Technische Hochschule Rosenheim

Chapter 7 – Advanced SQL

Databases lectures

Dr Kai Höfig

Overview

Chapter 7: Advanced SQL

- Aggregation and grouping
- Joins
- Sorting, top-k queries and null values
- Named and recursive expressions
- Division
- Integrity
- Triggers
- Views
- Access control
- PSM

SQL core

- **`select`** projection list
  arithmetic operations & aggregate functions
- **`from`** relations to be used, possible renaming
- **`where`** selection conditions, join conditions
  nested queries (once again an SFW block)
- **`group by`** grouping for aggregate functions
- **`having`** selection conditions for groups
- **`order by`** output order
- Calculation sequence:
  `from`, **`where`**, **`group by`**, **`having`**, **`select`**, **`order by`**

Aggregate functions and grouping

- Aggregate functions calculate new values for an entire column, such as the sum or the average of the values in a column

- Examples:
  - Determination of the average price of all items or the total sales of all products sold
  - If grouping is also utilised: calculation of functions per group, e.g. the average price per product group or the total sales per customer

- Simple example: calculating the total number of wines

```
select count(*) as Number
from WINES
```

Results:

Number of
7

Aggregate functions in standard SQL (1)

- `count`: calculates the number of values in a column or alternatively (in the special case `count(*)`) the number of tuples in a relation

- `sum`: calculates the sum of the values in a column (only for numerical value ranges)

- `avg`: calculates the arithmetic mean of the values in a column (only for numerical value ranges)

- `max` or `min`: calculate the largest or smallest value in a column

- Arguments of an aggregate function
  - an attribute of the relation specified by the `from` clause,
  - a valid scalar expression or
  - in the case of the `count` function, also the `*` symbol

Aggregate functions in standard SQL (1)

- Before the argument (except in the case of **`count(*)`**) optionally also the keywords **`distinct`** or **`all`**
  - **`distinct`**: before using the aggregate function, the duplicate values are eliminated from the set of values to which the function is applied
  - **`all`**: duplicates are included in the calculation (default setting)
  - null values are eliminated from the set of values before the function is applied
    Exception: **`count(*)`**

Aggregate functions – examples

- Number of different wine regions:

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

- Wines that are older than average:

```
select Name, Vintage
from   WINES
where  Vintage < ( select avg(Vintage) from WINES )
```

Aggregate functions - nesting

- Nesting of aggregate functions is <u>not</u> permitted

- **<u>Incorrect</u>** example:

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

- Possible correct formulation:

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

Aggregate functions in the `where` clause

- Aggregate functions only return one value
  → can be used in constant selections of the `where` clause

- Example: all vineyards that only deliver one wine:

```
select *
from   PRODUCER e
where  1 = (
       select count(*)
       from   WINES w
       where  w.Vineyard = e.Vineyard)
```

Grouping using `group by`

- Grouping using **`group by`** allows the summarising of tuples

- Example: number of wines by colour

```
select    Colour, count(*) as Number
from      WINES
group by  Colour
```

Results:

Colour	Number
Red	5
White	2

Selection of groups using **having**

- Selection of the grouping using **having** is possible
  - not possible in **where** due to the calculation sequence

- Example: regions with more than one wine

```
select    Region, count(*) as Number
from      PRODUCER natural join WINES
group by  Region
having    count(*) > 1
```

Results:

Region	Number
South Australia	2
California	3

Grouping: schematic sequence (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
…	…	…	…

Calculation sequence:
from, **where**, **group by**, **having**, **select**, **order by**

- Query:

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

Grouping: schematic sequence (2)

Grouping: Step 1

- **`from`** and **`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
…	…	…	…

Internal table in the DBMS, not visible externally!

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

Grouping: Step 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

Internal tables in the DBMS, not visible externally!

Grouping: schematic sequence (4)

Grouping: Step 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

Internal tables in the DBMS, not visible externally!

Grouping: schematic sequence (5)

Grouping: Step 4

- `select A, sum(D) as D_TOTAL`

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

Internal table in the DBMS, not visible externally!

A	D_TOTAL
1	9
4	1

Results table

Grouping

- Grouping operator γ:

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

- Adds new attributes to the attribute schema of $r(R)$ that correspond with the function applications $f_1(x_1)$, $f_2(x_2)$, …, $f_n(x_n)$
- Application of the functions $f_i(\ x_i\ )$ to the subset of the tuples of $r(R)$ that have the same attribute values for the A attributes
- In SQL:

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

- Formal semantics: see literature

Attributes for aggregation or `having`

- Permitted attributes after `select` when grouping of relation with schema R
  - Grouping attributes G
  - Aggregations of non-grouping attributes R - G.
- Permitted attributes after `having`
  - Grouping attributes G
  - Aggregations of non-grouping attributes R - G.

➔ **Common error although actually completely logical, otherwise pushing things together doesn't work anymore. If further restrictions are to be implemented, see the where section.**

Natural Join

- Natural join: equality condition for all attributes with the same name

- Example:

```
select     Name, Growing_area
from       WINES natural join PRODUCER
```

Do we really want the database system to guess the join for us?

In MySQL, not in TSQL

Equi Join

- Equi join: equality condition for explicitly specified and possibly different attributes

In MySQL, in TSQL

- Examples:

```
select      Name, Growing_area
from        WINES join PRODUCER
                  on (WINES.Vineyard = PRODUCER.Vineyard)
```

```
select      CUSTOMER.Name as Customer_name, WINES.Growing_area,
            WINES.Name as Wine_name
from        WINES join CUSTOMER
                  on (WINES.Growing_area =
CUSTOMER.Favourite_area)
```

Theta Join

- Theta join (θ join): any join condition

- Example:

In MySQL, in TSQL

```
Select      CUSTOMER.Name as Customer_name, WINE.Name as
Wine_name,
            Price
from        WINES join CUSTOMER
                  on (WINES.Price <= CUSTOMER.MaxPrice)
```

Customer_name	Wine_name	Price
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
...		

Semi Join

- Semi join: only attributes of an operand appear in the result
  - Purpose: elimination of dangling tuples
  - Left semi join: only attributes of the left operand appear in the result
  - Right semi join: only attributes of the right operand appear in the result
  - No explicit implementation in SQL, but very easy to implement by specifying the attributes after SELECT DISTINCT

- Example of a left semi join:

```
select distinct PRODUCER.*
from   PRODUCER e join WINES w on (e.Vineyard = w.Vineyard)
```

Vineyard	Growing_area	Region
Creek	Barossa Valley	South Australia
Helena	Napa Valley	California
Chateau La Rose	Saint-Emilion	Bordeaux
Müller	Rheingau	Hesse
Bighorn	Napa Valley	California

In MySQL, in TSQL

The simple joins in TSQL

- Natural join by explicitly stating the join condition, no explicit implementation exists.
- Equi join as well
- Theta join as well
- Semi join as well

Example relations for orders

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

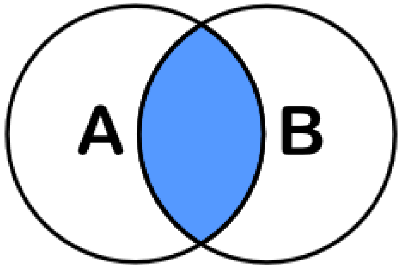
Inner join – the intersection

- **Only the A** and **only the B** for which information is present in **both** tables.
- In this example, there are neither customers without an order nor orders without a customer.

```
SELECT  customer.name, order.id, order.date
FROM    customer, order
WHERE   customer.id = order.customernumber
```

or

```
SELECT      customer.name, order.id, order.date
FROM        customer
INNER JOIN  order
ON          customer.id = order.customernumber
```




