Introduction to database design

KBL chapter 5
(pages 127-187)

Rasmus Pagh

Some figures are borrowed from the ppt slides from the book
used in the course, Database systems by Kiefer, Bernstein, Lewis

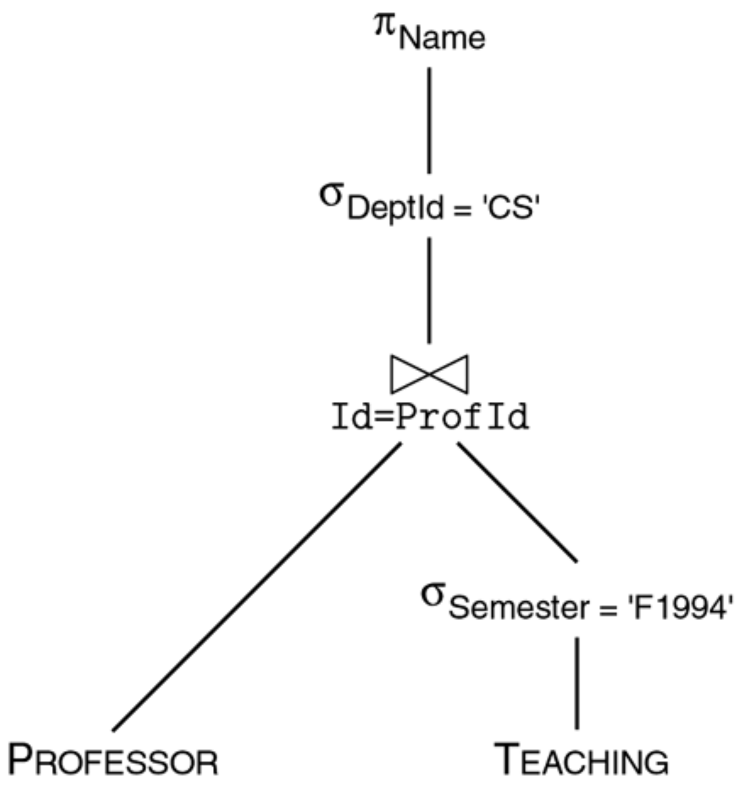
Today's lecture

This week and next week we cover KBL Chapter 5: SQL and relational algebra.

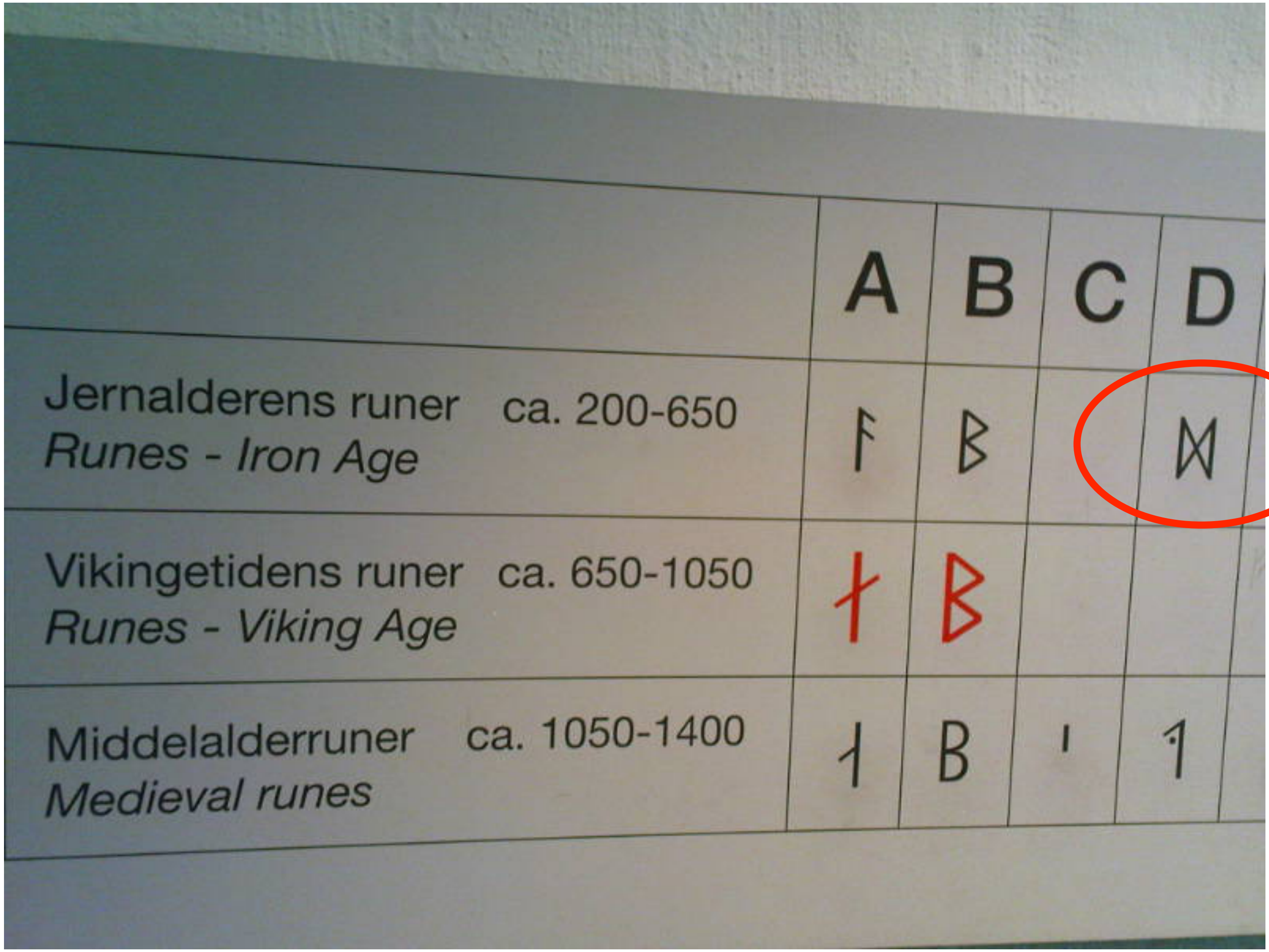
- SQL and relational algebra are relational query languages.
  - SQL is declarative: Describe **what** you want.
  - Relational algebra is procedural: Describe **how** to get what you want.

