Introduction to database design

First lecture: RG 3.6, 3.7, [4], most of 5
Second lecture: Rest of RG 5

Rasmus Pagh

Some figures are taken from the ppt slides from the book
Database systems by Kiefer, Bernstein, Lewis

Announcements

- 270 movie ratings received so far
  - would like at least 400
- Kulturnatten: If you are interested, please talk to me after the lecture.
- Hand-in 1:
  - Feedback in writing and/or orally (today?).
  - Generally good, though many groups need to polish their E-R diagrams.
  - Many groups uncertain on normalization, e.g. candidate key vs superkey.

Today's lecture

This week and next week we cover

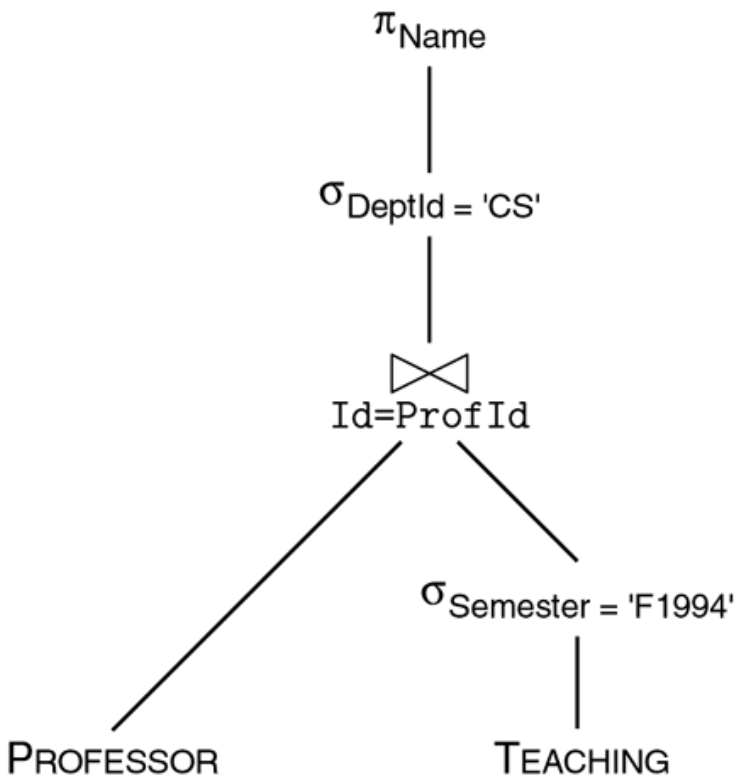
- SQL (RG chapter 5), and
- relational algebra (RG chapter 4).

Both are relational query languages.