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

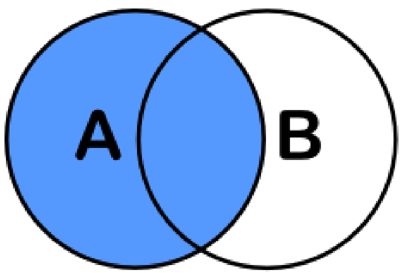
name	id	datum
Michaela	1	2018-04-21
Michaela	2	2018-03-11
Michaela	3	2018-04-21
Deike	4	2018-05-11
Deike	5	2018-06-03
Herbert	6	2018-07-04
Herbert	7	2018-02-05

Left join – all

- **All from A** with the information from B (if present)
- In the example, this includes all customers, even those who do not have an order, with the information from the *Order* table.

```
SELECT      customer.name, order.id, order.date
FROM        customer
LEFT JOIN   order
ON          customer.id = order.customernumber
```

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



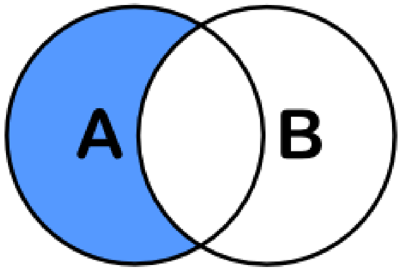
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key

Left join – only the complement

- **Only the A** that do **not have anything in B**
- In the example, this includes all customers who do not have an order yet.

```
SELECT      customer.name, order.id, order.date
FROM        customer
LEFT JOIN   order
ON          customer.id = order.customernumber
WHERE       order.customernumber IS NULL
```

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



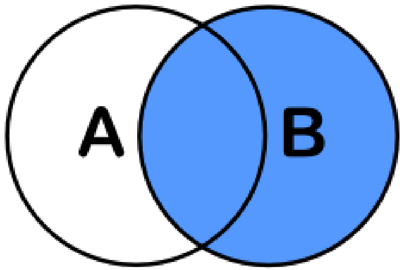
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL

Right join – all

- **All from B** with the information from A (if present)
- In the example, this includes all orders, even those which do not have a customer, with the information from the *Order* table.

```
SELECT      customer.name, order.id, order.date
FROM        customer
RIGHT JOIN  order
ON          customer.id = order.customernumber
```

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
```

Right join – all

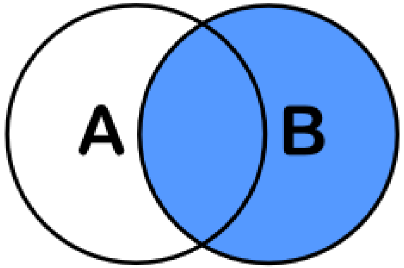
- **All from B** with the information from A (if present)
- In the example, this includes all orders, even those which do not have a customer, with the information from the *Order* table.

```
SELECT      customer.name, order.id, order.date
FROM        customer
RIGHT JOIN  order
ON          customer.id = order.customernumber
```

or

```
(
SELECT      customer.name, order.id, order.date
FROM        order
LEFT JOIN   customer
ON          customer.id = order.customernumber
)
```

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



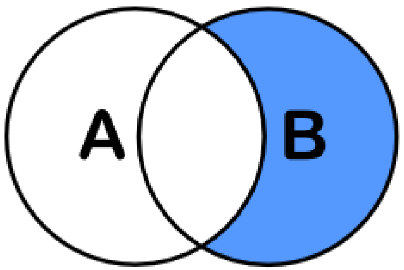
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key

Every left join can also be represented as a right join and vice versa

Right join – only the complement

- **Only the B** that do **not have anything in A**
- In the example, this includes all orders which do not have a customer.

```
SELECT      customer.name, order.id, order.date
FROM          customer
RIGHT JOIN   order
ON            customer.id = order.customernumber
WHERE        order.customernumber IS NULL
```




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

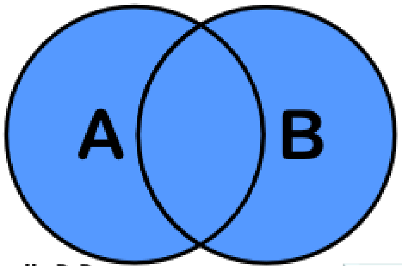
name	id	datum
NULL	8	2018-03-06

Full Outer Join

- All of A and all of B, each with the information from A and B, if present
- In the example, this includes all customers, even those who do not have an order, and all orders, even those which do not have a customer.

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           customer.name, order.id, order.date
FROM             customer
FULL OUTER JOIN  order
ON               customer.id = order.customernumber
```




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

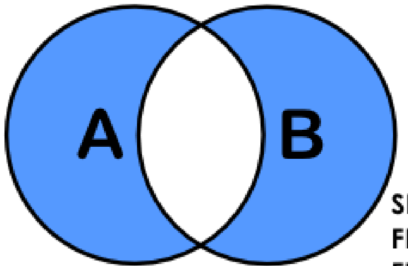
Not in MySQL, in TSQL

Full Outer Join

- All of A and all of B, each with the information from A and B, if present
- In the example, this includes all customers, even those who do not have an order, and all orders, even those which do not have a customer.

```
SELECT           customer.name, order.id, order.date
FROM             customer
FULL OUTER JOIN  order
ON               customer.id = order.customernumber
WHERE            customer.id IS NULL OR order.id IS NULL
```

