedingung über explizit angegebene und evtl. verschiedene Attribute

$$r(R) \bowtie_{C=D} r(S)$$

- Theta-join: beliebige Verbundbedingung

$$r(R) \bowtie_{\theta} r(S)$$
$$r(R) \bowtie_{C>D} r(S)$$

- Semi-join: nur Attribute eines Operanden erscheinen im Ergebnis

$$r(L) \ltimes r(R) = \pi_L(r(L) \bowtie r(R))$$
$$r(L) \rtimes r(R) = \pi_R(r(L) \bowtie r(R))$$

- Formale Semantik: siehe Literatur

Outer Joins

- Beachte: Notation der Symbole nicht Standardisiert!
- Full outer join: übernimmt alle Tupel beider Operanden

$$r \bowtie s$$

- Left outer join: übernimmt alle Tupel des linken Operanden

$$r \bowtie s$$

- Right outer join: übernimmt alle Tupel des rechten Operanden

$$r \bowtie s$$

- Formale Semantik: siehe Literatur

Sortierung mit `order by` (1)

- Notation

```
order by attributliste
```

- Beispiel

```
select   *
from     WEINE
order by Jahrgang
```

- Sortierung aufsteigend (**`asc`**) oder absteigend (**`desc`**)

- Sortierung als letzte Operation einer Anfrage
  → Sortierattribut muss in der **`select`**-Klausel vorkommen

Sortierung mit `order by` (2)

- Sortierung auch mit berechneten Attributen (Aggregaten) als Sortierkriterium möglich

- Beispiel

```
select   Weingut, count(*) as Anzahl
from     ERZEUGER natural join WEINE
group by Weingut
order by Anzahl desc
```

Sortierung: Top-k-Anfragen (1)

- Beispiel: Bestimme die 4 jüngsten Weine
- Lösung:

```
select   w1.WeinId, w1.Name, count(*) as Rang
from     WEINE w1, WEINE w2
where    w1.Jahrgang <= w2.Jahrgang      -- Schritt 1
group by w1.Name, w1.WeinID              -- Schritt 2
having   count(*) <= 4                   -- Schritt 3
order by Rang                            -- Schritt 4
```

- Ergebnis

WeinId	Name	Rang
3456	Zinfandel	1
2168	Creek Shiraz	2
4961	Chardonnay	3
2171	Pinot Noir	4

Sortierung: Top-k-Anfragen (2)

- **Top-k-Anfrage**: Liefert die besten k Elemente bzgl. einer Rangfunktion.

- **Lösungsmuster (Design-Pattern):**
  - **Schritt 1:** Zuordnung der nötigen Datensätze, um die Rangfunktion berechnen zu können
  - **Schritt 2:** Gruppierung nach den Elementen, Berechnung des Rangs
  - **Schritt 3:** Beschränkung auf Ränge ≤ k
  - **Schritt 4:** Sortierung nach Rang

- Beispiel: Ermittlung der k = 4 jüngsten Weine
  - Schritt 1: Zuordnung aller Weine die jünger sind
  - Schritt 2: Gruppierung nach Namen, Berechnung des Rangs
  - Schritt 3: Beschränkung auf Ränge ≤ 4
  - Schritt 4: Sortierung nach Rang

Sortierung: Top-k-Anfragen (3)

- **Top-k-Klausel**: Liefert die besten k Elemente bzgl. einer Sortierung.

```
select top(4)  w1.WeinId, w1.Name, count(*) as Rang
from     WEINE w1, WEINE w2
where    w1.Jahrgang <= w2.Jahrgang
group by w1.Name, w1.WeinID
order by Rang
```

```
select   w1.WeinId, w1.Name, count(*) as Rang
from     WEINE w1, WEINE w2
where    w1.Jahrgang <= w2.Jahrgang      -- Schritt 1
group by w1.Name, w1.WeinID              -- Schritt 2
having   count(*) <= 4                   -- Schritt 3
order by Rang                            -- Schritt 4
```

Behandlung von Nullwerten (1)

- Spezieller Wert in SQL: **`null`**
  - Bedeutung: unbekannt bzw. nicht anwendbar bzw. nicht vorhanden (je nach Anwendung)
- Test auf **`null`** Wert:
  - *`attr`* **`is null`** ergibt **`true`**, falls *`attr`* **`null`** ist
  - *`attr`* **`is not null`** ergibt **`false`**, falls *`attr`* **`null`** ist
  - Beispiel:

```
select * from ERZEUGER
       where Anbaugebiet is null
```

- Terme: Ergebnis ist **`null`**, sobald Nullwert in die Berechnung eingeht
  - Ausnahme: Aggregatfunktionen: Nullwerte vor Anwendung der Funktion entfernt
  - Ausnahme der Ausnahme: bei **`count`**(*) werden Nullwerte mitgezählt
- Vergleiche mit Nullwert: ergeben Wahrheitswert **`unknown`**
  - Somit 3 mögliche Werte für Boolesche Ausdrücke: `true`, `false` und `unknown`

Behandlung von Nullwerten (2)

- Boolesche Ausdrücke basieren damit auf dreiwertiger Logik
- Logiktabellen für die dreiwertige Logik

AND	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

NOT	
true	false
unknown	unknown
false	true

OR	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

Benannte Anfragen (1)

- Beispiel: Finde alle Weine die maximal 2 Jahre älter oder jünger als das durchschnittliche Alter aller Weine sind.

- Anfrage:

```
select  *
from    WEINE
where   Jahrgang >= (
            select avg(Jahrgang) from WEINE) - 2
    and Jahrgang <= (
            select avg(Jahrgang) from WEINE) + 2
```

- Unschön: Teilanfrage wird wiederholt
  - Duplizierter Code sollte vermieden werden (Fehleranfälligkeit)
  - Unübersichtlich

Benannte Anfragen (2)

- **Benannte Anfrage:** Anfrageausdruck, der in der folgenden Anfrage mehrfach referenziert werden kann (CTE, Common Table Expression)

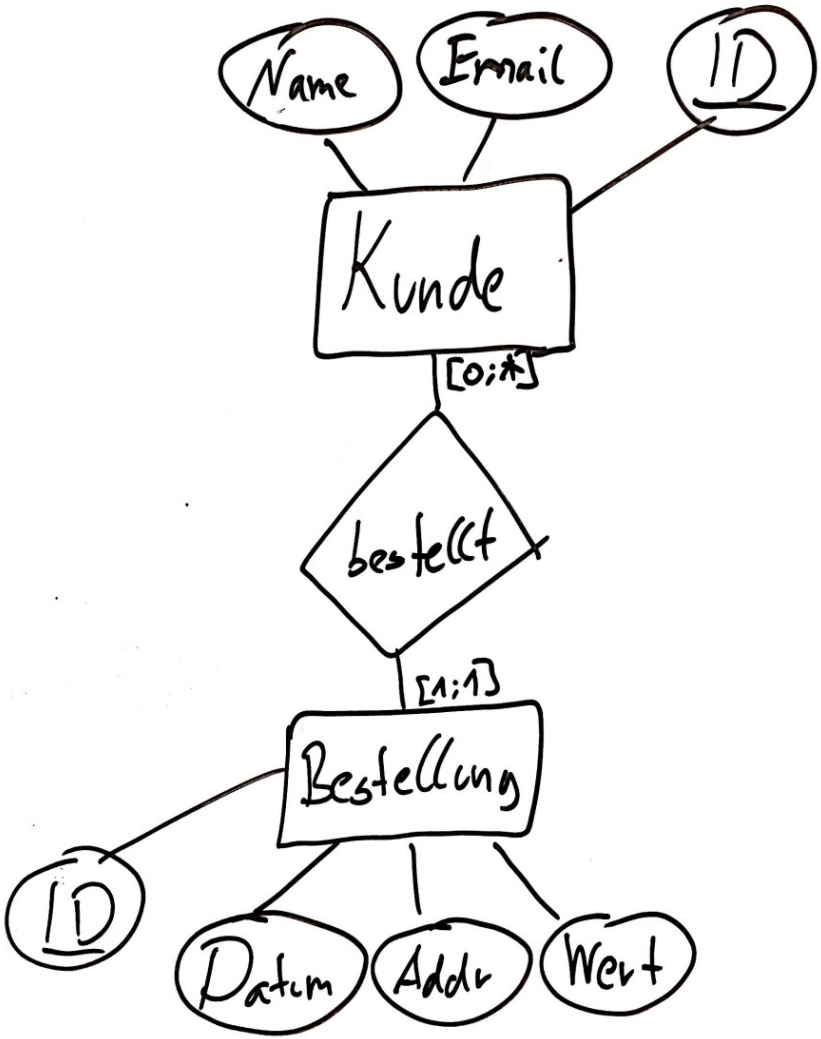
- Notation

```
with anfragename [(spaltenliste)] as (anfrageausdruck)
```