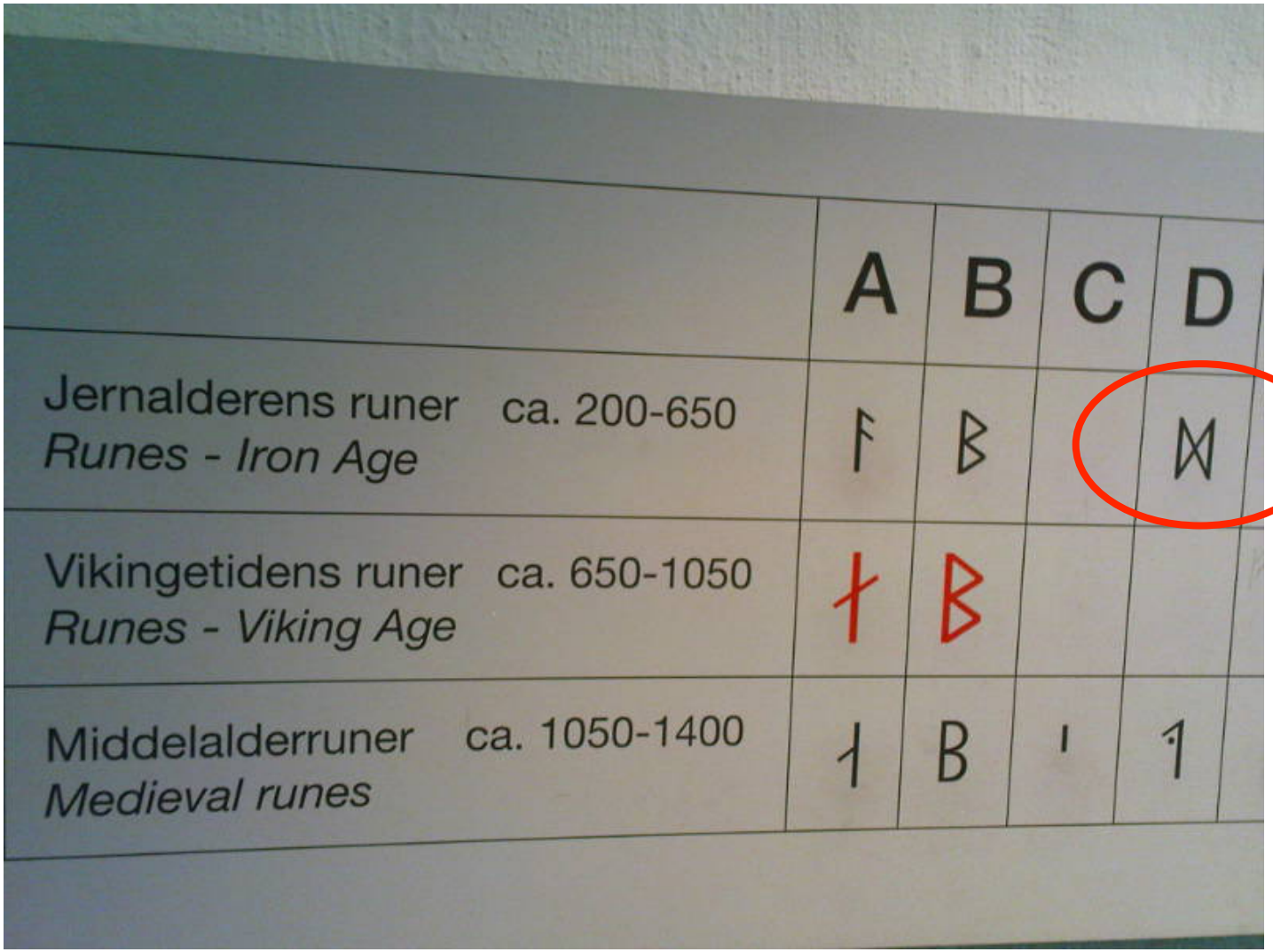
- SQL is declarative:
  Describe **what** you want.
- Relational algebra is procedural:
  Describe **how** to get what you want.
- Complementary views can improve understanding.

Relational algebra expression

(formatted as a tree)



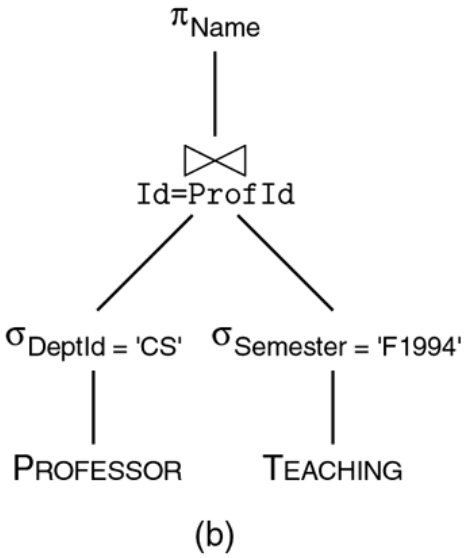
Greek letters, runes?