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

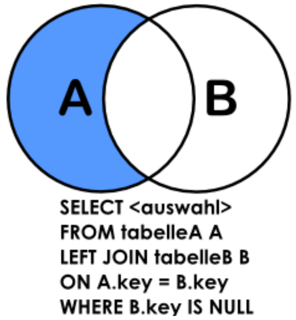


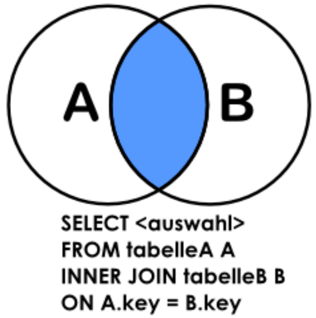

```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

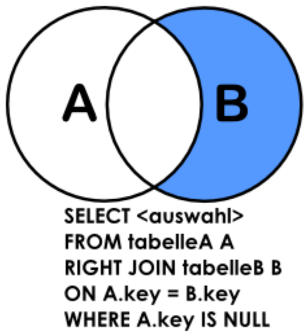
Not in MySQL, in TSQL

Constructing an outer join

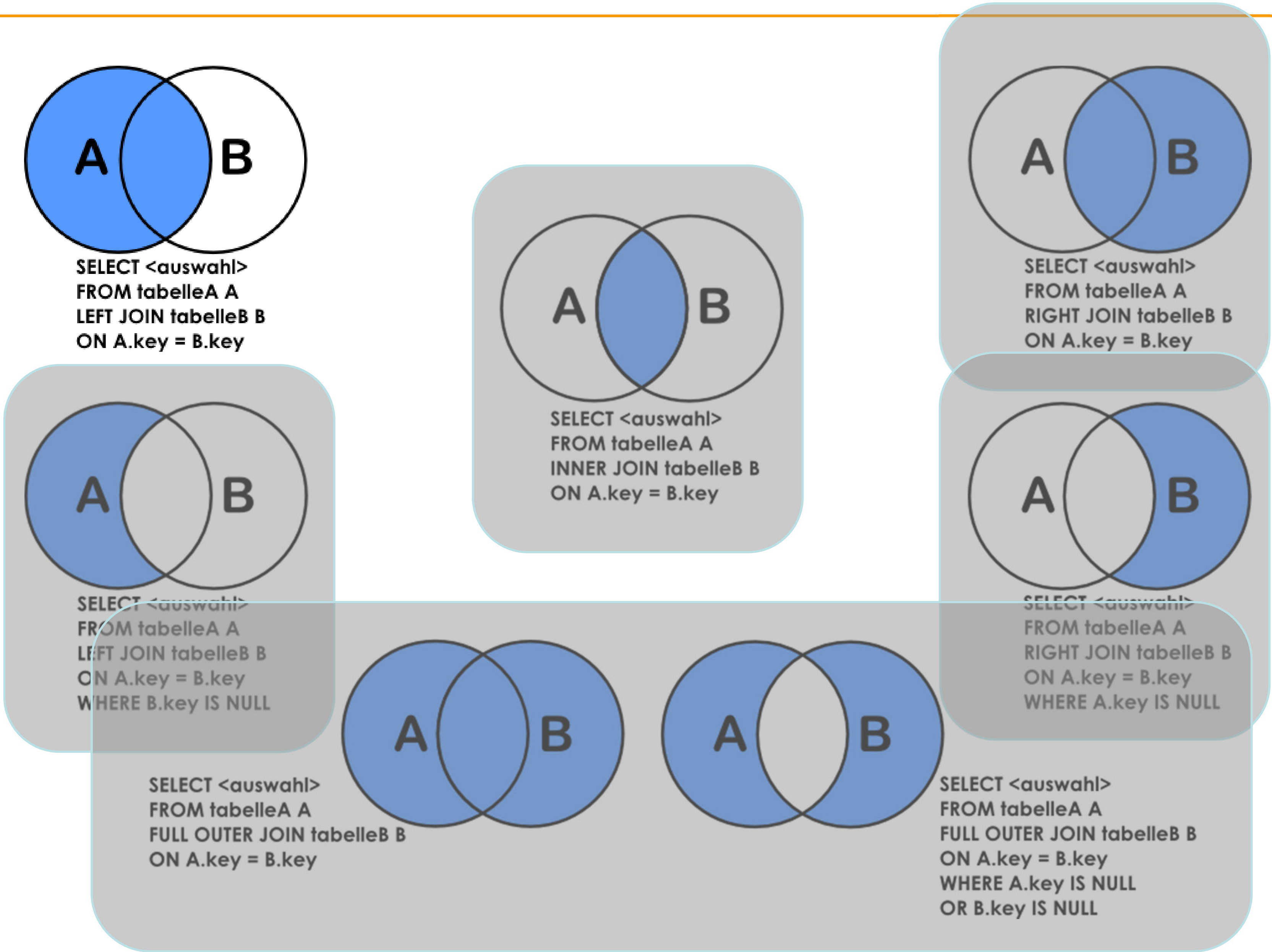
```
SELECT      customer.name, order.id, order.date
FROM        customer
LEFT JOIN   order
ON          customer.id = order.customernumber
WHERE       order.customernumber IS NULL

UNION

SELECT      customer.name, order.id, order.date
FROM        customer
INNER JOIN  order
ON          customer.id = order.customernumber

UNION

SELECT      customer.name, order.id, order.date
FROM        customer
RIGHT JOIN  order
ON          customer.id = order.customernumber
WHERE       customer.id IS NULL
```







Outer joins – overview



Outer joins – examples (all natural xxx joins)

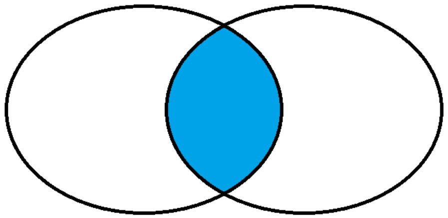
LEFT

A	B
1	2
2	3

RIGHT

B	C
3	4
4	5

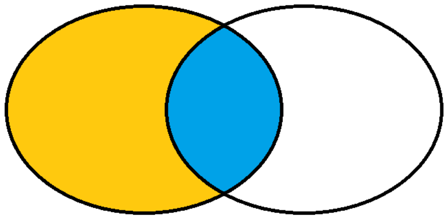
inner join



```
select A, LEFT.B, C
from   LEFT
       natural join
        RIGHT
```

A	B	C
2	3	4

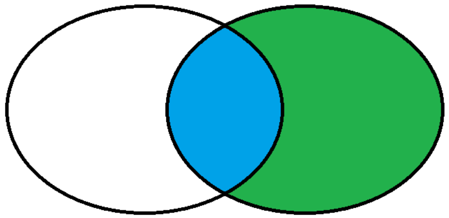
left outer join



```
select A, LEFT.B, C
from   LEFT
  natural left outer join
          RIGHT
```

A	B	C
1	2	⊥
2	3	4

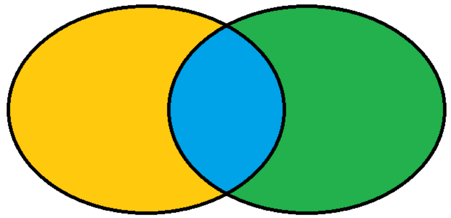
right outer join



```
select A, LEFT.B, C
from   LEFT
  natural right outer join
          RIGHT
```

A	B	C
2	3	4
⊥	4	5

(full) outer join



```
select A, LEFT.B, C
from   LEFT
  natural outer join
          RIGHT
```

A	B	C
1	2	⊥
2	3	4
⊥	4	5

Outer joins - replacement with union

- Outer joins are practical, but not absolutely necessary
- Example: left outer join

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

Join variants

- Given relations: $L(AB)$, $R(BC)$, $S(DE)$

- Equi join: equality condition for explicitly specified and possibly different attributes

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

- Theta join: any join condition

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

- Semi join: only attributes of an operand appear in the results

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

- Formal semantics: see literature

Outer joins

- Note: notation of the symbols is not standardised!
- Full outer join: takes all tuples of both operands

$$r ⟗ s$$

- Left outer join: takes all tuples of the left operand

$$r ⟖ s$$

- Right outer join: takes all tuples of the right operand

$$r ⟕ s$$

- Formal semantics: see literature

Sorting with `order by` (1)

- Notation

```
order by attribute_list
```

- Example

```
select   *
from     WINES
order by Vintage
```

- Sorting in ascending (**`asc`**) or descending (**`desc`**) order

- Sorting as the last operation of a query
  → Sorting attribute must occur in the **`select`** clause

Sorting with `order by` (2)

- Sorting also possible with calculated attributes (aggregates) as a sorting criterion

- Example

```
select   Vineyard, count(*) as Number
from     PRODUCER natural join WINES
group by Vineyard
order by Number desc
```

Sorting: Top-k queries (1)

- Example: determine the 4 youngest wines

- Solution:

```
select   w1.WineId, w1.Name, count(*) as Rank
from     WINES w1, WINES w2
where    w1.Vintage <= w2.Vintage          -- Step 1
group by w1.Name, w1.WineID                -- Step 2
having   count(*) <= 4                     -- Step 3
order by Rank                              -- Step 4
```

- Result

WineId	Name	Rank
3456	Zinfandel	1
2168	Creek Shiraz	2
4961	Chardonnay	3
2171	Pinot Noir	4

Sorting: Top-k queries (2)

- **Top-k query**: returns the best k elements with respect to a ranking function.

- **Design pattern:**
  - **Step 1:** Assignment of the required data sets to be able to calculate the ranking function
  - **Step 2:** Grouping according to the elements, calculation of the ranking
  - **Step 3:** Limiting to rankings ≤ k
  - **Step 4:** Sorting by rank

- Example: determining the k = 4 youngest wines
  - Step 1: Assignment of all wines that are younger
  - Step 2: Grouping according to the names, calculation of the ranking
  - Step 3: Limiting to rankings ≤ 4
  - Step 4: Sorting by rank

Sorting: Top-k queries (3)

- **Top-k clause**: returns the best k elements with respect to a sorting.