- Anfrage mit **with**

```
with ALTER(Durchschnitt) as (
     select avg(Jahrgang) from WEINE)
select *
from   WEINE, ALTER
where  Jahrgang >= Durchschnitt - 2
  and  Jahrgang <= Durchschnitt + 2
```

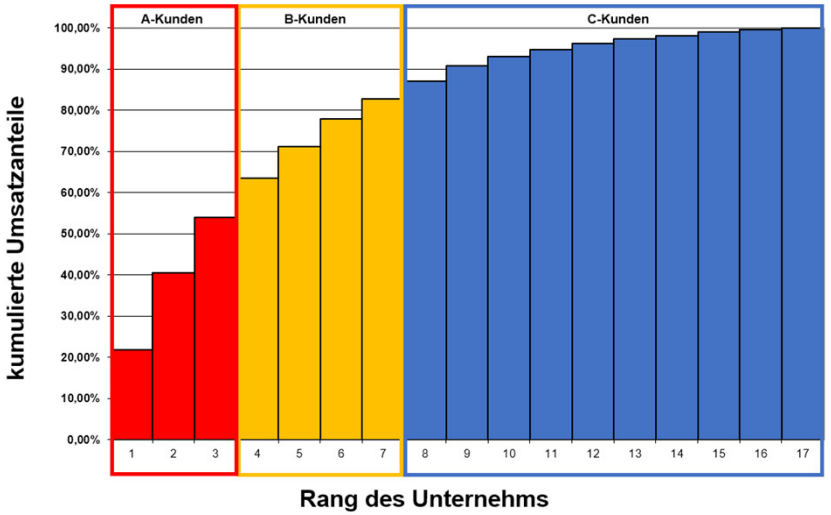
Interaktive Übung: ABC Analyse



Eine ABC Analyse trennt wichtiges von unwichtigem durch Gruppierung, z.B.:

Die besten Kunden die 80% vom Umsatz machen (A-Kunden), die die die nächsten 15% Umsatz (B-Kunden) machen und der Rest (C-Kunden)

Wie können Sie zu jedem Kunden angeben, ober ein A-, B-, oder C-Kunde ist?



Rekursive Anfragen (1)

- Typische Anwendung
  - Bill of Material-Anfragen
  - Berechnung der transitiven Hülle (z.B. Flugverbindungen)
  - Etc.

- Beispiel:
  - Busverbindungen in Oberbayern
  - Frage: Wo kann man überall von „Rosenheim“ aus mit dem Bus hinfahren?

BUSLINIE

Abfahrt	Ankunft	Distanz
Rosenheim	Wasserburg am Inn	27
Rosenheim	Kolbermoor	5
Kolbermoor	Großkarolinenfeld	6
Kolbermoor	Bad Aibling	7
Bad Aibling	Raubling	17

Rekursive Anfragen (2)

- Erster Versuch: Alle Busfahrten mit max. zweimalige Umsteigen

```
select Abfahrt, Ankunft
from   BUSLINIE
where  Abfahrt = 'Rosenheim'
  union
select B1.Abfahrt, B2.Ankunft
from   BUSLINIE B1, BUSLINIE B2
where  B1.Abfahrt = 'Rosenheim' and
       B1.Ankunft = B2.Abfahrt
  union
select B1.Abfahrt, B3.Ankunft
from   BUSLINIE B1, BUSLINIE B2, BUSLINIE B3
where  B1.Abfahrt = 'Rosenheim' and
       B1.Ankunft = B2.Abfahrt and
       B2.Ankunft = B3.Abfahrt
```

Rekursive Anfragen (3) - SQL:2003: `with recursive`-Klausel

```
with recursive rekursionstabelle as (
    select ...
    from tabelle
    where ...
      union all
    select ...
    from tabelle, rekursionstabelle
    where rekursionsbedingung
) [traversierungsklausel] [zyklusklausel]
anfrageausdruck
```

Initialisierung

Rekursiver Teil

Rekursionsschritt

Nicht-Rekursiver Teil

Rekursive Anfragen (4)

- Beispiel für Rekursion in SQL:2003

```
with recursive TOUR(Abfahrt, Ankunft) as (
    select Abfahrt, Ankunft
    from   BUSLINIE
    where  Abfahrt = 'Rosenheim'

      union all

    select T.Abfahrt, B.Ankunft
    from   TOUR T, BUSLINIE B
    where  T.Ankunft = B.Abfahrt)

select distinct * from TOUR
```

Initialisierung

Rekursiver Teil

Rekursionsschritt

Nicht-Rekursiver Teil

Rekursive Anfragen (5)

- Schrittweiser Aufbau der Rekursions-Tabelle **TOUR**
  - Initialisierung

Abfahrt	Ankunft
Rosenheim	Wasserburg am Inn
Rosenheim	Kolbermoor

  - Rekursions-Schritt (1. Iteration)

Abfahrt	Ankunft
Rosenheim	Wasserburg am Inn
Rosenheim	Kolbermoor
Rosenheim	Großkarolinenfeld
Rosenheim	Bad Aibling

  - Rekursions-Schritt (2. Iteration)

Abfahrt	Ankunft
Rosenheim	Wasserburg am Inn
Rosenheim	Kolbermoor
Rosenheim	Großkarolinenfeld
Rosenheim	Bad Aibling
Rosenheim	Raubling

Rekursive Anfragen (6)

- Beispiel: arithmetische Operationen im Rekursionsschritt

```
with recursive TOUR(Abfahrt, Ankunft, Strecke) as (

    select Abfahrt, Ankunft, Distanz as Strecke
    from   BUSLINIE
    where  Abfahrt = 'Rosenheim'

      union all

    select T.Abfahrt, B.Ankunft,
           Strecke + Distanz as Strecke
    from   TOUR T, BUSLINIE B
    where  T.Ankunft = B.Abfahrt)

select distinct * from TOUR
```

Sicherheit rekursiver Anfragen

- Sicherheit (= Endlichkeit der Berechnung) ist wichtige Anforderung an Anfragesprache

- Problem: Zyklen bei Rekursion

```
insert into BUSLINIE (Abfahrt, Ankunft, Distanz)
values ('Raubling', 'Kolbermoor', 12)
```

- 2 Möglichkeiten zur Behandlung in SQL
  1) Begrenzung der Rekursionstiefe
  2) Zyklen-Erkennung
     (seit SQL:2003 im Standard definiert, noch in keinem DBMS implementiert)

Sicherheit durch Einschränkung der Rekursionstiefe

- Beispiel: max. 2x Umsteigen

```
with recursive TOUR(Abfahrt, Ankunft, Umsteigen) as (

    select Abfahrt, Ankunft, 0
    from   BUSLINIE
    where  Abfahrt = 'Rosenheim'

      union all

    select T.Abfahrt, B.Ankunft, Umsteigen + 1
    from   TOUR T, BUSLINIE B
    where  T.Ankunft = B.Abfahrt and Umsteigen <= 2)

select distinct * from TOUR
```

Division (1)

Begriff Division

- Analogie zur arithmetischen Operation der ganzzahligen Division: Die ganzzahlige Division ist in dem Sinne die Inverse zur (ganzzahligen) Multiplikation, indem sie als Ergebnis die größte Zahl liefert, für die die Multiplikation mit dem Divisor kleiner ist als der Dividend.