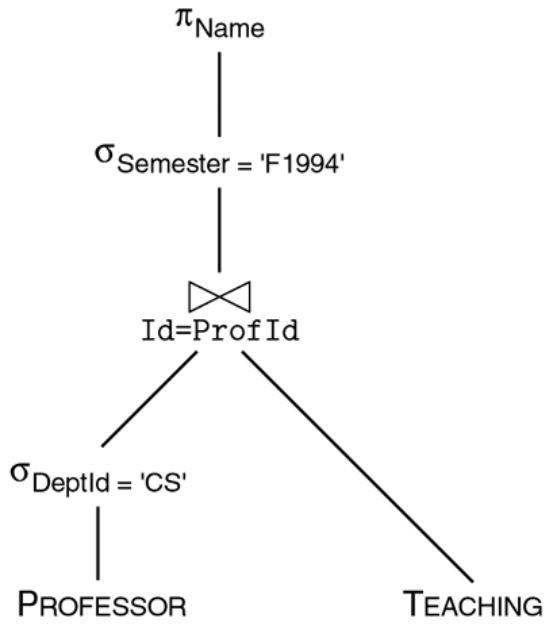
	A	B	C	D
Jernalderens runer ca. 200-650 *Runes - Iron Age*	ᚠ	ᛒ		ᛞ
Vikingetidens runer ca. 650-1050 *Runes - Viking Age*	ᚠ	ᛒ		
Middelalderruner ca. 1050-1400 *Medieval runes*	ᚠ	ᛒ	ᛁ	ᛏ

Query tree

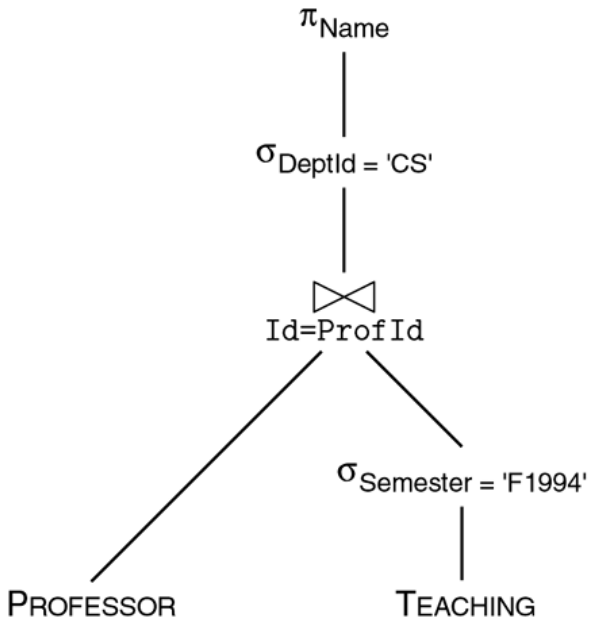
```
SELECT P.Name
FROM PROFESSOR P, TEACHING T
WHERE P.Id=T.ProfId AND T.Semester='F1994'
      AND P.DeptId='CS'
```



(b)



(c)



(d)

Relational algebra

E. F. Codd, 1970

- Relations are considered a ***set*** of *tuples*, whose components have names.
- Operators operate on 1 or 2 relations and produce a relation as a result
- An algebra with 5 basic operators:
  - Select
  - Project
  - Union
  - Set difference
  - Cartesian product

Project

$$\pi_{\text{attributelist}}(\text{relation})$$

- Projection chooses a subset of attributes.
- The result of a projection is a relation with the attributes given in attribute list.
  By default the result is a **set**, i.e., contains no duplicates.