```
select top(4)  w1.WineId, w1.Name, count(*) as Rank
from      WINES w1, WINES w2
where     w1.Vintage <= w2.Vintage
group by w1.Name, w1.WineID
order by Rank
```

```
select   w1.WineId, w1.Name, count(*) as Rank
from     WINES w1, WINES w2
where    w1.Vintage <= w2.Vintage          -- Step 1
group by w1.Name, w1.WineID                -- Step 2
having   count(*) <= 4                     -- Step 3
order by Rank                              -- Step 4
```

Handling null values (1)

- Special value in SQL: **`null`**
  - Meaning: unknown or not applicable or not present (depending on the application)
- Test for **`null`** value:
  - *`attr`* **`is null`** returns **`true`**, if *`attr`* is **`null`**
  - *`attr`* **`is not null`** returns **`false`**, if *`attr`* is **`null`**
  - Example:
    ```
    select * from PRODUCER
      where Growing_area is null
    ```
- Terms: result is **`null`** whenever a null value is included in the calculation
  - Exception: aggregate functions: null values eliminated before the function is applied
  - Exception to the exceptions: null values are included in the case of **`count`**(*)
- Comparisons with null value: result in truth (Boolean) value **`unknown`**
  - There are thus 3 possible values for Boolean expressions: `true`, `false` and `unknown`

Handling null values (2)

- Boolean expressions are thus based on trinary logic
- Logical tables for trinary logic

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

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

NOT	
true	false
unknown	unknown
false	true

Named queries (1)

- Example: find all wines that are at most 2 years older or younger than the average age of all wines.
- Query:

```
select   *
from     WINES
where    Vintage >= (
               select avg(Vintage) from WINES) - 2
     and Vintage <= (
               select avg(Vintage) from WINES) + 2
```

- Not nice: subquery is repeated
  - Duplicated code should be avoided (error susceptibility)
  - Unclear/confusing

Named queries (2)

- **Named query:** query expression that can be referenced more than once in the subsequent query (common table expression, CTE)

- Notation

```
  with query_name [(column_list)] as (query_expression)
```

- Query using **with**

```
with AGE(average) as (
    select avg(Vintage) from WINES)
select *
from   WINES, AGE
where  Vintage >= average - 2
  and  Vintage <= average + 2
```

Recursive queries (1)

- Typical application
  - Bill of material queries
  - Computing the transitive closure (e.g. flight connections)
  - etc.
- Example:
  - Bus connections in Upper Bavaria
  - Question: what are all the places we can go to by bus from “Rosenheim”?

BUS

Departure	Arrival	Distance
Rosenheim	Wasserburg am Inn	27
Rosenheim	Kolbermoor	5
Kolbermoor	Großkarolinenfeld	6
Kolbermoor	Bad Aibling	7
Bad Aibling	Raubling	17

Recursive queries (2)

- First attempt: all bus journeys with max. two changes

```
select Departure, Arrival
from   BUS
where  Departure = 'Rosenheim'
  union
select B1.Departure, B2.Arrival
from   BUS B1, BUS B2
where  B1.Departure = 'Rosenheim' and
       B1.Arrival = B2.Departure
  union
select B1.Departure, B3.Arrival
from   BUS B1, BUS B2, BUS B3
where  B1.Departure = 'Rosenheim' and
       B1.Arrival = B2.Departure and
       B2.Arrival = B3.Departure
```

Recursive queries (3) - SQL:2003: **`with recursive`** clause

```
with recursive recursion_table as (
   select ...
  from table
  where ...
    union all
  select ...
  from table, recursion_table
   where recursion_condition
) [traversal_clause] [cycle_clause]
query_expression
```

Initialisation

Recursive part

Recursion step

Non-recursive part

Recursive queries (4)

- Example of recursion in SQL:2003

```
with recursive TOUR(Departure, Arrival) as (
    select Departure, Arrival
    from   BUS
    where  Departure = 'Rosenheim'
      union all
    select T.Departure, B.Arrival
    from   TOUR T, BUS B
    where  T.Arrival = B.Departure)
select distinct * from TOUR
```

Initialisation

Recursive part

Recursion step

Non-recursive part

Recursive queries (5)

- Step-by-step construction of the **TOUR** recursion table

  - Initialisation

Departure	**Arrival**
Rosenheim	Wasserburg am Inn
Rosenheim	Kolbermoor

  - Recursion step (1st iteration)

Departure	**Arrival**
Rosenheim	Wasserburg am Inn
Rosenheim	Kolbermoor
Rosenheim	Großkarolinenfeld
Rosenheim	Bad Aibling

  - Recursion step (2nd iteration)

Departure	**Arrival**
Rosenheim	Wasserburg am Inn
Rosenheim	Kolbermoor
Rosenheim	Großkarolinenfeld
Rosenheim	Bad Aibling
Rosenheim	Raubling

Recursive queries (6)

- Example: arithmetic operations in the recursion step

```
with recursive TOUR(Departure, Arrival, Route) as (

    select Departure, Arrival, Distance as Route
    from   BUS
    where  Departure = 'Rosenheim'

      union all

    select T.Departure, B.Arrival,
           Route + Distance as Route
    from   TOUR T, BUS B
    where  T.Arrival = B.Departure)

select distinct * from TOUR
```

Computability of recursive queries

- Computability (= finiteness of the calculation) is an important requirement for the query language

- Problem: cycles with recursion

```
insert into BUS (Departure, Arrival, Distance)
values ('Raubling', 'Kolbermoor', 12)
```

- 2 ways to handle it in SQL
  1) Limiting the recursion level
  2) Cycle detection
     (defined in the standard since SQL:2003, not yet implemented in any DBMS)

Computability by limiting the recursion level

- Example: max. 2x changes

```
with recursive TOUR(Departure, Arrival, Change) as (

    select Departure, Arrival, 0
    from   BUS
    where  Departure = 'Rosenheim'

      union all

    select T.Departure, B.Arrival, Change + 1
    from   TOUR T, BUS B
    where  T.Arrival = B.Departure and Change <= 2)

select distinct * from TOUR
```

Division (1)

Term Division

- Analogy with the arithmetic operation of <u>integer</u> division: integer division is the inverse of (integer) multiplication, in that it produces the largest number for which multiplication by the divisor is less than the dividend.

- Similarly: $r = r_1 \div r_2$ is the largest relation for which $r \bowtie r_2 \subseteq r_1$.

Division (2) - example

- Example - relations

WINE_RECOMMENDATION

Wine	Critic
La Rose GrandCru	Parker
Pinot Noir	Parker
Riesling Reserve	Parker
La Rose GrandCru	Clarke
Pinot Noir	Clarke
Riesling Reserve	Gault-Millau

GUIDES1

Critic
Parker
Clarke

GUIDES2

Critic
Parker
Gault-Millau

Division (3) - example

- Division with first critic's list `WINE_RECOMMENDATION ÷ GUIDES1` returns

Wine
La Rose GrandCru
Pinot Noir

- Division with second critic's list `WINE_RECOMMENDATION ÷ GUIDES2` returns

Wine
Riesling Reserve

Division (4) - problem: universal quantifier

- Existential quantifier (implicitly) available using selection.
  Universal quantifier is not permitted, but necessary e.g. for division.

- Solution: can be simulated in relational algebra.

- Derivation of the division from Ω:
  Given are $r_1(R_1)$ and $r_2(R_2)$ with $R_2 \subseteq R_1$, $R' = R_1 - R_2$. Then

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

GUIDES1

Critic
Parker
Clarke

- Division of r_1 by r_2

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

WINE_RECOMMENDATION

Wine	Critic
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

- Implementation of the universal quantifier (division) in SQL:

- $r_1 \div r_2 = \pi_{R'}(\, r_1\,) - \pi_{R'}(\, (\pi_{R'}(\, r_1\,) \times r_2) - r_1\,)$, that means
  $WINE\_RECOMMENDATION \div GUIDES =$
  $\pi_{\{\text{Wine}\}}(WINE\_RECOMMENDATION\,) -$
  $\pi_{\{\text{Wine}\}}(\, (\pi_{\{\text{Wine}\}}(WINE\_RECOMMENDATION\,) \times GUIDES) -$
  $WINE\_RECOMMENDATION\,)$