- Analog gilt: $r = r_1 \div r_2$ ist die größte Relation, für die $r \bowtie r_2 \subseteq r_1$ ist.

Division (2) - Beispiel

- Beispiel - Relationen

WEIN_EMPFEHLUNG

Wein	Kritiker
La Rose GrandCru	Parker
Pinot Noir	Parker
Riesling Reserve	Parker
La Rose GrandCru	Clarke
Pinot Noir	Clarke
Riesling Reserve	Gault-Millau

GUIDES1

Kritiker
Parker
Clarke

GUIDES2

Kritiker
Parker
Gault-Millau

Division (3) - Beispiel

- Division mit erster Kritikerliste `WEIN_EMPFEHLUNG ÷ GUIDES1` liefert

Wein
La Rose GrandCru
Pinot Noir

- Division mit zweiter Kritikerliste `WEIN_EMPFEHLUNG ÷ GUIDES2` liefert

Wein
Riesling Reserve

Division (4) - Problem: All-Quantor

- Es-gibt-Quantor (implizit) über Selektion vorhanden.
  All-Quantor jedoch nicht erlaubt, aber nötig, z.B. für Division
- Lösung: kann in relationaler Algebra simuliert werden.
- Herleitung der Division aus Ω:
  Gegeben seine $r_1(R_1)$ und $r_2(R_2)$ mit $R_2 \subseteq R_1$, $R' = R_1 - R_2$. Dann ist

$$r_1 \div r_2 = r'(R') = \{ t \mid \forall\, t_2 \in r_2\ \exists\, t_1 \in r_1 : t_1(R') = t \wedge t_1(R_2) = t_2 \}$$

GUIDES1

Kritiker
Parker
Clarke

- Division von r_1 durch r_2

$$r_1 \div r_2 = \pi_{R'}(r_1) - \pi_{R'}((\pi_{R'}(r_1) \times r_2) - r_1)$$

WEIN_EMPFEHLUNG

Wein	Kritiker
La Rose GrandCru	Parker
Pinot Noir	Parker
Riesling Reserve	Parker
La Rose GrandCru	Clarke
Pinot Noir	Clarke
Riesling Reserve	Gault-Millau

Division (5) - Division in SQL

- Umsetzung der Allquantors (Division) in SQL:

- $r_1 \div r_2 = \pi_{R'}(r_1) - \pi_{R'}((\pi_{R'}(r_1) \times r_2) - r_1)$, d.h
  $WEIN\_EMPFEHLUNG \div GUIDES =$
  $\pi_{\{Wein\}}(WEIN\_EMPFEHLUNG) -$
  $\pi_{\{Wein\}}((\pi_{\{Wein\}}(WEIN\_EMPFEHLUNG) \times GUIDES) - WEIN\_EMPFEHLUNG)$

```
select Wein from WEIN_EMPFEHLUNG
   except
select w.Wein
from (
  select WEINE.wein as Wein, GUIDES.Kritiker as Kritiker
  from (
      select Wein from WEIN_EMPFEHLUNG) as WEINE, GUIDES
    except
  select * from WEIN_EMPFEHLUNG) as w
```

Division (6) - Division in SQL

- Alternativ: Simulation des Allquantors (Division) mit doppelter Negation:

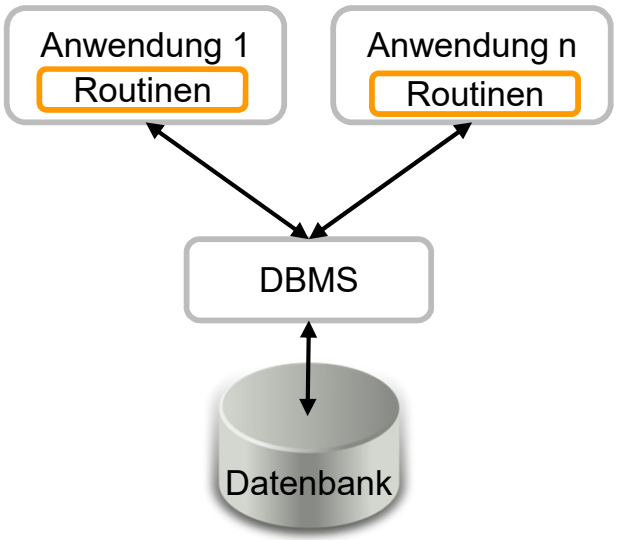
```
select distinct Wein
from   WEIN_EMPFEHLUNG w1
where  not exists (
  select * from GUIDES2 g
  where not exists (
    select * from WEIN_EMPFEHLUNG w2
    where w1.Wein = w2.Wein and
          g.Kritiker = w2.Kritiker))
```

- Sprachlich: „Gib alle die Weine aus, für die kein Kritiker existiert, der diesen Wein nicht empfohlen hat“.

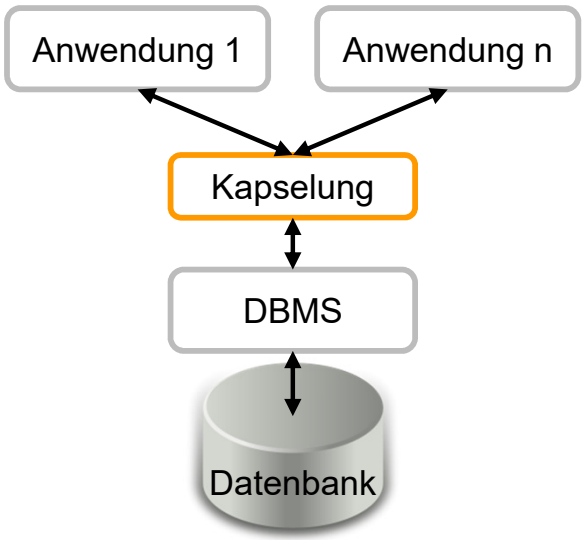
Integritätsbedingung

- Integritätsbedingung = Bedingung für die „Zulässigkeit“ („Korrektheit“) bezieht sich auf
  - (einzelne) Datenbankzustände
  - Zustandsübergänge
  - langfristige Datenbankentwicklungen