$\pi_{\text{year}}(\text{movie})$

```
SELECT DISTINCT year
FROM movie
```

Select

- Selection of a subset of the tuples in a relation fulfilling a condition
- Denoted $\sigma_{\text{condition}}(\text{relation})$
- Operates on one relation

$$\sigma_{\text{height}>210}(\text{person})$$

```
select *
from person
where height > 210;
```

Expressions in SELECT

You can define new attributes using expressions:

```
SELECT 10*floor(year/10)
FROM danishMovies;
```

You can give attributes new names:

```
SELECT year(birthdate) AS birthyear
FROM person;
```

Cartesian product
(aka. cross product)

Id	Name
111223344	Smith, Mary
023456789	Simpson, Homer
987654321	Simpson, Bart

A subset of $\pi_{\text{Id,Name}}(\text{STUDENT})$

Id	DeptId
555666777	CS
101202303	CS

A subset of $\pi_{\text{Id,DeptId}}(\text{PROFESSOR})$

STUDENT.Id	Name	PROFESSOR.Id	DeptId
111223344	Smith, Mary	555666777	CS
111223344	Smith, Mary	101202303	CS
023456789	Simpson, Homer	555666777	CS
023456789	Simpson, Homer	101202303	CS
987654321	Simpson, Bart	555666777	CS
987654321	Simpson, Bart	101202303	CS

$R \times S$ for relations R and S is the relation containing all tuples that can be formed by **concatenation** of a tuple from R and a tuple from S.

In SQL: `SELECT * FROM R,S;`

Join

Cartesian product is almost always used together with a condition on what tuples should be "joined". Special notation:

$$R \bowtie_{\text{condition}} S$$

is equivalent to

$$\sigma_{\text{condition}}(R \times S)$$

Join example

(equi-join)

```
SELECT *
FROM movie, person, involved
WHERE movie.id=movieId AND
      person.id=personId
```

Two equivalent relational algebra expressions:

$$(\text{movie} \bowtie_{\text{movie.id}=\text{movieId}} \text{involved}) \bowtie_{\text{person.id}=\text{personId}} \text{person}$$

$$\text{movie} \bowtie_{\text{movie.id}=\text{movieId}} (\text{involved} \bowtie_{\text{person.id}=\text{personId}} \text{person})$$

Natural join

- A join where all attributes with the **same name** in the two relations are included in the join condition as **equalities** is called **natural join**.
- The resulting relation only includes one copy of each attribute.
- Natural join is denoted:

$$R \bowtie S$$

Semantics of SELECT statement

```
SELECT A1,A2,…
FROM R1,R2,…
WHERE <condition>
```

Algorithm for evaluating:

1. **FROM** clause is evaluated. Cartesian product of relations is computed.
2. **WHERE** clause is evaluated. Rows not fulfilling condition are deleted.
3. **SELECT** clause is evaluated. All columns not mentioned are removed.

A way to think about evaluation, but in practice more efficient evaluation algorithms are used.

Self-join via tuple variables

```
SELECT m1.title,m1.year,m2.year
FROM mov m1, mov m2
WHERE m1.title=m2.title AND
      m1.id>m2.id;
```

Conceptually, the tuple variables `m1` and `m2` act as "copies" of `mov`.

String operations

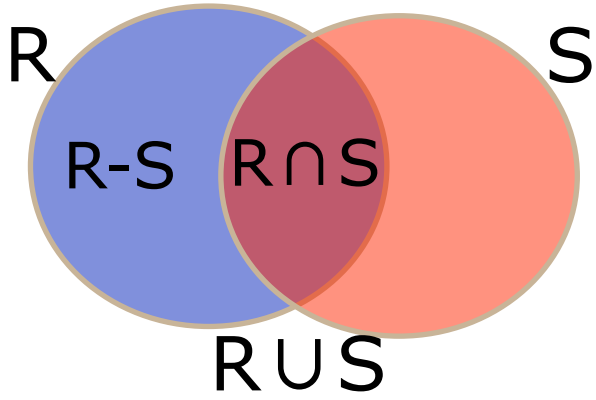
- Expressions can involve string ops:
  - Comparisons of strings using =, <,… Strings are compared according to lexicographical order, e.g., 'green'>'blue'.
  - MySQL: Not case sensitive! 'Green'='green'
  - Concatenation: 'Data' || 'base' = 'Database'
  - LIKE, 'Dat_b%' LIKE 'Database'
    - _ matches any single character
    - % matches any string of 0 or more characters
    - title='%green%' is true for all titles with 'green' as a substring, e.g. 'The Green Mile'
  - Details needed for project: See MySQL documentation. (http://dev.mysql.com/doc/refman/5.5/en/string-functions.html)

Date operations

- You will probably need them in the second hand-in.
- See MySQL documentation for details.
  http://dev.mysql.com/doc/refman/5.5/en/date-and-time-functions.html

Set operations

Set operations are union (R∪S), set difference (R-S), and intersection (R∩S).