```
select Wine from WINE_RECOMMENDATION
   except
select w.Wine
from (
  select WINES.wine as Wine, GUIDES.Critic as Critic
  from (
      select Wine from WINE_RECOMMENDATION) as WINES, GUIDES
    except
  select * from WINE_RECOMMENDATION) as w
```

Division (6) - division in SQL

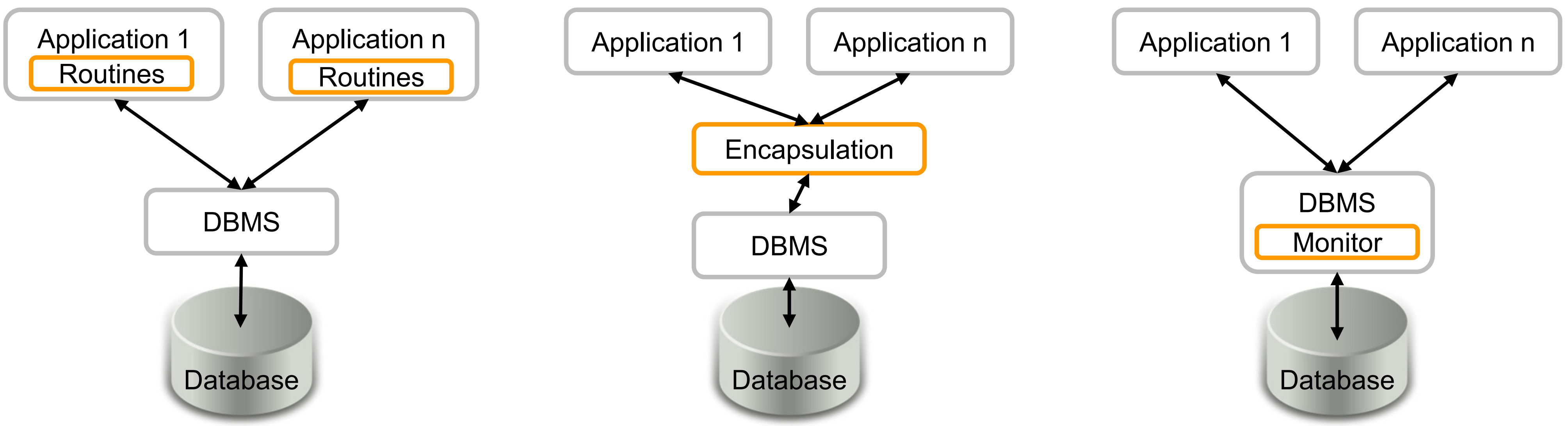
- Alternatively: simulation of the universal quantifier (division) with double negation:

```
select distinct Wine
from   WINE_RECOMMENDATION w1
where  not exists (
  select * from GUIDES2 g
  where not exists (
    select * from WINE_RECOMMENDATION w2
    where w1.Wine = w2.Wine and
          g.Critic = w2.Critic))
```

- Linguistically: "Output all the wines for which there is no critic who has not recommended this wine."

Integrity constraint

- Integrity constraint = condition for the "permissibility" ("correctness") refers to
  - (individual) database states
  - state transitions
  - long-term database developments
- Architectures for integrity assurance



Through the application | Through encapsulation | Through monitor

Inherent integrity constraints in the relational model

- **Domain integrity**
  - SQL allows the specification of value ranges for attributes
  - Allow or prohibit null values
  - Statements: `create domain, not null, default, check`
- **Key integrity**
  - Specify a key for a relation
  - Statement: `primary key …`
- **Referential integrity**
  - Specify foreign keys
  - Statement: `foreign key … references …`

Domain integrity (1) – user-defined value ranges

- **`create domain`**: specification of a user-defined value range
- Example

```
create domain WineColour varchar(4) default 'Red'
   check (value in ('Red', 'White', 'Rose'))
```

- Application

```
create table WINES (
   WineID int primary key,
   Name varchar(20) not null,
   Colour WineColour,
   ...)
```

Not in MySQL, not in TSQL

Domain integrity (2) – local integrity constraints

- **check**: specifying additional local integrity constraints within the value ranges, attributes and relational schemas to be defined

- Example: limiting the permissible values

- Application

```
create table WINES (
   WineID int primary key,
   Name varchar(20) not null,
   Year int check(Year between 1980 and 2010),
   ...)
```

Referential integrity

Maintaining referential integrity

- Verification of foreign key constraints after database changes
- For $\pi_A(r_1) \subseteq \pi_K(r_2)$, e.g. $\pi_{Publisher\_name}(\text{BOOKS}) \subseteq \pi_{Publisher\_name}(\text{PUBLISHERS})$
  - Tuple t is inserted into r_1
    ➔ check if $t' \in r_2$ exists with: $t'(K) = t(A)$, i.e. $t(A) \in \pi_K(r_2)$;

    if not: reject
  - Tuple t' is deleted from r_2
    ➔ check whether $\sigma_{A=t'(K)}(r_1) = \emptyset$, i.e. no tuple from r_1 references t'

    if not empty: reject or delete tuples from r_1 that reference t' (for cascading deletion)

Verification modes of conditions (1)

- **`on update | delete`**
  - Specifies a trigger event that initiates verification of the condition
- **`cascade | set null | set default | no action`**
  - Cascading: handling some integrity violations extends across multiple stages, e.g. deletion in response to violation of referential integrity
- **`deferred | immediate`**

  Not in MySQL, not in TSQL

  - Specifies the time for verification of a condition
  - `deferred`: defer until the end of the transaction
  - `immediate`: immediate verification upon every relevant database change

Verification modes of conditions – cascading deletion

- Example

```
create table WINES (
   WineID int primary key,
   Name varchar(50) not null,
   Price float not null,
   Year int not null,
   Vineyard varchar(30),
   foreign key (Vineyard) references PRODUCER (Vineyard)
   on delete cascade)
```

The `assertion` clause

- Assertion: predicate that expresses a condition that must always be met by the database

- Syntax (SQL:2003)
  - `create assertion name check ( predicate )`
  - But: not implemented in any current commercial system!

- Examples:

```
create assertion Prices check
      ((select sum (Price) from WINES) < 10000)
create assertion Prices2 check
      (not exists (select * from WINES where Price > 200))
```

Not in MySQL, not in TSQL

Triggers

- Triggers: statement/procedure that is automatically executed by the DBMS when a certain event occurs

- Applications
  - Enforcing integrity constraints ("implementation" of integrity rules)
  - Auditing of DB actions
  - Propagation of DB changes

- Specification of
  - Event and condition for activating the trigger
  - Action(s) to execute
  - This is why it is often called the ECA (event-condition-action) rule

- Available in most commercial systems (different syntax)

Example: realisation of calculated attribute through two triggers

- Inserting new orders:

```
create trigger OrderCountPLUS
on Order
after insert
as begin
     update Customer
     set NumberOfOrders = NumberOfOrders + 1
     where CNo in (select Cno from inserted)
end
```

- Similarly for deleting orders:

```
create trigger OrderCountMINUS
on Order
after delete
as begin
     update Customer
     set NumberOfOrders = NumberOfOrders - 1
     where CNo in (select CNo from deleted)
end
```

Triggers: syntax in SQL Server (simplified)

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

- **`FOR | AFTER`**: trigger is executed after the triggering statement
- **`INSTEAD OF`**: trigger is executed instead of the triggering statement
- **`INSERT, UPDATE, DELETE`**: at which statements should the trigger be executed
- **`sql_statement`**: trigger action to be executed
  - May use the special tables **`deleted`** and/or **`inserted`**

Triggers: tables `inserted` and `deleted`

- deleted table
  - copies of rows affected (=deleted) by DELETE and UPDATE statements
  - deleted table and trigger table generally do not contain the same rows.

- inserted table
  - copies of rows affected by INSERT and UPDATE statements
  - during an insert or update transaction, new rows are added to both the inserted table and the trigger table
  - rows in the inserted table are copies of the new rows in the trigger table

- update = delete + subsequent insert
  - old rows initially copied into deleted table
  - subsequently new rows are copied into the trigger table and inserted table

More examples of triggers (1)

- No customer account may fall below 0:

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

More examples of triggers (2)

- Producers must be deleted if they no longer offer wines:

```
create trigger useless_Vineyard
on WINES
after delete
as
  delete from PRODUCER
  where Vineyard in
      (select Vineyard
       from deleted d
       where not exists (select *
                         from WINES w
                         where w.Vineyard = d.Vineyard))
```

Integrity assurance through triggers

1) Specify object o_i for which the condition ϕ should be monitored
   - generally consider multiple o_i if condition applies across relations
   - candidates for o_i are tuples of the relation names that appear in ϕ

2) Specify the elementary database changes u_{ij} to objects o_i that can violate ϕ
   - rules: e.g. check existence requirements when deleting and updating, but not when inserting etc.

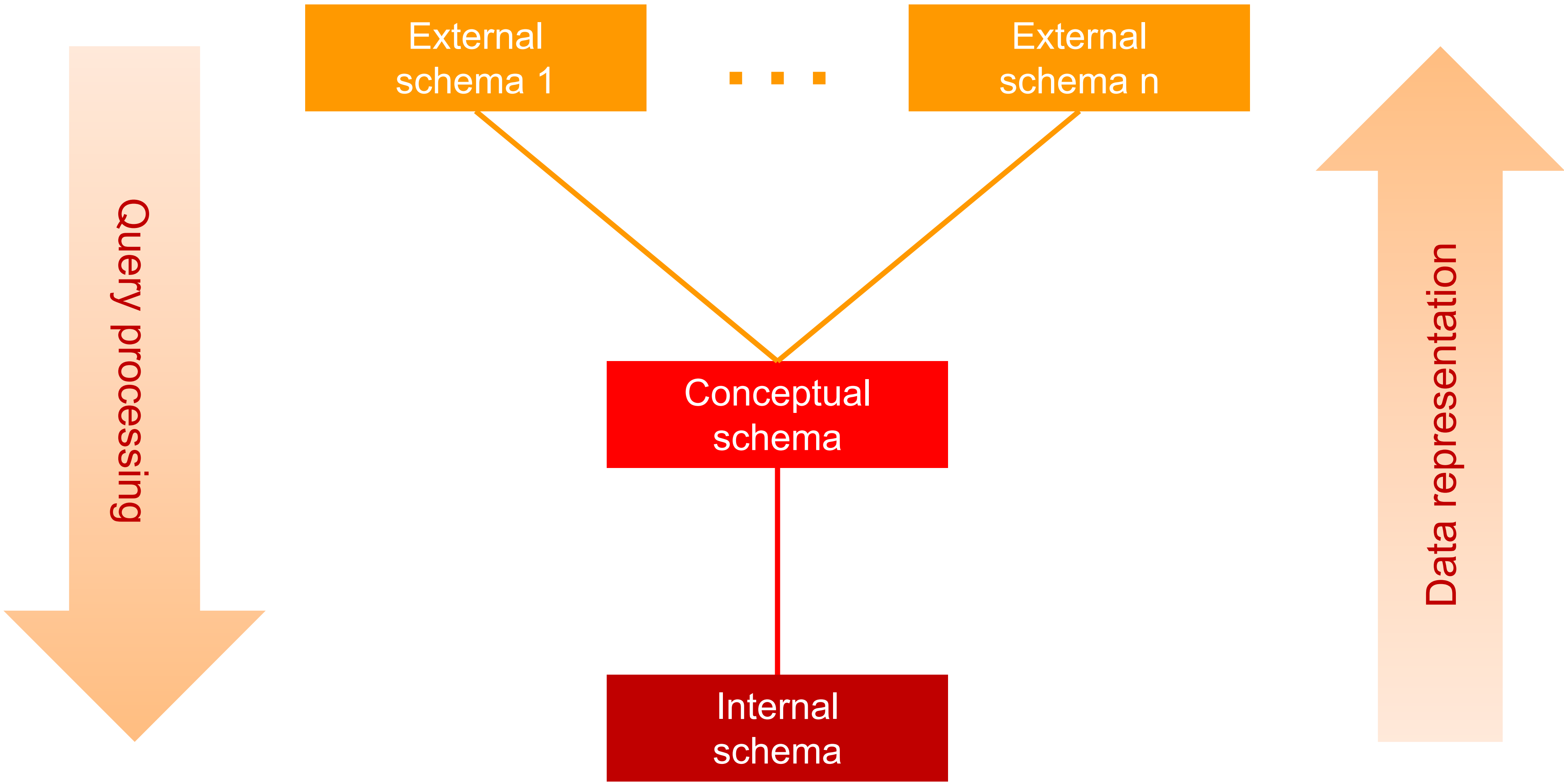
3) Specify the reaction r_i to an integrity violation, depending on the application
   - reset the transaction (**`rollback`**)
   - corrective database changes

4) Formulate following triggers

```
create trigger t-phi-ij on oᵢ after uᵢⱼ
          as if(¬ϕ) begin rᵢ end
```

5) If possible, simplify triggers created

Schema architecture – general (rev.)



Schema architecture – general (rev.)

Query processing

Name	Zutaten	Temperatur	Rührdauer
Akapulko-Zartherb	30% Criollo Venezuela 20% Nacional Ecuador 48% Zucker 2% Vanille	76 Grad	90h
Criollo Venezuela 80%	80% Crillo Venezuela 20% Zucker	77 Grad	80h
Milch Zartschmelzend	35% Arriba Venezuela 65% Zucker	78 Grad	30h

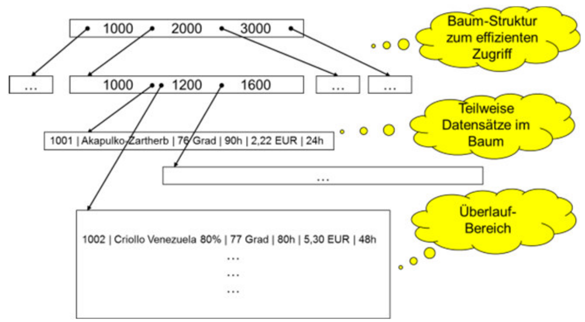
Name	Kakao	Preis	Lieferzeit
Akapulko-Zartherb	50%	2,22 EUR	24h
Criollo Venezuela 80%	80%	5,30 EUR	48h
Milch Zartschmelzend	35%	1,50 EUR	12h

Data representation

SNr	Name	Temperatur	Rührdauer	Preis	Lieferzeit
1001	Akapulko-Zartherb	76 Grad	90h	2,22 EUR	24h
1002	Criollo Venezuela 80%	77 Grad	80h	5,30 EUR	48h
1003	Milch Zartschmelzend	78 Grad	30h	1,50 EUR	12h

ZNr	Name	IstKakao
2001	Criollo Venezuela	True
2002	Nacional Ecuador	True
2003	Arriba Venezuela	True
2004	Vanille	False
2005	Zucker	false

SNr	ZNr	Anteil
1001	2001	30%
1001	2002	20%
1001	2005	48%
1001	2004	2%
1002	2001	80%
1002	2005	20%
1003	2003	35%
1003	2005	65%



Views (1)

- Views: virtual relations
  (or virtual database objects in other data models)

- Views are external DB schemas following the 3-level schema architecture

- View definition
  - Relational schema (implicit or explicit)
  - Calculation rule for virtual relation, such as SQL query

Views (2)

- Advantages
  - Simplification of queries for the user of the database, for example by implementing frequently required subqueries as a view
  - Possibility for structuring the database description, tailored to user classes
  - Logical data independence enables stability of the interface for applications in the event of changes to the database structure (correspondingly in reverse direction)
  - Restriction of accesses to a database in the context of access control

- Challenges
  - Automatic query transformation
  - Execution of updates to views

Terms

- Query: sequence of operations that calculates a result relation from the base relations
  - display the result relation interactively on the screen or
  - further processing by the programme ("embedding")
- View: sequence of operations that are saved under a view name over the long term and can be called up again under this name. Results in a **view relation.**
- Snapshot: result relation of a query that is stored under a snapshot name, but is never calculated a second time (with changed base relations) (e.g. annual balances).

Definition of views in SQL

- View definition
  - Relational schema (implicit or explicit)
  - Calculation rule for virtual relation, such as SQL query

- Syntax in SQL

