8. SQL – Data Handling

8.1 Update, Deletion, Insertion and bulk load*

8.2 The query language SQL

8.2.1 Search predicates

8.2.2 Arithmetic expressions and functions in predicates

8.2.3 Different kinds of join

8.2.4 Output improvement

8.3 Advanced SQL

8.3.1 Subselects and Correlated subqueries

8.3.2 Quantified expressions, SOME, ANY

8.3.3 Grouping and Aggregation

8.3.4 Transitive closure

8.3.5 Final remarks: NULLS, temporary relations and more

Lit.: Melton / Simon, Understanding SQL 1999, chap. 8,9; Kemper / Eickler chap 4,
SQL chapter in any book on DBS

8.3 Advanced SQL

8.3.1 Subselects and correlated subqueries

Using result relations instead of relation constants

```
SELECT title, director
FROM Movie
WHERE director IN ('Spielberg','Lubitsch',
                    'Kubrick');
```

```
SELECT title, director
FROM Movie
WHERE director IN
      (SELECT name
       FROM Customer c);
```

Independent outer and inner SQL block

- Independent subquery
- Can it be expressed differently ?

Advanced SQL: Subselects

Correlated Subqueries

Find movies which have been rented in the same year they have been produced:

```
SELECT title
 FROM Movie m
 WHERE to_char(year,'YYYY') IN
 (SELECT  to_char(r.from_date, 'YYYY')
       FROM Rental r, Tape t
       WHERE r.tape_Id = t.t_id
        AND t.m_Id = m.m_Id);
```

- – Could be expressed simpler, how?
- – Always possible to avoid subqueries?

Advanced SQL: Subselects

…. Some can be expressed as an ordinary join

```
SELECT DISTINCT title
FROM Movie m NATURAL JOIN Tape t  JOIN Rental r ON
  (r.tape_Id = t.t_id)
WHERE to_char(m.year,'YYYY') =
      to_char(r.from_date,'YYYY');
```

Using an explicit quantifier EXISTS:

$$\{m.title \mid M(m) \wedge \exists t \exists r\, (Tape(t) \wedge Rental(r) \wedge m.m\_id = t.m\_Id$$
$$\wedge\, r.t\_Id = t.id \wedge m.year = r.from\_date\,\}$$ -> see e.g book by Kemper

```
SELECT title FROM Movie m
WHERE EXISTS
 (SELECT  *
  FROM Tape t JOIN Rental r ON(r.tape_Id = t.t_id)
  WHERE t.m_Id = m.m_Id
        AND to_char(m.year,'YYYY') =
            to_char(r.from_date,'YYYY'));
```

8.3.2 Advanced SQL: Quantifiers

EXISTS

– Translates directly from calculus

```
SELECT title
FROM Movie m
WHERE EXISTS
       (SELECT *
        FROM  Tape t
        WHERE t.format = 'VHS'
          AND t.m_Id = m.m_Id);
```

– Not needed: algebraic expression translates into ordinary join: $\pi_{\texttt{title}}$ (M $\bowtie$ $\sigma_{\texttt{format='VHS'}}$ T)

```
SELECT title
FROM Movie m, Tape t
WHERE t.m_Id = m.m_Id AND t.format = 'VHS'
```

Advanced SQL: EXISTS

But NOT EXISTS enhances the language

Find Customers (names) who have not loaned the tape with No 11.

```
SELECT name
FROM customer c, Rental r
WHERE
   NOT( c.mem_No = r.mem_No
             AND r.tape_Id = 11);
… more rows than Customers in DB)
```

Wrong result!

```
SELECT name
FROM customer c
WHERE
  NOT EXISTS
   (SELECT * FROM Rental r
    WHERE c.mem_No = r.mem_No
             AND r.tape_Id = 11);
```

Could NOT IN be used?