Note that two relations have to be **union-compatible** for set operations to make sense, meaning that they have the same set of attributes.

Set operations - examples

$$\sigma_{\text{birthyear}}(\text{person}) \cup \sigma_{\text{deathyear}}(\text{person})$$

All birth years and death years of a person

$$\sigma_{\text{birthyear}}(\text{person}) \cap \sigma_{\text{deathyear}}(\text{person})$$

All years with both a birth and a death.

Set operations

- UNION (∪), INTERSECT(∩), and EXCEPT(-).

```
(SELECT *
FROM Car C
WHERE C.Color='green')
UNION
(SELECT *
FROM Car C
WHERE C.Color='blue')
```

```
(SELECT C.Regnr, C.Color
FROM Car C
WHERE C.Color='green')
EXCEPT
(SELECT *
FROM Car C
WHERE C.Regnr=1234)
```

MySQL supports UNION, but requires relations to be ”encapsulated” in SELECT.

Aggregation by example

```
SELECT AVG(height) FROM person
```

```
SELECT COUNT(DISTINCT country)
FROM movie
```

`NULL` values: Not taken into account, except in `COUNT(*)`

Aggregation functions

Functions:

- `COUNT ([DISTINCT] attr)`: Number of rows
- `SUM ([DISTINCT] attr)`: Sum of attr values
- `AVG ([DISTINCT] attr)`: Average over attr
- `MAX (attr)`: Maximum value of attr
- `MIN (attr)`: Minumum value of attr

- `DISTINCT`: only one unique value for attr is used

More functions: See MySQL manual

http://dev.mysql.com/doc/refman/5.5/en/group-by-functions.html

Grouping

When more than one value should be computed, use grouping with aggregation:

```
SELECT year,count(*) FROM movie
GROUP BY year;

SELECT country,avg(imdbRank)r
FROM movie
GROUP BY country ORDER BY r;
```

The columns returned can only be the aggregate or columns mentioned in the `GROUP BY` clause. (Why?)

HAVING

```
SELECT director,COUNT(*)
FROM danishMovies
GROUP BY director
HAVING COUNT(*)>5;
```

`HAVING` is a condition on the group.
Use any condition that makes sense:

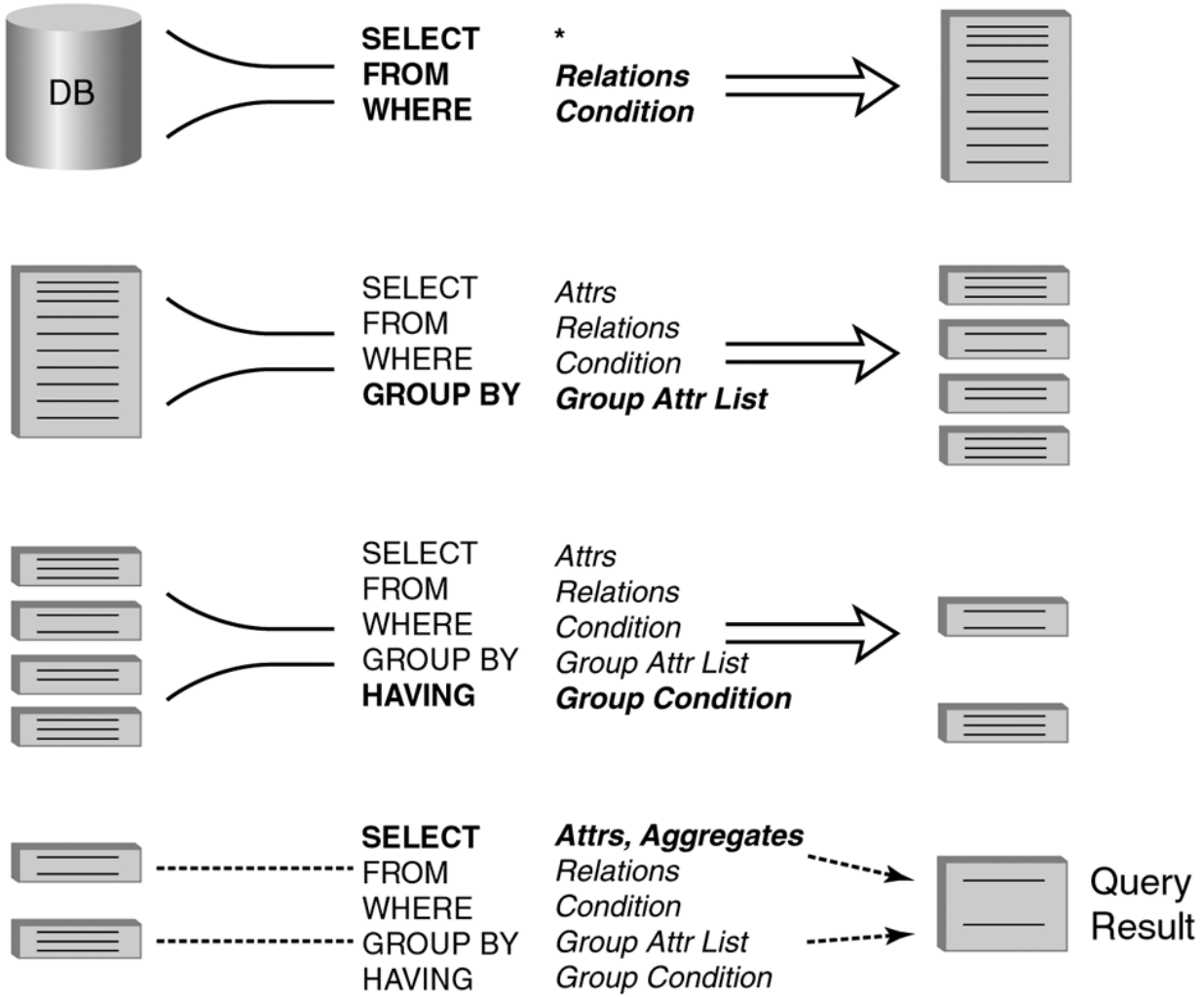
- Aggregates over tuples in group
- Conditions on tuple attributes

Evaluation algorithm

Algorithm for evaluating a `SELECT-FROM-WHERE`:

1. **FROM:** Cartesian product of tables is computed. Subqueries are computed recursively.
2. **WHERE:** Rows not fulfilling condition are deleted. Note that aggregation is evaluated after WHERE, i.e. aggregate values can't be in the condition.
3. **GROUP BY:** Table is split into groups.
4. **HAVING:** Eliminates groups that don't fulfill the condition.
5. **SELECT:** Aggregate function is computed and all columns not mentioned are removed. One row for each group is produced.
6. **ORDER BY:** Rows are ordered.

In a figure…


DB
SELECT
FROM
WHERE
*
Relations
Condition
SELECT
FROM
WHERE
GROUP BY
Attrs
Relations
Condition
Group Attr List
SELECT
FROM
WHERE
GROUP BY
HAVING
Attrs
Relations
Condition
Group Attr List
Group Condition
SELECT
FROM
WHERE
GROUP BY
HAVING
Attrs, Aggregates
Relations
Condition
Group Attr List
Group Condition
Query
Result

Subqueries 1: In FROM clause

A relation in the `FROM` clause can be defined by a subquery. **Example**:

```
SELECT country, MAX(c)
FROM (SELECT country, language, COUNT(*) c
      FROM movie
      GROUP BY country,language) cl
GROUP BY country;
```

A larger example

```
SELECT movie.country, language
FROM(SELECT country, MAX(c) AS ma
     FROM (SELECT country, language, count(*) c
           FROM movie
           GROUP BY country,language) cl
     GROUP BY country) cm,
    movie
WHERE cm.country = movie.country
GROUP BY movie.country, movie.language, cm.ma
HAVING COUNT(*)=cm.ma;
```

What does this compute?