Relational algebra expression

(formatted as a tree)



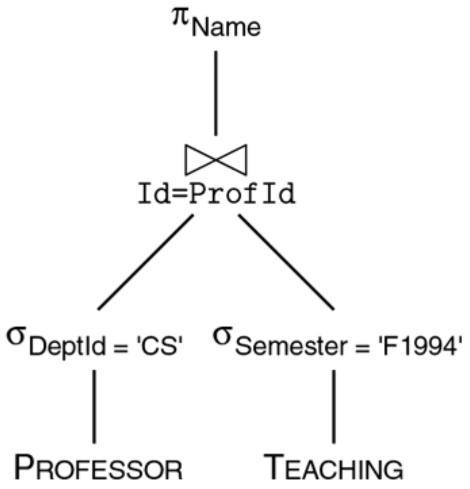
Greek letters, runes?



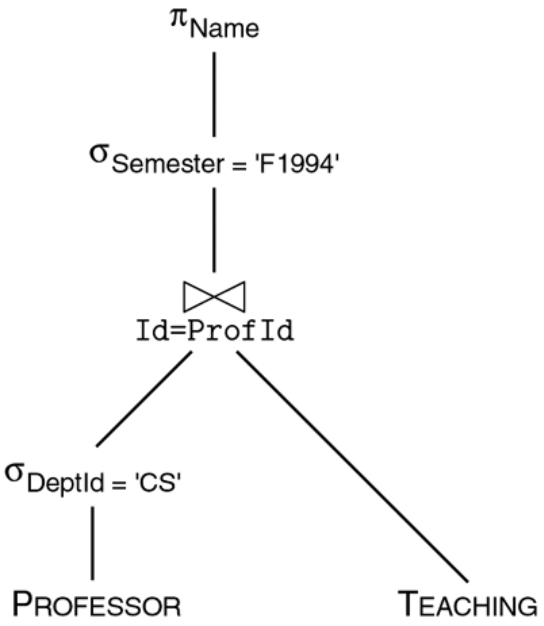
	A	B	C	D
Jernalderens runer ca. 200-650 *Runes - Iron Age*	ᚨ	ᛒ		ᛞ
Vikingetidens runer ca. 650-1050 *Runes - Viking Age*	ᛅ	ᛒ		
Middelalderruner ca. 1050-1400 *Medieval runes*	ᛆ	ᛒ	ı	ᛑ