Advanced SQL Universal Quantifier

Example

Find movies available in exactly one format

Reformulate precisely: Find movie titles such that: there exists tape with this movie with a particular format and all other tapes either contain a different movie or they have the same format.

$$\{\text{m.title} \mid \text{Movie}(m) \land \exists\, t\ (\text{Tape}(t) \land \text{t.m\_Id} = \text{m.m\_Id} \\ \land \forall\, x\ (\text{Tape}(x) \land t <> x \Rightarrow \text{t.format} = \text{x.format} \lor \text{x.m\_Id} <> \text{m.m\_Id}))\ \}$$

SQL does not have universal quantification but "NOT EXISTS"

Identities to use:

$\forall\, x\ (P(x)) \equiv \neg \exists\, x\ \neg P(x)$ and $Q \Rightarrow A \equiv \neg Q \lor A$

$$\{\text{m.title} \mid \text{Movie}(m) \land \exists\, t\ (\text{Tape}(t) \land \text{t.m\_Id} = \text{m.mId} \\ \land \neg \exists\, x\ (\text{Tape}(x) \land t <> x \land \text{t.format} <> \text{x.format} \land \\ \text{x.m\_Id} = \text{m.m\_Id}))\ \}$$

Advanced SQL Universal Quantifier

$$\{\text{m.title} \mid \text{Movie}(m) \land \exists\, t\ (\text{Tape}(t) \land \text{t.m\_Id} = \text{m.mId} \\ \land \neg \exists\, x\ (\text{Tape}(x) \land t <> x \land \text{t.format} <> \text{x.format} \land \\ \text{x.m\_Id} = \text{m.ml\_d}))\ \}$$

```
SELECT DISTINCT m.title
FROM Movie m, Tape t
WHERE m.m_Id = t.m_Id
AND NOT EXISTS
  (SELECT *
  FROM Tape x
  WHERE x.t_id <> t.t_id
  AND t.format <> x.format
  AND x.m_Id = m.m_Id);
```

Advanced SQLUniversal Quantifier

Quantification and value comparison

Find the most expensive movies

Find movie, the price of which is higher or equal than that of all other movies. (i.e. the most expensive ones)

$\{m.title \mid Movie(m) \wedge$
$\forall\, m1\, (M(m1) \Rightarrow m1{=}m \vee m.price\_Day >= m1.\ price\_Day) \}$

$\{m.title \mid Movie(m) \wedge$
$\neg \exists\, m1\, (M(m1) \wedge m1 <> m \wedge m.price\_Day < m1.\ price\_Day) \}$

```
SELECT m.title FROM Movie m
WHERE NOT EXISTS
 (SELECT m1.m_id  FROM Movie m1
  WHERE m1.m_id <> m.m_id AND m.price_day < m1.price_day)
```

Advanced SQL Universal value Quantification

Quantified comparison operator

Find movie, the price of which is higher or equal than that of all other movies. (i.e. the most expensive ones)

$\{m.t \mid M(m) \wedge$
$\forall\, m1\, (M(m1) \Rightarrow m1 <> m \vee m.price\_Day >= m1.\ price\_Day) \}$

```
SELECT title
FROM Movie m
WHERE m.price_Day >= ALL
   (SELECT price_Day
    FROM Movie m1
    WHERE m1.m_id <> m.m_id);
```

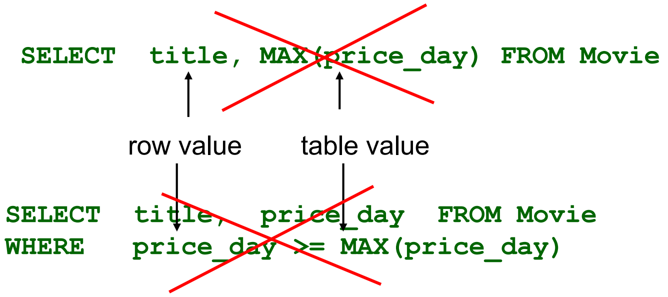
or because of '>=' :

```
SELECT title
FROM Movie m
WHERE m.price_Day >= ALL
   (SELECT price_Day
    FROM Movie m1);
```

Advanced SQL: Row and table predicates

"Find most expensive movie"