Alternative syntax

- Some DBMSs (e.g. Oracle) give this alternative to subqueries in `FROM`:

```
WITH cl AS (SELECT country, language, count(*) c
            FROM movie
            GROUP BY country,language),
     cm AS (SELECT country, MAX(c) AS ma
            FROM cl
            GROUP BY country)
SELECT movie.country, language
FROM cm, movie
WHERE cm.country = movie.country
GROUP BY movie.country, movie.language, cm.ma
HAVING COUNT(*)=cm.ma;
```

Subqueries 2: In WHERE

```
SELECT director,title,year
FROM danishMovies d1
WHERE year =
     (SELECT MAX(year)
      FROM danishMovies d2
      WHERE d1.director=d2.director);
```

The newest movie(s) for each director.

Subquery with negation

```
SELECT director,title,year
FROM danishMovies d1
WHERE year NOT IN
     (SELECT MAX(year)
      FROM danishMovies d2
      WHERE d1.director=d2.director);
```

Not expressible as a `SELECT-FROM-WHERE` without a subquery in `WHERE`.

Subroutines in SQL

Views are used to define queries that are used several times as part of other queries:

```
CREATE VIEW imdb AS
(SELECT title,year,name,birthdate,height
FROM movie, involved, person
WHERE movie.id=movieId and personId=person.id);
```

The view can be used in different queries:

```
SELECT * FROM imdb WHERE name='Tom Cruise';
```

```
SELECT name,COUNT(*) FROM imdb
GROUP BY name HAVING COUNT(*)>200
```

Views

- A view defines **a subquery**.
- Defining a view does **not** execute any query.
- When a view is used, the **query definition** is **copied** into the query (as a subquery).

Views can be used for:

1. Defining queries used as subqueries, making code more modular.
2. Logical data independence.
3. Customizing views for different users.
4. Access control.

Views and access control

Views can be used to limit the access to data, the right to update data, etc.

Example: `GRANT SELECT ON imdb TO ALL`

Meaning: All users can see the table `imdb`, but *not* the underlying relations.

Other options:

- `GRANT INSERT`, `GRANT ALL`, and more
- `TO ALL`, `TO user`, `TO group`

(Full) outer join, by example

SupplName	PartNumber
Acme Inc.	P120
Main St. Hardware	N30
Electronics 2000	RM130

SUPPLIER relation

PartNumber	PartName
N30	10'' screw
KCL12	2lb hammer
P120	10-ohm resistor

PARTS relation

SupplName	PartNumber	PartNumber2	PartName
Acme Inc.	P120	P120	10-ohm resistor
Main St. Hardware	N30	N30	10'' screw
Electronics 2000	RM130	NULL	NULL
NULL	NULL	KCL12	2lb hammer

Full outer join SUPPLIER $\overset{outer}{\bowtie}_{\text{PartNumber=PartNumber}}$ PARTS

Outer join in SQL

- **Syntax**:
  `R FULL OUTER JOIN S ON <condition>.`
- Semantics:
  Output the normal (inner) join result
  `SELECT * FROM R,S WHERE <condition>`,
  plus tuples from `R` and `S` that were *not* output (padded with `NULL`s).

- **Variants**: Left and right outer joins (supported in MySQL).