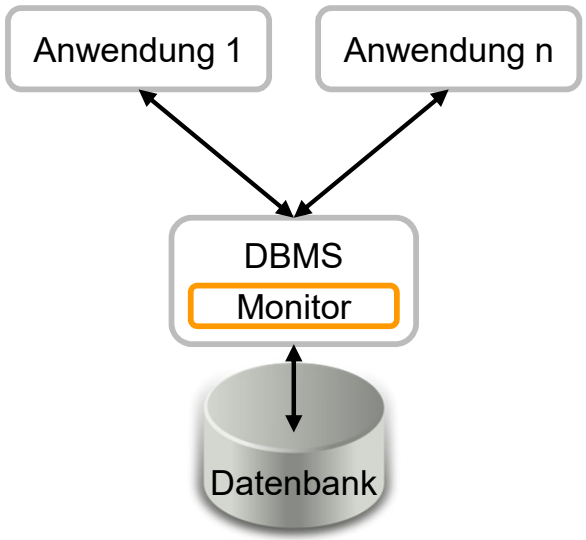
- Architekturen zur Integritätssicherung



Durch die Anwendung



Durch Einkapselung



Durch Monitor

Inhärente Integritätsbedingungen im Relationalen Modell

- Typintegrität
  - SQL erlaubt Angabe von Wertebereichen zu Attributen
  - Erlauben oder Verbieten von Nullwerten
  - Statements: `create domain, not null, default, check`
- Schlüsselintegrität
  - Angabe eines Schlüssels für eine Relation
  - Statement: `primary key …`
- Referenzielle Integrität
  - Angabe von Fremdschlüsseln
  - Statement: `foreign key … references …`

Typintegrität (1) – benutzerdefinierte Wertebereiche

- **`create domain`**: Festlegung eines benutzerdefinierten Wertebereichs
- Beispiel

```
create domain WeinFarbe varchar(4) default 'Rot'
   check (value in ('Rot', 'Weiß', 'Rose'))
```

- Anwendung

```
create table WEINE (
   WeinID int primary key,
   Name varchar(20) not null,
   Farbe WeinFarbe,
   ...)
```

Nicht in MySQL, nicht in TSQL

Typintegrität (2) – lokale Integritätsbedingungen

- **`check`**: Festlegung weiterer lokaler Integritätsbedingungen innerhalb der zu definierenden Wertebereiche, Attribute und Relationenschemata
- Beispiel: Einschränkung der zulässigen Werte
- Anwendung

```
create table WEINE (
   WeinID int primary key,
   Name varchar(20) not null,
   Jahr int check(Jahr between 1980 and 2010),
   ...)
```

Referenzielle Integrität

Erhaltung der referenziellen Integrität

- Überprüfung der Fremdschlüsselbedingungen nach Datenbankänderungen
- Für $\pi_A(r_1) \subseteq \pi_K(r_2)$, z.B. $\pi_{Verlagsname}(\text{BÜCHER}) \subseteq \pi_{Verlagsname}(\text{VERLAGE})$
  - Tupel t wird eingefügt in r_1
    ➔ überprüfen, ob $t' \in r_2$ existiert mit: $t'(K) = t(A)$, d.h. $t(A) \in \pi_K(r_2)$;

    falls nicht: abweisen
  - Tupel t' wird aus r_2 gelöscht
    ➔ überprüfen, ob $\sigma_{A=t'(K)}(r_1) = \emptyset$, d.h. kein Tupel aus r_1 referenziert t'

    falls nicht leer: abweisen oder Tupel aus r_1, die t' referenzieren, löschen (bei kaskadierendem Löschen)

Überprüfungsmodi von Bedingungen (1)

- **`on update | delete`**
  - Angabe eines Auslöseereignisses, das die Überprüfung der Bedingung anstößt
- **`cascade | set null | set default | no action`**
  - Kaskadierung: Behandlung einiger Integritätsverletzungen pflanzt sich über mehrere Stufen fort, z.B. Löschen als Reaktion auf Verletzung der referentieller Integrität
- **`deferred | immediate`** Nicht in MySQL, nicht in TSQL
  - legt Überprüfungszeitpunkt für eine Bedingung fest
  - `deferred`: Zurückstellen an das Ende der Transaktion
  - `immediate`: sofortige Prüfung bei jeder relevanten Datenbankänderung

Überprüfungsmodi von Bedingungen – Kaskadiertes Löschen

- Beispiel

```
create table WEINE (
   WeinID int primary key,
   Name varchar(50) not null,
   Preis float not null,
   Jahr int not null,
   Weingut varchar(30),
   foreign key (Weingut) references ERZEUGER (Weingut)
   on delete cascade)
```

Die `assertion`-Klausel

- Assertion: Prädikat, das eine Bedingung ausdrückt, die von der Datenbank immer erfüllt sein muss

- Syntax (SQL:2003)
  - `create assertion name check ( prädikat )`
  - Aber: in keinem aktuellen kommerziellen System implementiert!

- Beispiele:

```
create assertion Preise check
      ((select sum (Preis) from WEINE) < 10000)
create assertion Preise2 check
      (not exists (select * from WEINE where Preis > 200))
```

Nicht in MySQL, nicht in TSQL

Trigger

- Trigger: Anweisung/Prozedur, die bei Eintreten eines bestimmten Ereignisses automatisch vom DBMS ausgeführt wird

- Anwendungen
  - Erzwingen von Integritätsbedingungen („Implementierung“ von Integritätsregeln)
  - Auditing von DB-Aktionen
  - Propagierung von DB-Änderungen

- Spezifikation von
  - Ereignis (event) und Bedingung (condition) für Aktivierung des Triggers
  - Aktion(en) (action) zur Ausführung
  - Daher oft ECA-Regel genannt

- Verfügbar in den meisten kommerziellen Systemen (unterschiedliche Syntax)

Beispiel: Realisierung berechnetes Attribut durch zwei Trigger

- Einfügen von neuen Aufträgen:

```
create trigger AuftragszaehlungPLUS
on Auftrag
after insert
as begin
      update Kunde
      set AnzAuftraege = AnzAuftraege + 1
      where KNr in (select Knr from inserted)
end
```

- analog für Löschen von Aufträgen:

```
create trigger AuftragszaehlungMINUS
on Auftrag
after delete
as begin
      update Kunde
      set AnzAuftraege = AnzAuftraege - 1
      where KNr in (select KNr from deleted)
end
```

Trigger: Syntax im SQL Server (vereinfacht)

```
CREATE TRIGGER trigger_name
ON { table | view }
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
  AS { sql_statement  [ ; ] [ ,...n ] [ ; ] > }
```

- **FOR | AFTER**: Trigger wird nach dem auslösenden Statement ausgeführt
- **INSTEAD OF**: Trigger wird an Stelle des auslösenden Statements ausgeführt
- **INSERT, UPDATE, DELETE**: bei welchen Statements soll der Trigger ausgeführt werden
- **sql_statement**: auszuführende Trigger Aktion
  - Darf die speziellen Tabellen **deleted** und/oder **inserted** verwenden.