Could built-in function MAX (<column>) be used?



```
~~SELECT   title, MAX(price_day) FROM Movie~~
```

row value

table value

```
~~SELECT   title,  price_day  FROM Movie~~
~~WHERE    price_day >= MAX(price_day)~~
```

NEVER mix row and table expression!

```
SELECT   title,  price_day  FROM Movie
WHERE    price_day >=
   (SELECT MAX(price_day) FROM Movie );
```

correct:
MAX(..) is the
only value

Advanced SQL

More quantified comparison predicates

= SOME same as IN

<=SOME ≡ x… ∃ t (x.a <= t.a)

```
SELECT title, director
FROM Movie m
WHERE director = SOME
      (SELECT name
       FROM Customer c);
```

Can be expressed simpler by inner join:

```
.. WHERE m.director
    c.name;
```

< SOME ≡ x… ∃ t (x.a < t.a)

```
SELECT title, price_Day
FROM Movie m
WHERE price_Day  < ANY
      (SELECT price_Day
       FROM Movie);
```

SOME is ANY :)

ANY is sometimes misleading.
Used in SQL 92 / 99.
EVERY not in Oracle

Advanced SQL Universal Quantifier

Systematic development of universally quantified expressions

- Write a tuple calculus expression
- replace $\forall x\,(P(x))$ by $\neg\exists x \neg P(x))$
- Replace => using the rule from propositional logic:
  $Q \Rightarrow A \equiv \neg Q \vee A$
- Translate to SQLish

OR

- Describe a counterexample
  e.g. a movie m does not belong to answer set if there exists at least two copies, both of which.....
- Express it in SQL
- Express combined
  search condition
  using **NOT EXISTS**

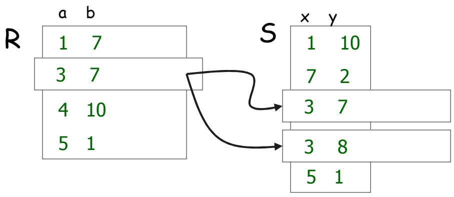
```
            ...
            NOT EXISTS(SELECT *
FROM Tape x
WHERE x.t_id <> t.t_id
  AND t.format <> x.format
  AND x.m_Id = m.m_Id);
```

Advanced SQL Subqueries

Are nested subqueries really necessary?

e.g. MSQL does not allow subqueries

- No correlated subqueries, if the one row from correlated tupels is used for qualification



$\{r.b \mid R(r) \wedge \exists s\,(S(s) \wedge s.x = r.a \wedge s.y >= 7)\}$

- Needed if more than one row is used from subordinate query

$\{r.b \mid R(r) \wedge \forall s\,(S(s) \wedge s.x = r.a \Rightarrow s.y >= 7)\}$

8.3.3 Grouping and Aggregation

Aggregate (or set) functions

f_A :: table -> value, where A is some Subset of Σ(table)

Aggregate functions are table functions, i.e. defined on tables or subsets of tables

COUNT, SUM, AVG, MIN, MAX are standard functions sometimes statistical functions (e.g. variance) in addition

```
COLUMN AVG FORMAT 9.99;   -- Oracle Formatting
SELECT MIN(price_Day) AS "MIN", MAX(price_Day) AS "MAX",
       AVG(price_Day) AS AVG
FROM Movie;
```

Advanced SQL: aggregate functions

Target list contains only aggregate functions or none of them (exception: groups, see below)

Result semantically undefined otherwise:

```
SELECT AVG(price_Day), Movie.mid
  FROM Movie
```

does not make sense, see above

Aggregate values different from row values, therefore subqueries inevitable, if aggregates used

```
SELECT m.Title
FROM Movie m
WHERE price_Day >
 (SELECT AVG(price_Day)
    FROM Movie
  );
```

Advanced SQL aggregate functions

- Quantifiers and counting (in finite sets)

```
select x
from   R
where  EXISTS
          (select * from S...)
```

≡

```
select x
from   R
where  0 <
          (select count(* ) from S...)
```

```
SELECT DISTINCT title, format
FROM Movie m , Tape t
WHERE  m.m_Id = t.m_Id
AND  0  =
   (SELECT COUNT(*)
   FROM Tape x
   WHERE x.t_id <> t.t_id
   AND t.format <> x.format
   AND m.m_Id =  x.m_Id );
```