Problem session

- Suppose you have a DBMS that does not support:
  - INTERSECT
  - EXCEPT
  - FULL OUTER JOIN
- How can you simulate the above using the following joins?
  - LEFT JOIN
  - RIGHT JOIN
  - SELECT-FROM-WHERE

Runtime errors

If the subquery returns more than one tuple in a place where a single value is expected, a *runtime error* results.

```
SELECT director,title,year
FROM danishMovies d1
WHERE d1.title =
    (SELECT title
     FROM danishMovies d2
     WHERE d1.director=d2.director and
           d1.year=d2.year);
```

Beware of NULLs!

- Things are not always what they appear.
  - Aggregates treat NULLs differently
  - Logic is different. (`x IS NULL` vs `x=NULL`)
  - Different DBMSs handle NULLs differently…

- Example:

```
SELECT * FROM BestMovies
WHERE ((country="Canada") or
 (country!="Canada" and imdbRank>9.5));
```

  Different behavior for NULL /empty string…

Beware of NULLs, cont.

Questions | Tags | Users | Badges | Unanswered

Why does Oracle 9i treat an empty string as NULL?

39

I know that it *does* consider ' ' as `NULL`, but that doesn't do much to tell me *why* this is the case. As I understand the SQL specifications, ' ' is not the same as `NULL` -- one is a valid datum, and the other is indicating the absence of that same information.

7 Answers

active | oldest | **votes**

43

I believe the answer is that Oracle is very, very old.

Back in the olden days before there was a SQL standard, Oracle made the design decision that empty strings in VARCHAR/ VARCHAR2 columns were NULL and that there was only one sense of NULL

NULLs and boolean logic

$cond_1$	$cond_2$	$cond_1$ AND $cond_2$	$cond_1$ OR $cond_2$
true	*true*	*true*	*true*
true	*false*	*false*	*true*
true	*unknown*	*unknown*	*true*
false	*true*	*false*	*true*
false	*false*	*false*	*false*
false	*unknown*	*false*	*unknown*
unknown	*true*	*unknown*	*true*
unknown	*false*	*false*	*unknown*
unknown	*unknown*	*unknown*	*unknown*

cond	NOT *cond*
true	*false*
false	*true*
unknown	*unknown*

Updating the database

```
INSERT INTO TableName(a1,…,an)
VALUES (v1,…,vn)

INSERT INTO TableName(a1,…,an)
SelectStatement

DELETE FROM TableName
WHERE Condition

UPDATE TableName
SET a1=v1,…ai=vi
WHERE Condition
```

Updating a view!?

```
CREATE VIEW movietitles AS
(SELECT title FROM movie);
```

What are the effects of each of these updates:

```
INSERT INTO movietitles values
('Superrrrrman');

DELETE FROM movietitles
WHERE title='Superrrrrman';
```

Updating using a view

Insertion: For unspecified attributes, use `NULL` or default values if possible.
Deletion: May be unclear what to delete.
Several restrictions, e.g. exactly one table can be mentioned in the `FROM` clause.

Not all views are updatable. **Example**:

```
CREATE VIEW uniqueMovieTitles AS
(SELECT DISTINCT title FROM movie);
```

Materialized views

(not available in MySQL)

Views are computed **each time** they are accessed – possibly inefficient.

Materialized views are computed and stored physically for faster access.

When the base tables are updated the view changes and must be recomputed:
- May be inefficient when many updates
- Main issue – when and how to update the stored view

Updating materialized views

When is the view updated

- **ON COMMIT** – when the base table(s) are updated
- **ON DEMAND** – when the user decides, typically when the view is accessed

How is the view updated

- **COMPLETE** – the whole view is recomputed
- **FAST** – some method to update only the changed parts.
  - For some views the incremental way is not possible with the available algorithms.)

Related course goal

Students should be able to:

- write SQL queries, involving multiple relations, compound conditions, grouping, aggregation, and subqueries.

Next steps

- We will cover the rest of the slides next week.
- Exercises:
  - 12.30: Another chance to look at normalization exam problems posed two weeks ago. (Or exercises on SQL.)
  - 13.15: Presentation of solutions to two exam problems.
  - Ca. 13.30: Exercises on SQL.
- Hand-in 2 released early next week.