Trigger: Tabellen `inserted` und `deleted`

- deleted-Tabelle
  - Kopien der von DELETE- und UPDATE-Anweisungen betroffenen (=gelöschten) Zeilen
  - deleted-Tabelle und Triggertabelle enthalten i.allg. nicht die gleichen Zeilen.
- inserted-Tabelle
  - Kopien der durch INSERT- und UPDATE-Anweisungen betroffenen Zeilen
  - Während einer Einfügungs- oder Aktualisierungstransaktion werden sowohl der inserted-Tabelle als auch der Triggertabelle neue Zeilen hinzugefügt
  - Zeilen der inserted-Tabelle sind Kopien der neuen Zeilen in der Triggertabelle
- Aktualisierung (update) = Löschen + anschließendes Einfügen
  - alten Zeilen zunächst in deleted-Tabelle kopiert
  - anschließend neuen Zeilen in Triggertabelle und inserted-Tabelle kopiert

Weitere Beispiele für Trigger (1)

- Kein Kundenkonto darf unter 0 absinken:

```
create trigger bad_account
on Account
after update, insert
as begin
   if (exists (select *
               from inserted
               where Balance < 0) )
     rollback;
end
```

Weitere Beispiele für Trigger (2)

- Erzeuger müssen gelöscht werden, wenn sie keine Weine mehr anbieten:

```
create trigger unnützes_Weingut
on WEINE
after delete
as
  delete from ERZEUGER
  where Weingut in
       (select Weingut
        from deleted d
        where not exists (select *
                          from WEINE w
                          where w.Weingut = d.Weingut))
```

Integritätssicherung durch Trigger

1) Bestimme Objekt o_i, für das die Bedingung ϕ überwacht werden soll
   - i.d.R. mehrere o_i betrachten, wenn Bedingung relationsübergreifend ist
   - Kandidaten für o_i sind Tupel der Relationsnamen, die in ϕ auftauchen

2) Bestimme die elementaren Datenbankänderungen u_{ij} auf Objekten o_i, die ϕ verletzen können
   - Regeln: z.B. Existenzforderungen beim Löschen und Ändern prüfen, jedoch nicht beim Einfügen etc.

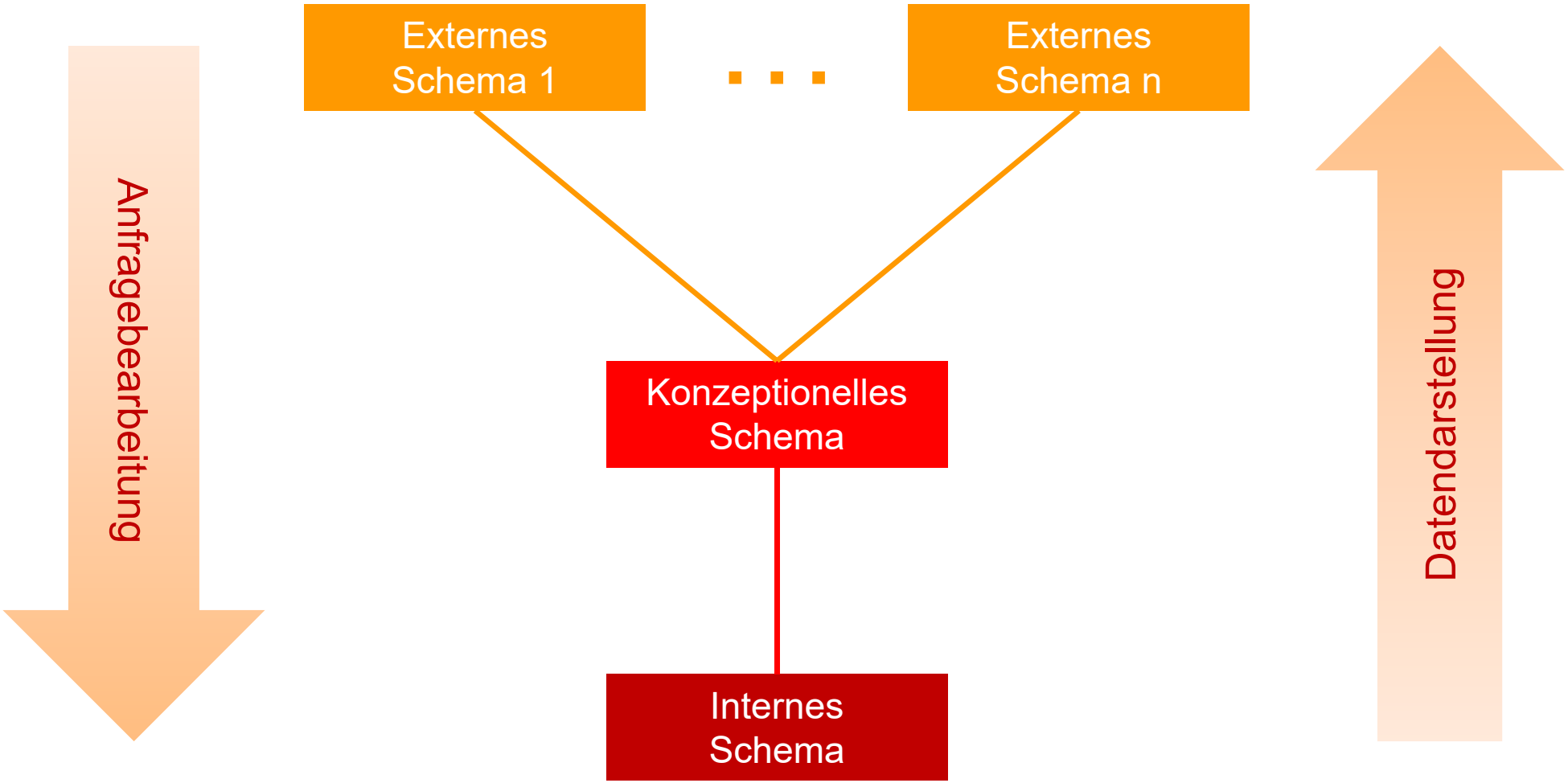
3) Bestimme je nach Anwendung die Reaktion r_i auf Integritätsverletzung
   - Rücksetzen der Transaktion (**`rollback`**)
   - korrigierende Datenbankänderungen

4) Formuliere folgende Trigger

```
create trigger t-phi-ij on o_i after u_ij
          as if(¬φ) begin r_i end
```

5) Wenn möglich, vereinfache entstandene Trigger

Schema-Architektur – allgemein (Wh)



Sichten (1)

- Sichten (englisch view): virtuelle Relationen (bzw. virtuelle Datenbankobjekte in anderen Datenmodellen)
- Sichten sind externe DB-Schemata folgend der 3-Ebenen-Schemaarchitektur
- Sichtdefinition
  - Relationenschema (implizit oder explizit)
  - Berechnungsvorschrift für virtuelle Relation, etwa SQL-Anfrage

Sichten (2)

- Vorteile
  - Vereinfachung von Anfragen für den Benutzer der Datenbank, etwa indem oft benötigte Teilanfragen als Sicht realisiert werden
  - Möglichkeit der Strukturierung der Datenbankbeschreibung, zugeschnitten auf Benutzerklassen
  - Logische Datenunabhängigkeit ermöglicht Stabilität der Schnittstelle für Anwendungen gegenüber Änderungen der Datenbankstruktur (entsprechend in umgekehrter Richtung)
  - Beschränkung von Zugriffen auf eine Datenbank im Zusammenhang mit der Zugriffskontrolle
- Herausforderungen
  - Automatische Anfragetransformation
  - Durchführung von Änderungen auf Sichten

Begriffe

- Anfrage: Folge von Operationen, die aus den Basisrelationen eine Ergebnisrelation berechnet
  - Ergebnisrelation interaktiv auf dem Bildschirm anzeigen oder
  - per Programm weiterverarbeiten („Einbettung“)
- Sicht: Folge von Operationen, die unter einem Sichtnamen langfristig abgespeichert wird und unter diesem Namen wieder aufgerufen werden kann. Ergibt eine **Sichtrelation.**
- Snapshot: Ergebnisrelation einer Anfrage, die unter einem Snapshot-Namen abgelegt wird, aber nie ein zweites Mal (mit geänderten Basisrelationen) berechnet wird (etwa Jahresbilanzen).