Note: ANSI Join vs explicit join

NATURAL join

```
SELECT DISTINCT title, format
FROM Movie m NATURAL JOIN Tape t
WHERE
 0 =
  (SELECT COUNT(*)
  FROM Tape x
  WHERE x.t_id <> t.t_id
  AND t.format <> x.format
  AND m.m_Id =  x.m_Id );
```

Syntax error: `m.m_Id` projected by Natural Join

```
… AND m_Id =  x.m_Id );
```

Result set empty, wrong result

```
… AND x.t_id = t.t_id
  AND format <> x.format
  AND m_Id =  x.m_Id );
```

Wrong result

Advanced SQL: aggregates

Find the movies m for which holds: for every format there is at least one copy of m having this format

```
SELECT DISTINCT t.m_Id
FROM Tape t
WHERE
 (SELECT COUNT(DISTINCT format)
  FROM Tape tx
  WHERE  tx.m_Id = t.m_Id) =
 (SELECT COUNT(DISTINCT format)
  FROM Format );
```

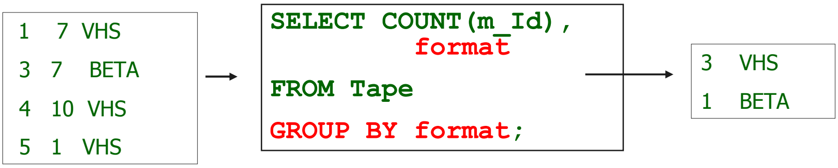
- Naiv formulation of query
- May be optimized by rewriting and optimization.

Advanced SQL: Grouping

IMPORTANT

Grouping

GROUP BY <columns> groups all rows with the same values in <columns> into "virtual subtables"



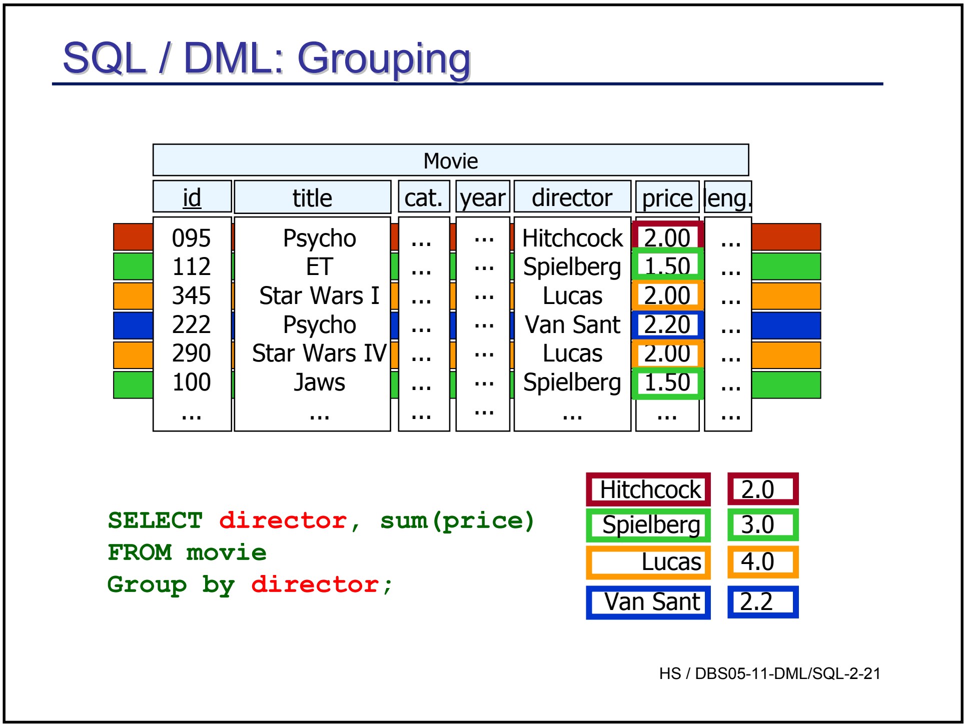
1	7	VHS
3	7	BETA
4	10	VHS
5	1	VHS

```
SELECT COUNT(m_Id),
            format
FROM Tape
GROUP BY format;
```

3	VHS
1	BETA

Aggregation is performed over each group.
⇨ Exactly one result row for each group.