Query tree

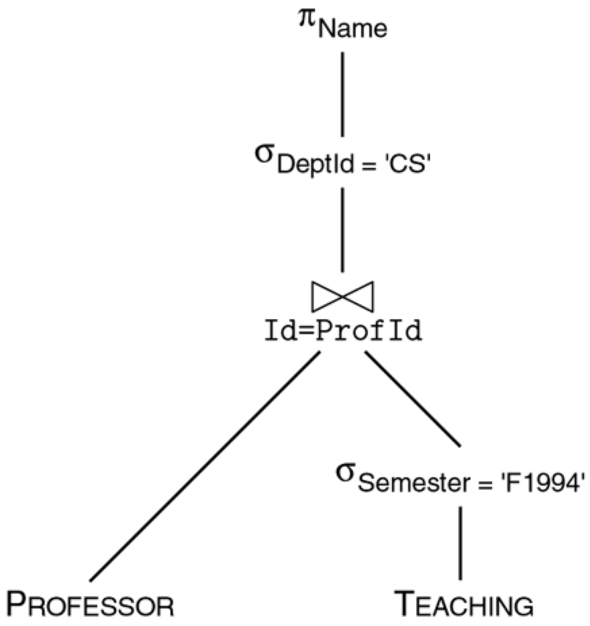
```
SELECT P.Name
FROM PROFESSOR P, TEACHING T
WHERE P.Id=T.ProfId AND T.Semester='F1994'
       AND P.DeptId='CS'
```



(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

Select

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

$$\sigma_{DeptId='CS'}(PROFESSOR)$$

```
SELECT *
FROM PROFESSOR
WHERE DeptId=‘CS’
```

Project

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

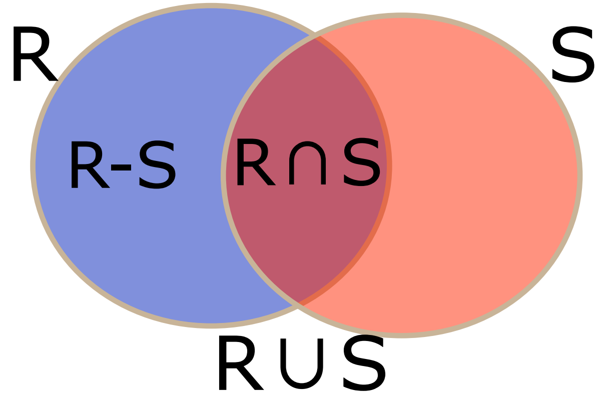
- Projection chooses a subset of attributes.
- The result of a projection is a relation with the attributes given in attribute list.
  By default the result is a **set**, i.e., contains no duplicates.

$$\pi_{Color}(Cars)$$

```
SELECT DISTINCT Color
FROM Cars
```

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

$$\sigma_{Color='Pink'}(Cars) \cup \sigma_{Color='Green'}(Cars)$$

All pink and all green cars

$$\pi_{Id}(PROFESSOR) \cap \pi_{Id}(STUDENT)$$

All *ID*s of professors for which there is a student with the same id.

Problem session

Assume that we have the relations

```
TRANSCRIPT(StudId,CrsCode,Semester,Grade)
TEACHING(ProfId,CrsCode,Semester)
```

What do these relational algebra expressions mean?

$$\pi_{CrsCode,Semester}(\sigma_{Grade='C'}(TRANSCRIPT))$$
$$\cap\pi_{CrsCode,Semester}(\sigma_{CrsCode='MAT123'}(TEACHING))$$

$$\pi_{CrsCode,Semester}(\sigma_{Grade='C'}(TRANSCRIPT))$$
$$\cup\pi_{CrsCode,Semester}(\sigma_{CrsCode='MAT123'}(TEACHING))$$

$$\pi_{CrsCode,Semester}(\sigma_{Grade='C'}(TRANSCRIPT))$$
$$-\pi_{CrsCode,Semester}(\sigma_{CrsCode='MAT123'}(TEACHING))$$

Cartesian product
(aka. cross product)

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

A subset of $\pi_{\mathtt{Id,Name}}(\textsc{Student})$

Id	DeptId
555666777	CS
101202303	CS

A subset of $\pi_{\mathtt{Id,DeptId}}(\textsc{Professor})$

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×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.

Cartesian product

- In SQL: `SELECT * FROM R,S;`
- If **R** has n tuples and **S** has m tuples, then **R×S** contain n·m tuples.
- Can be computationally expensive!

- Renaming necessary when **R** and **S** have attributes with the same name.
- Renaming is denoted by [name1,…] after an expression.