Definition von Sichten in SQL

- Sichtdefinition
  - Relationenschema (implizit oder explizit)
  - Berechnungsvorschrift für virtuelle Relation, etwa SQL-Anfrage

- Syntax in SQL

```
create view SichtName [ SchemaDeklaration ]
as SQLAnfrage
[ with check option ]
```

Sichten - Beispiel

- Alle Rotweine aus Bordeaux

```
create view BordeauxRotweine as
   select   Name, Jahrgang, WEINE.Weingut
   from     WEINE natural join ERZEUGER
   where    Farbe = 'Rot' and
            Region = 'Bordeaux'
```

Problembereiche bei Sichten

Zwei große Herausforderungen beim Sichtenkonzept

1) Automatische Anfragetransformation

2) Durchführung von Änderungen auf Sichten (engl. view update problem)
   - Projektionssichten
   - Selektionssichten
   - Verbundsichten
   - Aggregierungssichten

Anfragen auf Sichten

- SELECT Statement auf Sicht: Sicht wird durch ihre Definition ersetzt
  - Möglich wegen Orthogonalität von SQL (seit SQL92)
  - Resultierendes Statement wird vereinfacht und optimiert

- Beispiel

```
select *
from BordeauxRotweine
where Jahrgang = 2000
```

Wird zu

```
select *
from (  select   Name, Jahrgang, WEINE.Weingut
        from     WEINE natural join ERZEUGER
        where    Farbe = 'Rot' and
                 Region = 'Bordeaux') as BordeauxRotweine
where Jahrgang = 2000
```

SQL Anfragetransformation bei Verwendung von Sichten

SQL-Anfrage verwendet Sicht ➔ erforderliche Transformationen:

- Seit SQL92: geschachtelte **`select`**-Statements im **`from`**-Teil erlaubt
  - ➔ Transformation durch einfaches syntaktisches Ersetzen
- Vor SQL92: keine geschachtelten **`select`**-Statements im **`from`**-Teil erlaubt
  - ➔ Transformation durch „Mischen“
  - **`select`**: Sichtattribute evtl. umbenennen bzw. durch Berechnungsterm ersetzen
  - **`from`**: Namen der Originalrelationen
  - konjunktive Verknüpfung der **`where`**-Klauseln von Sichtdefinition und Anfrage (evtl. Umbenennungen)
  - Führte zu diversen Probleme bei Aggregierungssichten

Kriterien für Änderungen auf Sichten

- Effektkonformität
  - Benutzer sieht Effekt als wäre die Änderung auf der Sichtrelation direkt ausgeführt worden
- Minimalität
  - Basisdatenbank sollte nur minimal geändert werden, um den erwähnten Effekt zu erhalten
- Konsistenzerhaltung
  - Änderung einer Sicht darf zu keinen Integritätsverletzungen der Basisdatenbank führen
- Respektierung des Datenschutzes
  - Wird die Sicht aus Datenschutzgründen eingeführt, darf der bewusst ausgeblendete Teil der Basisdatenbank von Änderungen der Sicht nicht betroffen werden

Projektionssichten

- ING := $\pi_{WeinID,Name,Weingut}(WEINE)$
- In SQL:

```
create view ING as
   select WeinID, Name, Weingut
   from WEINE
```

- Änderungsanweisung für die Sicht ING:

```
insert into ING(WeinID, Name, Weingut)
   values (3333, 'Dornfelder', 'Müller')
```

- Korrespondierende Anweisung auf der Basisrelation WEINE:

```
insert into WEINE
   values (3333, 'Dornfelder', null, null, 'Müller')
```

Selektionssichten (1)

- IJ := $\sigma_{Jahrgang>2000}(\pi_{WeinID,Jahrgang}(WEINE)$
- In SQL:

```
create view IJ as
    select   WeinID, Jahrgang
    from     WEINE
    where    Jahrgang > 2000
```

- Tupelmigration: Ein Tupel

  `WEINE(3456, 'Zinfandel', 'Rot', 2004, 'Helena')`

  wird aus der Sicht „herausbewegt“:

```
update IJ
    set      Jahrgang = 1998
    where    WeinID = 3456
```

Selektionssichten (1)

- Kontrolle der Tupelmigration

```
create view IJ as
     select   WeinID, Jahrgang
     from     WEINE
     where    Jahrgang > 2000
     with check option
```

Verbundsichten (1)

- WE := WEINE □ ERZEUGER

- In SQL:

```
create view WE as
   select   WeinID, Name, Farbe, Jahrgang, WEINE.Weingut,
                 Anbaugebiet, Region
   from     WEINE, ERZEUGER
   where    WEINE.Weingut = ERZEUGER.Weingut
```

- Änderungsoperationen in der Regel nicht eindeutig übersetzbar:

```
insert into WE
   values (3333, 'Dornfelder', 'Rot', 2002, 'Helena',
           'Barossa Valley', 'South Australia')
```

Verbundsichten (2)

- Änderung wird transformiert zu

```
insert into WEINE
  values (3333, 'Dornfelder', 'Rot', 2002, 'Helena')
```

- Plus
  - Entweder: Einfügeanweisung auf ERZEUGER:

```
insert into ERZEUGER
  values ('Helena', 'Barossa Valley', 'South Australia')
```

  - oder alternativ:

```
update ERZEUGER
  set Anbaugebiet='Barossa Valley',Region='South Australia'
  where Weingut='Helena'
```

    - besser bzgl. Minimalitätsforderung
    - widerspricht aber Effektkonformität!

Aggregierungssichten

- Beispiel in SQL:

```
create view FM (Farbe, MinJahrgang) as
    select      Farbe, min(Jahrgang)
    from        WEINE
    group by    Farbe
```

- Folgende Änderung ist nicht eindeutig umsetzbar:

```
update FM
    set     MinJahrgang = 1993
    where   Farbe = 'Rot'
```

Zusammenfassung der Problembereiche

- Verletzung der Schemadefinition (z.B. Einfügen von Nullwerten bei Projektionssichten)
- Datenschutz: Seiteneffekte auf nicht-sichtbaren Teil der Datenbank vermeiden (Tupelmigration, Selektionssichten)
- nicht immer eindeutige Transformation: Auswahlproblem
- Aggregierungssichten (u.a.): keine sinnvolle Transformation möglich
- elementare Sichtänderung soll genau einer atomaren Änderung auf Basisrelation entsprechen: 1:1-Beziehung zwischen Sichttupeln und Tupeln der Basisrelation (kein Herausprojizieren von Schlüsseln)

Rechtevergabe in Datenbanksystemen

- Zugriffsrechte: WER – WOMIT – WAS

```
(AutorisierungsID, DB-Ausschnitt, Operation)
```

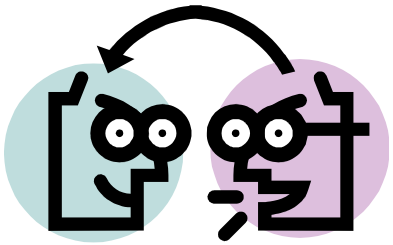
- `AutorisierungsID` ist interne Kennung eines „Datenbankbenutzers“
- `DB-Ausschnitt`: Relationen und Sichten
- `Operation`: Lesens, Einfügen, Ändern, Löschen

Rechtevergabe in SQL

- Syntax

```
grant <Rechte>
      on <Tabelle>
      to <BenutzerListe>
      [with grant option]
```

- Erläuterungen:
  - In `<Rechte>`-Liste: `all` bzw. Langform `all privileges` oder Liste aus `select`, `insert`, `update`, `delete`
  - Hinter `on`: Relationen- oder Sichtname
  - Hinter `to`: Autorisierungsidentifikatoren (auch `public`, `group`)
  - Spezielles Recht: Recht auf die Weitergabe von Rechten (`with grant option`)