In each group the values of grouping attributes are constants ⇨these attributes – and no others! - may occur in target list together with aggregate functions.



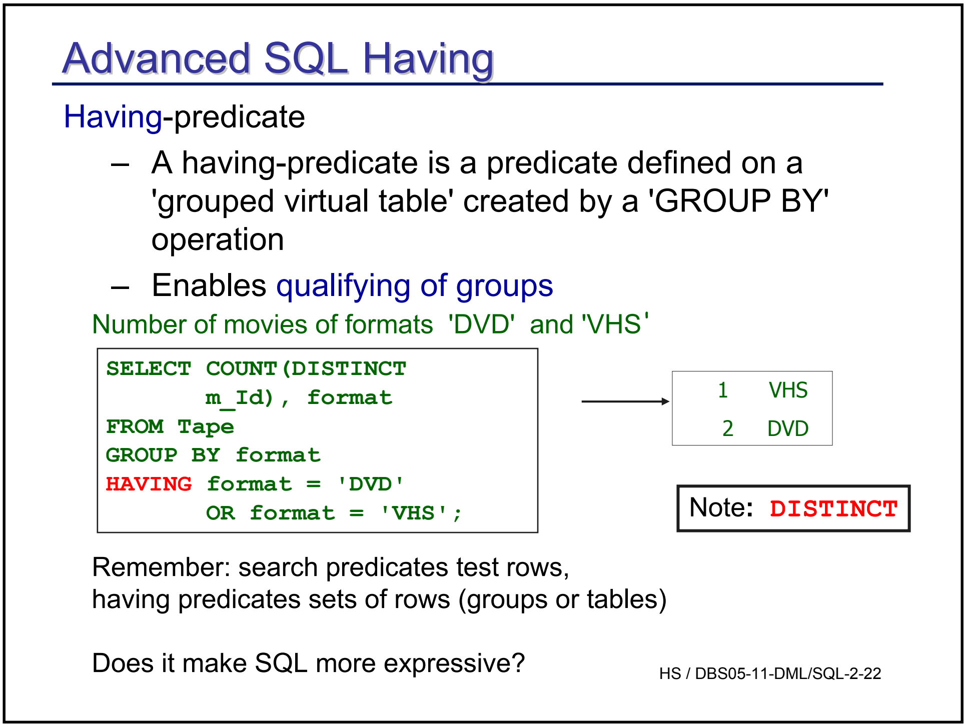
SQL / DML: Grouping

Movie

	id	title	cat.	year	director	price	leng.	
🟥	095	Psycho	...	...	Hitchcock	2.00	...	🟥
🟩	112	ET	...	...	Spielberg	1.50	...	🟩
🟧	345	Star Wars I	...	...	Lucas	2.00	...	🟧
🟦	222	Psycho	...	...	Van Sant	2.20	...	🟦
🟧	290	Star Wars IV	...	...	Lucas	2.00	...	🟧
🟩	100	Jaws	...	...	Spielberg	1.50	...	🟩
	...	...	...	...	...	...	...	

```
SELECT director, sum(price)
FROM movie
Group by director;
```

Hitchcock	2.0
Spielberg	3.0
Lucas	4.0
Van Sant	2.2



Advanced SQL Having

Having-predicate

- A having-predicate is a predicate defined on a 'grouped virtual table' created by a 'GROUP BY' operation
- Enables qualifying of groups

Number of movies of formats 'DVD' and 'VHS'

```
SELECT COUNT(DISTINCT
       m_Id), format
FROM Tape
GROUP BY format
HAVING format = 'DVD'
       OR format = 'VHS';
```

1	VHS
2	DVD

Note: DISTINCT

Remember: search predicates test rows,
having predicates sets of rows (groups or tables)

Does it make SQL more expressive?