Join

$$R \bowtie_{joincondition} S$$

is equivalent to

$$\sigma_{joincondition}(R \times S)$$

Join example

(equi-join)

```
SELECT *
FROM Cars C, Owners O
WHERE C.Ownerid=O.Id
```

$$Cars \bowtie_{Ownerid=Id} Owners$$

$$\sigma_{Ownerid=Id}(Cars \times Owners)$$

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.

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
    - Car.Color=‘%green%’ is true for all colors with ‘green’ as a substring, e.g. ‘lightgreen’ ‘greenish’
  - 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

Expressions in SELECT

You can define new attributes using expressions:

```
SELECT C.Ownerid, T.Amount/12
FROM Car C, Cartax T
WHERE C.Color=‘Green’ AND C.Regnr=T.Regnr
```

You can give attributes new names:

```
SELECT C.Ownerid AS Id,
        T.Amount/12  AS MonthlyTax
```

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 SUM(T.Amount)
FROM Cartax T, Car C
WHERE T.Regnr=C.Regnr AND C.Ownerid=1234
```

```
SELECT COUNT(DISTINCT T.Amount)
FROM Cartax T, Car C
WHERE T.Regnr=C.Regnr AND C.Ownerid=1234
```

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, e.g. the total amount of tax each owner has to pay, use grouping together with aggregation:

```
SELECT C.Ownerid AS Id, SUM(T.Amount) AS TotalTax
FROM Cartax T, Car C
WHERE T.Regnr=C.Regnr
GROUP BY C.Ownerid
```

Grouping

The resulting columns can only be the aggregate or columns mentioned in the GROUP BY clause.

```
SELECT C.Ownerid AS Id, SUM(T.Amount) AS TotalTax
FROM Cartax T, Car C
WHERE T.Regnr=C.Regnr
GROUP BY C.Ownerid
```

Ownerid	Regnr
1234	1
1234	2
4321	3
8888	4
8888	5

Regnr	Amount
1	300
2	450
3	210
4	11
5	19

Id	TotalTax
1234	750
4321	210
8888	30

HAVING

```
SELECT C.OwnerId, SUM(T.Amount)
FROM Car C, Cartax T
WHERE C.Regnr=T.Regnr
GROUP BY C.OwnerId
HAVING SUM(T.Amount)<=1000
```

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

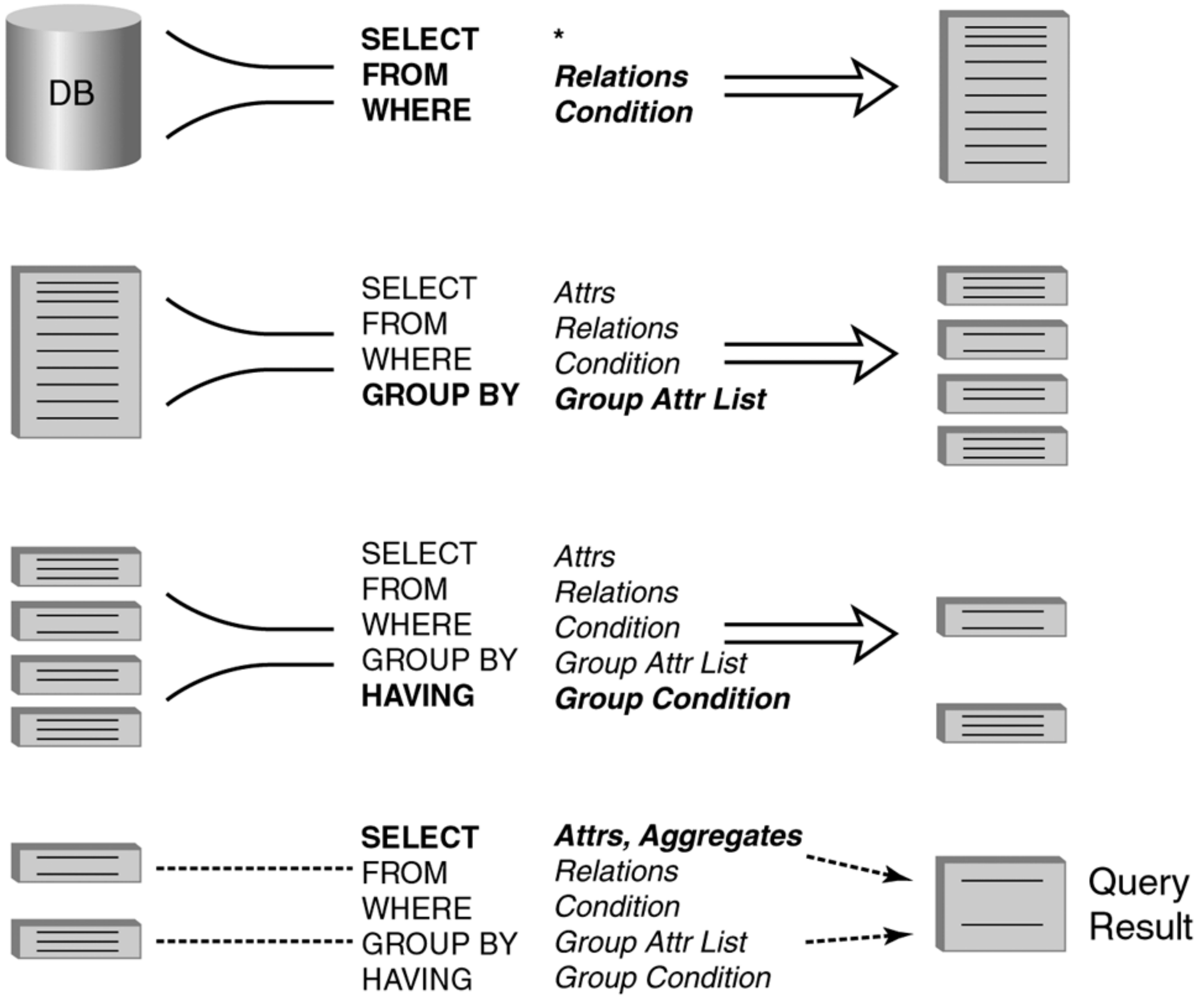
- Aggregates over tuples in group
- Conditions on tuple attributes

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...



SELECT *****
FROM ***Relations***
WHERE ***Condition***

SELECT *Attrs*
FROM *Relations*
WHERE *Condition*
GROUP BY ***Group Attr List***

SELECT *Attrs*
FROM *Relations*
WHERE *Condition*
GROUP BY *Group Attr List*
HAVING ***Group Condition***

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

Subqueries 1: In FROM clause

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