Beispiel

- Autorisierung für public: „Jeder Benutzer kann seine Aufträge sehen und neue Aufträge einfügen (aber nicht löschen!).“

```
create view MeineAufträge as
    select *
    from AUFTRAG
    where KName = user;
```

```
grant select, insert
    on MeineAufträge
    to public;
```

Zurücknahme von Rechten

- Syntax

```
revoke <Rechte>
     on <Tabelle>
     from <BenutzerListe>
     [restrict | cascade ]
```

- Erläuterungen:
  - `restrict`: Falls Recht bereits an Dritte weitergegeben: Abbruch von `revoke`
  - `cascade`: Rücknahme des Rechts mittels `revoke` an alle Benutzer propagiert, die es von diesem Benutzer mit `grant` erhalten haben

Rollenmodell ab SQL:2003

- Zur einfacheren Verwaltung von Rechten wurden ab SQL:2003 Rollen eingeführt
- Statt Nutzern direkt Rechte zu geben, werden Rechte an Rollen vergeben und Rollen an Nutzer → einfacherer Transfer bei Wechsel einer Person
- Beispiel

```
create role weindb_admin_role;
grant weindb_admin_role to gunter;
grant select
    on WEINE
    to public;
grant all
    on WEINE
    to weindb_admin_role;
```

Rechtevergabe in kommerziellen DBMS

- Rechtevergabe in kommerziellen DBMS ist deutlich komplexer
- Grundlage sind meist weiterhin die Konzepte Nutzer und Rolle
- Häufig Einbindung in andere Rechtesysteme
  - Nutzer des Betriebssystems (z.B. Windows-Nutzer)
  - Einbindung in Identitätsmanagementsysteme (z.B. Active Directory)
  - Eigene Nutzerverwaltung des DBMS
  - ➔ Oft sind mehrere solcher Systeme gleichzeitig/parallel im Einsatz
- Beispiel
  - Sie melden sich in der Übung mit der „SQL Server Kennung“ an, die Sie in der ersten Übungsgruppe bekommen haben
  - Gleichzeitig meldet sich unser Systemadministrator mit seiner Windows-Kennung an, die von einem Domain Kontroller verwaltet wird

Kapitel 11: SQL/PSM

In diesem Kapitel wollen wir folgende Fragen betrachten

- Was sind Stored Procedures und Stored Functions?
- Warum verwendet man sie?
- Wie schreibt man sie?
- Welche Sprachkonstrukte werden von SQL/PSM unterstützt?

Literatur: CompleteBook Chap 9.4 ; Biberbuch Kap 13.5

Kapitel 11: SQL/PSM

ACHTUNG:
Alle Statements diesem Kapitel verwenden die MS SQL Server Syntax, nicht den ANSI SQL Standard

SQL/PSM - Motivation

- Probleme von Client-Server Systemen („Embedded SQL“ und „CLI“):
  - Ständiger Wechsel der Ausführungskontrolle zwischen Anwendung (=Client) und DBS (=Server)
  - Keine anweisungsübergreifende Optimierung möglich
- Lösung: gespeicherte Prozeduren (stored procedures)
  - Im DBMS verwaltete & ausgeführte Software-Module (Prozeduren/Funktionen)
  - Aufruf aus Anwendungen und Anfragen / Aktionsteilen von Triggern
- Vorteile von stored procedures
  - Strukturierungsmittel für größere Anwendungen: redundanzfreie Darstellung relevanter Aspekte der Anwendungsfunktionalität
  - Prozeduren nur vom DBMS abhängig
  - Optimierung der Prozeduren
  - Ausführung der Prozeduren unter Kontrolle des DBMS
    - Rechtevergabe für Prozeduren

SQL/PSM: Der Standard

- ANSI Standard für prozedurale Erweiterungen: SQL/PSM (Persistent Stored Modules)
- Bestandteile:
  - Gespeicherte Module aus Prozeduren und Funktionen
  - Einzelroutinen
  - Einbindung externer Routinen (implementiert in C, Java, . . .)
  - Syntaktische Konstrukte für Schleifen, Bedingungen etc.
- Umsetzung (mehr oder weniger konform) in allen aktuellen DBMS
  - Oracle: PL/SQL
  - IBM DB2: sehr nah an SQL/PSM
  - Informix: SPL
  - Microsoft SQL Server: Transact-SQL
  - MySQL, PostgreSQL: nah an SQL/PSM

Kapitel 11: SQL/PSM

Variablen

- Variablen enthalten einen einzelnen Datenwert eines bestimmten Typs
- Variablennamen müssen mit mit einem @ Zeichen beginnen
- Deklaration von Variablen (mit optionaler Initialisierung)

```
DECLARE @local_variable [AS] data_type  [ = value ] [;]
```

- Zuweisung von Werten an Variablen

```
SET @local_variable = expression [;]
```

Beachte: `null` ist als Wert für `value` und `expression` möglich.

- Beispiel

```
DECLARE @i int = 0;
SET @i = 10;
```

Zuweisung von Variablen im SELECT

- Variablen können mittels “=„ hinter SELECT belegt werden
- Liefert das SELECT mehrere Tupel zurück, werden die Werte des letzten Tupels verwendet (das ist aber schlechter Stil)
- Beispiele

```
DECLARE @WeinName NVARCHAR(100)
DECLARE @WeinFarbe NVARCHAR(100)

SELECT @WeinName = name, @WeinFarbe = Farbe
FROM Weine
WHERE WeinID = 2171
```

```
DECLARE @AnzHelena INT

SELECT @AnzHelena = COUNT(*)
FROM Weine
WHERE Weingut = 'Helena'
```

Ablaufsteuerung

- Gruppierung von Statements (zu einem Statement-Block)

```
BEGIN
  { sql_statement | statement_block }
END
```

- Bedingte Ausführung

```
IF Boolean_expression
  { sql_statement | statement_block }
[ ELSE
  { sql_statement | statement_block } ];
```

- PRINT-Statement – Ausgabe auf dem Bildschirm

```
PRINT <string_expression> [;]
```

Funktion

- „A user-defined function is a Transact-SQL […] routine that accepts parameters, performs an action, such as a complex calculation, and returns the result of that action as a value.“ (Quelle: SQL Server Documentation)

- Erzeugen einer Funktion (vereinfacht)

```
CREATE FUNCTION [ schema_name.] function_name
( [ { @parameter_name [ AS ] parameter_data_type  [ = default ]}
    [ ,...n ]
  ]
)
RETURNS return_data_type
[ AS ]
BEGIN
   function_body
   RETURN scalar_expression
END [ ; ]
```

Funktion - Beispiel

- Definition: Funktion, um eine Zahl um einen Prozentsatz zu erhöhen

```
CREATE FUNCTION addPercent(@value NUMERIC(10,2), @percent INT)
RETURNS NUMERIC(10,2)
BEGIN
  RETURN @value * (1.0 + CAST(@percent AS NUMERIC(10,2)) / 100.0)
END
```

- Aufrufe der Funktion

```
SELECT dbo.addPercent(4.99, 50) AS NeuerPreis
```

```
UPDATE Weine SET Preis = dbo.addPercent(Preis, 50)
WHERE Weingut='Creek'
```

Funktion – komplexeres Beispiel

Beispiel: Funktion, die einen Wein als billig, akzeptabel oder teuer bewertet.

```
CREATE FUNCTION bewertung( @preis NUMERIC(10,2) )
RETURNS NVARCHAR(50)
BEGIN
  DECLARE @bewertet NVARCHAR(50);

  IF @preis < 5.00
    SET @bewertet = 'billiger';
  ELSE IF @preis < 20.00
    SET @bewertet = 'akzeptabler';
  ELSE
    SET @bewertet = 'teurer';

  SET @bewertet = @bewertet + ' Wein';
  RETURN @bewertet;
END
```

```
PRINT dbo.bewertung(18.00);
```