Advanced SQL: Having

Predicates in having clause:
defined only on grouped columns or aggregated by a set function

Number of copies for those movies with more than one format, provided movie_id is greater than 1.

```
SELECT m_Id, COUNT(DISTINCT format)
FROM Tape
GROUP BY m_Id
HAVING COUNT(DISTINCT format)> 1 AND m_Id > 1;
```

"No having without 'GROUP BY' "

More expressive?

Yes: Table functions / predicates not defined on rows, but on groups

Advanced SQL   Having

Universal quantification on finite sets using having

```
SELECT m_Id
FROM Tape t
GROUP BY m_Id
HAVING COUNT(DISTINCT format) =
(SELECT COUNT(DISTINCT format)
 FROM Tape);
```

More examples:

Find number of tapes and customer number for all customers with some given name pattern ("%h%")

```
SELECT  mem_no, COUNT(tape_id)
FROM Customer c NATURAL JOIN
Rental r
WHERE c.name LIKE '%A%'
GROUP BY mem_no;
```

Grouping after selection

GROUP BY

- A realistic example[1]

product (product_id, name, price, cost)
sales (product_id, units, date, ...)

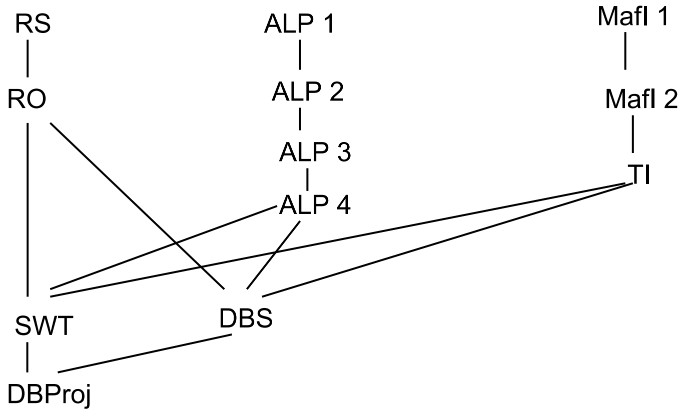
"Find for each product the profit made within the last 4 weeks"

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
 FROM products p LEFT JOIN sales s USING (product_id)
WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
GROUP BY product_id, p.name, p.price, p.cost
HAVING sum(p.price * s.units) > 5000;
```

[1] taken from Postgres manual

8.3.4 Transitive closure

Representing a directed Graph
Example: course prerequisites



Represent graph by a set of nodes and a set of edges

Transitive closure

- Example: Find courses required for SWT

enhanced SQL:1999

```
-- Nodes
CREATE TABLE Course(
lnr int  primary key,
name varchar(20));
```

```
-- Edges
CREATE TABLE Requires(
pre int  references course(lnr),
suc int  references course(lnr),
constraint req_pk primary key(pre, suc));
```

ANSI SQL: Transitive closure

ANSI SQL (SQL 99) syntax for recursive traversals

Steps: 1. construct nonrecursive relation containing all path

2. Select data

```
WITH RECURSIVE PreCourse( pre, suc  )
   AS (SELECT pre,suc FROM Requires r WHERE pre
        NOT IN (SELECT suc FROM Requires r1)
   UNION
       SELECT pre,suc
       FROM Requires r, PreCourses p
       WHERE p.suc = r.pre
       )
SELECT p1.suc , c.name
FROM preCourse p1, course c
WHERE    p1.suc = c.lnr;
```

Advanced SQL Transitive closure

- Transitive Closure in Oracle: CONNECT

ORACLE

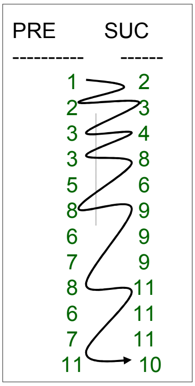
Courses

LNR	NAME
1	ALP I
2	ALP 2
3	ALP 3
4	ALP 4
5	RS
6	RO
7	Theory
8	SWP
9	DBS
10	DBProj
11	SWT