```
create view ViewName [ SchemaDeclaration ]
as SQLQuery
[ with check option ]
```

Views - example

- All red wines from Bordeaux

```
create view BordeauxRedWines as
   select   Name, Vintage, WINES.Vineyard
   from     WINES natural join PRODUCER
   where    Colour = 'Red' and
            Region = 'Bordeaux'
```

Problem areas with views

Two big challenges with the view concept

1) Automatic query transformation

2) Execution of updates to views (view update problem)
   - Projection views
   - Selection views
   - Join views
   - Aggregation views

Queries on views

- SELECT statement on view: view is replaced by its definition
  - Possible due to SQL orthogonality (since SQL92)
  - Resulting statement is simplified and optimised

- Example

```
select *
from BordeauxRedWines
where Vintage = 2000
```

Becomes

```
select *
from (  select  Name, Vintage, WINES.Vineyard
        from    WINES natural join PRODUCER
        where   Colour = 'Red' and
            Region = 'Bordeaux') as BordeauxRedWines
where Vintage = 2000
```

SQL query transformation when using views

SQL query uses view ➔ required transformations:

- Since SQL92: nested `select` statements allowed in `from` part
  - ➔ Transformation by simple syntax replacement
- Before SQL92: no nested `select` statements allowed in `from` part
  - ➔ Transformation by "mixing"
  - `select`: possibly rename the view attributes or replace them with a calculation term
  - `from`: names of the original relations
  - conjunctive combination of the `where` clauses of view definition and query (possible renamings)
  - leads to various problems with aggregation views

Criteria for updates to views

- Effect conformance
  - User sees the effect as if the update had been executed directly on the viewing relation
- Minimality
  - Base database should only be modified minimally in order to achieve the effect mentioned
- Consistency preservation
  - Updating a view must not lead to integrity violations in the base database
- Respect for data protection
  - If the view is introduced for data protection reasons, the consciously hidden part of the base database must not be affected by updates to the view

Projection views

- INV := $\pi_{\text{WineID,Name,Vineyard}}$(WINES)
- In SQL:

```
create view INV as
   select WineID, Name, Vineyard
   from WINES
```

- Update statement for the INV view:

```
insert into INV(WineID, Name, Vineyard)
   values (3333, 'Dornfelder', 'Müller')
```

- Corresponding statement for the WINES base relation:

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

Selection views (1)

- $IV := \sigma_{Vintage>2000}(\pi_{WineID,Vintage}(WINES)$
- In SQL:

```
create view IV as
    select  WineID, Vintage
    from    WINES
    where   Vintage > 2000
```

- Tuple migration: A tuple
  `WINES(3456, ’Zinfandel’, ’Red’, 2004, ’Helena’)`
  is “moved out” of the view:

```
update IV
    set     Vintage = 1998
    where   WineID = 3456
```

Selection views (1)

- Control of the tuple migration

```
create view IV as
     select   WineID, Vintage
     from     WINES
     where    Vintage > 2000
     with check option
```

Join views (1)

- WP := WINES ⋈ PRODUCER
- In SQL:

```
create view WP as
   select   WineID, Name, Colour, Vintage,
WINES.Vineyard,
               Growing_area, Region
   from     WINES, PRODUCER
   where    WINES.Vineyard = PRODUCER.Vineyard
```

- Change operations usually not uniquely translatable:

```
insert into WP
   values (3333, ’Dornfelder’, ’Red’, 2002, ’Helena’,
           ’Barossa Valley’, ’South Australia’)
```

Join views (2)

- Update is transformed into

```
insert into WINES
  values (3333, 'Dornfelder', 'Red', 2002, 'Helena')
```

- Plus
  - Either: insert statement for PRODUCER:

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

  - or alternatively:

```
update PRODUCER
  set Growing_area='Barossa Valley',Region='South
Australia'
  where Vineyard='Helena'
```

    - better in terms of minimality requirement
    - but contradicts effect conformance!

Aggregation views

- Example in SQL:

```
create view CM (Colour, MinVintage) as
    select      Colour, min(Vintage)
    from        WINES
    group by    Colour
```

- The following update cannot be uniquely implemented:

```
update CM
    set      MinVintage = 1993
    where    Colour = 'Red'
```

Summary of the problem areas

- Violation of the schema definition (e.g. insertion of null values for projection views)
- Data protection: avoid side effects on the non-visible part of the database (tuple migration, selection views)
- Not always unique transformation: selection problem
- Aggregation views (among others): no meaningful transformation possible
- Elementary view change should correspond with precisely one atomic change to base relation: 1:1 relationship between the view tuples and the tuples of the base relation (no projecting of keys)

Rights assignment in database systems

- Access rights: WHO – WITH WHAT – WHAT

```
(AuthorisationID, DB_extract, Operation)
```

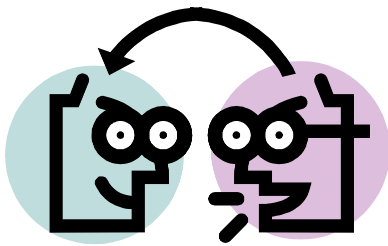
- `AuthorisationID` is the internal identifier of a "database user"
- `DB_extract`: relations and views
- `Operation`: read, insert, update, delete

Rights assignment in SQL

- Syntax

```
grant <Rights>
     on <Table>
     to <UserList>
     [with grant option]
```

- Explanations:
  - In `<Rights>` list: `all` or long form `all privileges` or list from `select`, `insert`, `update`, `delete`
  - After `on`: relation or view name
  - After `to`: authorisation identifiers (also `public`, `group`)
  - Special right: right to pass on rights (`with grant option`)



Example

- Authorisation for public: “Every user can view his orders and insert new orders (but not delete them!).”