Prozedur

- „Accept input parameters and **return multiple values** in the form of **output parameters** to the calling procedure or batch. Contain programming statements that perform operations in the database, including calling other procedures. Return a status value to a calling procedure or batch to indicate success or failure (and the reason for failure).“ (Quelle: SQL Server Documentation)

- Erzeugen einer Prozedur (vereinfacht)

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name
  [ { @parameter data_type } [ = default ] [ OUT | OUTPUT ] [ ,...n ]
AS { [ BEGIN ]
  sql_statement [;] [ ...n ]
[ END ] }  [;]
```

OUT | OUTPUT

Indicates that the parameter is an output parameter. Use OUTPUT parameters to return values to the caller of the procedure. (Quelle: SQL Server Documentation)

Prozedur - Beispiel

- Prozedur um alle Weine eines Weinguts prozentual zu erhöhen.
  Rückgabe: Anzahl Weine, die erhöht wurden und neuer maximaler Preis.

```
CREATE PROCEDURE increasePrices
    @myWeingut NVARCHAR(20),
    @percent INT,
    @count INT OUT,
    @newMax NUMERIC(10,2) OUT
AS BEGIN
  UPDATE Weine SET Preis = dbo.addPercent(Preis, @percent)
  WHERE Weingut = @myWeingut;

  SELECT @count = COUNT(*), @newMax = MAX(Preis)
  FROM Weine
  WHERE Weingut = @myWeingut;
END
```

Aufruf von Prozeduren

- Ausführen einer Prozedur (vereinfacht)

```
[ { EXEC | EXECUTE } ] module_name
    [ [ @parameter = ] { value | @variable [ OUTPUT ] } ] [ ,...n ]
  [;]
```

- Beispiel

```
DECLARE @Anzahl INT
DECLARE @Teuerster NUMERIC(10,2)
DECLARE @UmProzent INT = 30

EXEC dbo.increasePrices 'Helena', @UmProzent, @Anzahl OUT,
     @Teuerster OUT

-- mögliche Alternative:
-- EXECUTE dbo.increasePrices @count=@Anzahl OUT,
--   @newMax=@Teuerster OUT, @myWeingut='Helena', @percent=@UmProzent

PRINT 'Helena hat ' + CAST(@Anzahl AS NVARCHAR(10)) + ' Weine, ' +
  'der teuerste kostet ' + CAST(@Teuerster AS NVARCHAR(10))+ ' EUR'
```

Interaktive Übung

```
with
    ABC_Punkte(A,B,C) as(
        select  sum(bestellwert)*0.8,
                sum(bestellwert)*0.95,
                sum(bestellwert)
        from bestellung),

    Kundenumsatz(kunde, id, umsatz) as(
        select  kunde.name,
                kunde.id,
                isnull(sum(bestellung.bestellwert),0) as umsatz
        from    kunde left join bestellung on (kunde.id=bestellung.kundennummer)
        group by kunde.name, kunde.id
    ),
   ABC_Kunden (kunde, umsatz, kumuliert, typ) as(
     select  K.kunde,
             K.umsatz,
             (Select sum(V.umsatz) from Kundenumsatz as V where V.umsatz>=K.umsatz),
             case
                 when (Select sum(V.umsatz) from Kundenumsatz as V where V.umsatz>=K.umsatz)<=(select A from ABC_Punkte) then 'A'
                 when (Select sum(V.umsatz) from Kundenumsatz as V where V.umsatz>=K.umsatz)<=(select B from ABC_Punkte) then 'B'
                 when (Select sum(V.umsatz) from Kundenumsatz as V where V.umsatz>=K.umsatz)<=(select C from ABC_Punkte) then 'C'
             end
      from    Kundenumsatz as K
   )

select * from ABC_Kunden order by umsatz desc
```

- Erzeugen Sie stattdessen

  `getABClevel(level char)`, `getKundenumsatz(id int)`,
  `getKumulierterUmsatz(id int)` `getCustomerClassification(id int)`

Kapitel 11: SQL/PSM

Schleifen

- WHILE-Schleife

```
WHILE Boolean_expression
  { sql_statement | statement_block | BREAK | CONTINUE }
```

`BREAK`
Beendet die **innersten** WHILE-Schleife.

`CONTINUE`
Startet die die WHILE-Schleife neu (alle Anweisungen nach CONTINUE werden ignoriert).

SELECT-Queries in PSM

1) Single-Value Queries: zwei Möglichkeiten

```
DECLARE @AnzRotweine INT
SET @AnzRotweine = (SELECT COUNT(*) FROM Weine WHERE Farbe='Rot')
-- oder
SELECT @AnzRotweine = COUNT(*) FROM Weine WHERE Farbe='Rot'
```

2) Single-Row Queries: Werte mittels „=„ im SELECT zuweisen

```
DECLARE @meinName NVARCHAR(100);
DECLARE @meineFarbe NVARCHAR(20);
SELECT @meinName = Name, @meineFarbe = Farbe
  FROM Weine WHERE WeinID=4711
```

3) Mehrere Ergebnis-Tupel: Verwendung von Cursor

Cursor - „Tupel-Iterator“ für ein Query

- Deklaration des Cursors mittels

```
DECLARE cursor_name CURSOR LOCAL FOR select_statement [;]
```

- Vor Verwendung: Öffnen des Cursors

```
OPEN cursor_name
```

- Nach Verwendung: Schließen und Deallozieren des Cursors

```
DEALLOCATE cursor_name
```

- Holen des nächsten Tupels in Variablen und Fortbewegen des Cursors:

```
FETCH [ [ NEXT | PRIOR | FIRST | LAST |
    ABSOLUTE { n | @nvar } | RELATIVE { n | @nvar } ]
FROM ]   cursor_name } INTO @variable_name [ ,...n ]
```

@@FETCH_STATUS
Enthält den Status der letzten FETCH-Anweisung (0 = alles ok, sonst Fehler)

WHILE-Schleife mit CURSOR - Beispiel

- Berechnung durchschnittlicher Preis und Varianz eines Weinguts

```
CREATE PROCEDURE meanVar(@Weingut NVARCHAR(100), @mean REAL OUT, @var REAL OUT)
AS BEGIN
  DECLARE @dieserPreis NUMERIC(10,2), @anzWeine INTEGER;
  DECLARE WeinCursor CURSOR LOCAL FOR SELECT Preis FROM Weine WHERE Weingut=@Weingut;
  SET @mean = 0.0;
  SET @var = 0.0;
  SET @anzWeine = 0;
  OPEN WeinCursor;
  FETCH NEXT FROM WeinCursor INTO @dieserPreis;
  WHILE @@FETCH_STATUS = 0 BEGIN
    SET @anzWeine = @anzWeine + 1;
    SET @mean = @mean + @dieserPreis;
    SET @var = @var + @dieserPreis * @dieserPreis;
    FETCH NEXT FROM WeinCursor INTO @dieserPreis;
  END
  DEALLOCATE WeinCursor;
  SET @mean = @mean / @anzWeine;
  SET @var = @var / @anzWeine - @mean * @mean;
END
```

Zusammenfassung SQL/PSM

- SQL/PSM =
  mächtige Möglichkeit „im DBS zu programmieren“
- Vollständige Programmiersprache
  - mit allen diesbezüglichen Vor- und Nachteilen
- Ermöglicht
  - Erhebliche Performance Steigerungen
  - Besser strukturierten Code