```
SELECT l.name
FROM Course l, requires r
WHERE  r.suc = l.lnr
START WITH pre = 1
CONNECT BY PRIOR suc = pre;
```

```
NAME
-----------
ALP 2
ALP 3
ALP 4
SWP
DBS
SWT
DBProj
```

Requires



PRE	SUC
1	2
2	3
3	4
3	8
5	6
8	9
6	9
7	9
8	11
6	11
7	11
11	10

1. Only ONE path ("start with..")
2. from leaf to root: exchange **suc** and **pre**

Advanced SQL Transitive closure

LEVEL attribute for hierachical relationships

ORACLE

All lectures required for lecture XYZ

```
SELECT l.name, LEVEL
FROM lecture l, requires r
WHERE  r.suc = l.lnr
START WITH pre = 1
CONNECT BY PRIOR suc = pre;
```

```
NAME        LEVEL
---------- ----------
ALP 2           1
ALP 3           2
ALP 4           3
SWP             3
DBS             4
SWT             4
RA              5
```

8.3.4 Final remarks: NULL et al.

Some remarks about NULL

- Null treated in SQL as "unknown"
- Semantics of "unkown" used in predicates
  - If predicate evaluates to unknown for row r, r is not returned
  - UNKNOWN AND TRUE = UNKOWN
  - UNKNOWN OR TRUE = TRUE
  - Arithmetical expression evaluate to NULL if some attribute value is NULL
- In some cases, UNKOWN behaves like FALSE

Note: different real world semantics conceivable
e.g. : "not defined"

Example: Person(…, sex, civil_serviceDate,…) ,
c_sDate not defined for female (in GER)

SQL NULL

Be careful with aggregates:

```
SELECT * FROM testNull

    X          Y
 ------- ----------
    1          1
    2          1
    3
    4
```

```
SELECT SUM(y)/COUNT(y)
   from testNull;
 SUM(Y)/COUNT(Y)
---------------
              1
```

Same result for

```
SELECT AVG(y) from….
```

But:

```
SELECT SUM(y)/COUNT(*)
FROM testNull;

SUM(Y)/COUNT(*)
---------------
,5
```

SQL / DML Structuring

Structuring SQL programs

.... is not easy

e.g.
```
tmp = SELECT  a from B where P;
      SELECT x from Y where Y.z  in tmp;
```
not allowed in most systems

What does it mean?

a) Is tmp a name for the SQL expression ?
Unfortunately not allowed in SQL-2
but in SQL-3: `SELECT  x,y,z   FROM (SELECT ...) AS tmpRel`
`tmpRel` does only exist inside the outer `SELECT` Block.

b) Or a "stateful relation variable"?
Bad: the "state of the variable" is always a snapshot
in a multi user environment.
E.g.:
```
tmp = SELECT  stock_rate  from stocks where
         ...;
```

SQL / DML Structuring

Temporary tables

SQL -3

- When inconsistency is not an issue, temporary tables make sense
- Instead of assignment <relation_var> := select ... the relation variable has to be declared :
  `create temporary table myTmp ( ....)`
- and assigned a value by means of an **INSERT** statement:

useful variant of insert ⇨
```
insert into mytmp
     (select x,y,z from R where…)
```

- What's makes the difference ?
  Tempory tables are local snapshots, they are "dropped" at the end of a session.

Structuring

Subquery factoring / local definition

```
WITH r as (
   select  m.title, m.m_id AS x, tt.m_id
   from movie m, tape tt
   where m.year > to_date(2000,'YYYY') )

select r.title, t.t_id
from r, tape t
where r.x = t.m_id;
```

Local definition

Compare let in Haskell

Compare **WITH RECURSIVE....**

Summary

- SQL: THE interlingua of data management
- Differences (standard, systems) considerable
- Eventually convergence towards SQL 99
- Set manipulation as dominating operation
- Set specification in a declarative way
- Grouping: frequent operation
- Many language enhancements in SQL 99 (transitive closure, structuring)
- Interactive language: embedding into host language to be discussed