```
create view MyOrders as
     select *
     from ORDER
     where CName = user;
```

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

Withdrawal (revocation) of rights

- Syntax

```
revoke <Rights>
     on <Table>
     from <UserList>
     [restrict | cascade ]
```

- Explanations:
  - `restrict`: if rights have already been passed on to third parties: termination of `revoke`
  - `cascade`: withdrawal of rights propagated using `revoke` to all users who received it from this user via `grant`

Role model from SQL:2003

- Roles were introduced from SQL:2003 to simplify the administration of rights
- Instead of giving rights directly to users, rights are assigned to roles and then roles to users → easier to transfer when a person changes
- Example

```
create role winedb_admin_role;
grant winedb_admin_role to gunter;
grant select
    on WINES
    to public;
grant all
    on WINES
    to winedb_admin_role;
```

Rights assignment in commercial DBMS

- Rights assignment in commercial DBMS is considerably more complex
- Usually, the concepts of users and roles remain the basis
- Often integration with other rights systems
  - Users of the operating system (e.g. Windows users)
  - Integration with identity management systems (e.g. Active Directory)
  - Own user administration of the DBMS
  - → Often multiple systems like this are used simultaneously/in parallel
- Example
  - You log in to the exercise with the "SQL Server ID" that you received in the first exercise group
  - At the same time, our system administrator logs in with their Windows ID, which is administered by a domain controller

Chapter 11: SQL/PSM

In this chapter, we will address the following questions

- What are stored procedures and stored functions?
- Why do we use them?
- How do we write them?
- Which language constructs are supported by SQL/PSM?

Literature: CompleteBook Chap 9.4 ; Beaver book Chap 13.5

Chapter 11: SQL/PSM

PLEASE NOTE!

All statements in this chapter use the MS SQL Server syntax, not the ANSI SQL standard

SQL/PSM - Motivation

- Problems of client-server systems ("embedded SQL" and "CLI"):
  - Constant change of execution control between application (=client) and DBS (=server)
  - No optimisation across multiple statements possible
- Solution: stored procedures
  - Software modules administered & executed in DBMS (procedures/functions)
  - Called from applications and queries / action parts of triggers
- Advantages of stored procedures
  - Structuring tools for larger applications: redundancy-free representation of relevant aspects of the application functionality
  - Procedures only depend on the DBMS
  - Optimisation of procedures
  - Execution of the procedures under control of the DBMS
    - Rights assignment for procedures

SQL/PSM: the standard

- ANSI standard for procedural extensions: SQL/PSM (Persistent Stored Modules)

- Components:
  - Stored modules from procedures and functions
  - Individual routines
  - Integration of external routines (implemented in C, Java, . . .)
  - Syntax constructs for loops, conditions, etc.

- Implementation (more or less compliant) in all current DBMS
  - Oracle: PL/SQL
  - IBM DB2: very similar to SQL/PSM
  - Informix: SPL
  - Microsoft SQL Server: Transact-SQL
  - MySQL, PostgreSQL: similar to SQL/PSM

Chapter 11: SQL/PSM

Variables

- Variables contain a single data value of a specific type
- Variable names must begin with an @ character
- Declaration of variables (with optional initialisation)

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

- Assigning values to variables

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

Note: `null` is possible as a value for `value` and `expression`

- Example

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

Assigning variables in SELECT

- Variables can be assigned by means of "=" after SELECT
- If the SELECT returns multiple tuples, the values of the last tuple are used (but this is bad style)
- Examples

```
DECLARE @WineName NVARCHAR(100)
DECLARE @WineColour NVARCHAR(100)

SELECT @WineName = name, @WineColour = Colour
FROM Wines
WHERE WineID = 2171
```

```
DECLARE @NumberOfHelena INT

SELECT @NumberOfHelena = COUNT(*)
FROM Wines
WHERE Vineyard = 'Helena'
```

Flow control

- Grouping statements (into a statement block)

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

- Conditional execution

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

- PRINT statement – output to screen

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

Functions

- “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.” (Source: SQL Server Documentation)

- Creating a function (simplified)

```
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 [ ; ]
```

Functions - example

- Definition: function to increase a number by a percentage

```
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
```

- Calls of the function

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

```
UPDATE Wines SET Price = dbo.addPercent(Price, 50)
WHERE Vineyard='Creek'
```

Functions – more complex example

Example: function that rates a wine as cheap, acceptable or expensive.

```
CREATE FUNCTION rating( @price NUMERIC(10,2) )
RETURNS NVARCHAR(50)
BEGIN
  DECLARE @rated NVARCHAR(50);

  IF @price < 5.00
    SET @rated = 'cheaper';
  ELSE IF @price < 20.00
    SET @rated = 'more acceptable';
  ELSE
    SET @rated = 'more expensive';

  SET @rated = @rated + 'Wine';
  RETURN @rated;
END
```

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

Procedures

- "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)." (Source: SQL Server Documentation)

- Creating a procedure (simplified)

```
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. (Source: SQL Server Documentation)

Procedures - example

- Procedure to increase all wines from a vineyard by a percentage.
  Return: number of wines that have been increased and new maximum price.

```
CREATE PROCEDURE increasePrices
    @myVineyard NVARCHAR(20),
    @percent INT,
    @count INT OUT,
    @newMax NUMERIC(10,2) OUT
AS BEGIN
  UPDATE Wines SET Price = dbo.addPercent(Price, @percent)
  WHERE Vineyard = @myVineyard;

  SELECT @count = COUNT(*), @newMax = MAX(Price)
  FROM Wines
  WHERE Vineyard = @myVineyard;
END
```

Calling procedures

- Executing a procedure (simplified)

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

- Example

```
DECLARE @Number INT
DECLARE @MostExpensive NUMERIC(10,2)
DECLARE @ByPercent INT = 30

EXEC dbo.increasePrices 'Helena', @ByPercent, @Number OUT,
     @MostExpensive OUT

-- possible alternative:
-- EXECUTE dbo.increasePrices @count=@Number OUT,
--    @newMax=@MostExpensive OUT, @myVineyard='Helena',
@percent=@ByPercent

PRINT 'Helena has ' + CAST(@Number AS NVARCHAR(10)) + ' Wines, ' +
  'the most expensive costs ' + CAST(@MostExpensive AS NVARCHAR(10))+
  ' EUR'
```

Chapter 11: SQL/PSM

Loops

- WHILE loop

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

`BREAK`
Terminates the **innermost** WHILE loop

`CONTINUE`
Restarts the WHILE loop (all statements after CONTINUE are ignored)

SELECT queries in PSM

1) Single-value queries: two options

```
DECLARE @NumOfRedWines INT
SET @NumOfRedWines = (SELECT COUNT(*) FROM Wines WHERE Colour='Red')
-- or
SELECT @NumOfRedWines = COUNT(*) FROM Wines WHERE Colour='Red'
```

2) Single-row queries: Assign values using "=" in SELECT

```
DECLARE @myName NVARCHAR(100);
DECLARE @myColour NVARCHAR(20);
SELECT @myName = Name, @myColour = Colour
  FROM Wines WHERE WineID=4711
```

3) Multiple result tuples: using a cursor

Cursors - "tuple iterator" for a query

- Declaration of the cursor using

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

- Before use: open the cursor

```
OPEN cursor_name
```

- After use: close and deallocate the cursor

```
DEALLOCATE cursor_name
```

- Getting the next tuple in variables and moving the cursor:

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

@@FETCH_STATUS

Contains the status of the last FETCH statement (0 = everything OK, otherwise error)

WHILE loop with CURSOR - example

- Calculation of average price and variance of a vineyard

```
CREATE PROCEDURE meanVar(@Vineyard NVARCHAR(100), @mean REAL OUT, @var REAL OUT)
AS BEGIN
  DECLARE @thisPrice NUMERIC(10,2), @NumOfWines INTEGER;
  DECLARE WineCursor CURSOR LOCAL FOR SELECT Price FROM Wines WHERE
Vineyard=@Vineyard;
  SET @mean = 0.0;
  SET @var = 0.0;
  SET @NumOfWines = 0;
  OPEN WineCursor;
  FETCH NEXT FROM WineCursor INTO @thisPrice;
  WHILE @@FETCH_STATUS = 0 BEGIN
    SET @NumOfWines = @NumOfWines + 1;
    SET @mean = @mean + @thisPrice;
    SET @var = @var + @thisPrice * @thisPrice;
    FETCH NEXT FROM WineCursor INTO @thisPrice;
  END
  DEALLOCATE WineCursor;
  SET @mean = @mean / @NumOfWines;
  SET @var = @var / @NumOfWines - @mean * @mean;
END
```

Summary SQL/PSM

- SQL/PSM =
  powerful possibility to "programme in the DBS"
- Complete programming language
  - with all related advantages and disadvantages
- Enables
  - significant performance increases
  - better structured code