```
SELECT O.FirstName, O.LastName, TPO.TotalTax
FROM  Owner O,
      (SELECT Sum(T.Amount) AS TotalTax,
              T.OwnerId AS Id
       FROM Cartax T, Car C
       WHERE T.Regnr=C.Regnr
       GROUP BY C.Ownerid) AS TPO
WHERE TPO.Id=O.Id
```

Alternative syntax

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

```
WITH (SELECT Sum(T.Amount) AS TotalTax,
              T.OwnerId AS Id
      FROM Cartax T, Car C
      WHERE T.Regnr=C.Regnr
      GROUP BY C.Ownerid) AS TPO
SELECT O.FirstName, O.LastName, TPO.TotalTax
FROM  Owner O, TPO
WHERE TPO.Id=O.Id
```

Subroutines in SQL

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

```
CREATE VIEW OwnerColor AS
SELECT O.Id, C.Color
FROM Owner O, Car C
WHERE O.Id=C.Ownerid
```

The view can be used in different queries:

```
SELECT COUNT(*)
FROM OwnerColor O
WHERE O.Color=‘pink’
```

```
SELECT O.Color,COUNT(*)
FROM OwnerColor O
     GROUP BY O.Color
 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).

```
CREATE VIEW OwnerColor AS
SELECT O.Id, C.Color
FROM Owner O, Car C
WHERE O.Id=C.Ownerid
```

```
SELECT COUNT(*)
FROM OwnerColor OC
WHERE OC.Color='pink'
```

```
---------------------------------------------------
SELECT COUNT(*)
FROM (SELECT O.Id, C.Color FROM Owner O, Car C
WHERE O.Id=C.Ownerid) AS OC
WHERE OC.Color='pink'
```

Usage of views

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 OwnerColor TO ALL`

Meaning: All users can see the table OwnerColor, but not the underlying relations Car and Owner.

Other options:

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

Subqueries 2: In WHERE

```
SELECT C.Regnr
FROM Car C
WHERE C.Ownerid IN
    (SELECT O.Id
     FROM Owner O
     WHERE O.Lastname = ‘Sørensen’)
```

All registration numbers for cars owned by a person named Sørensen.

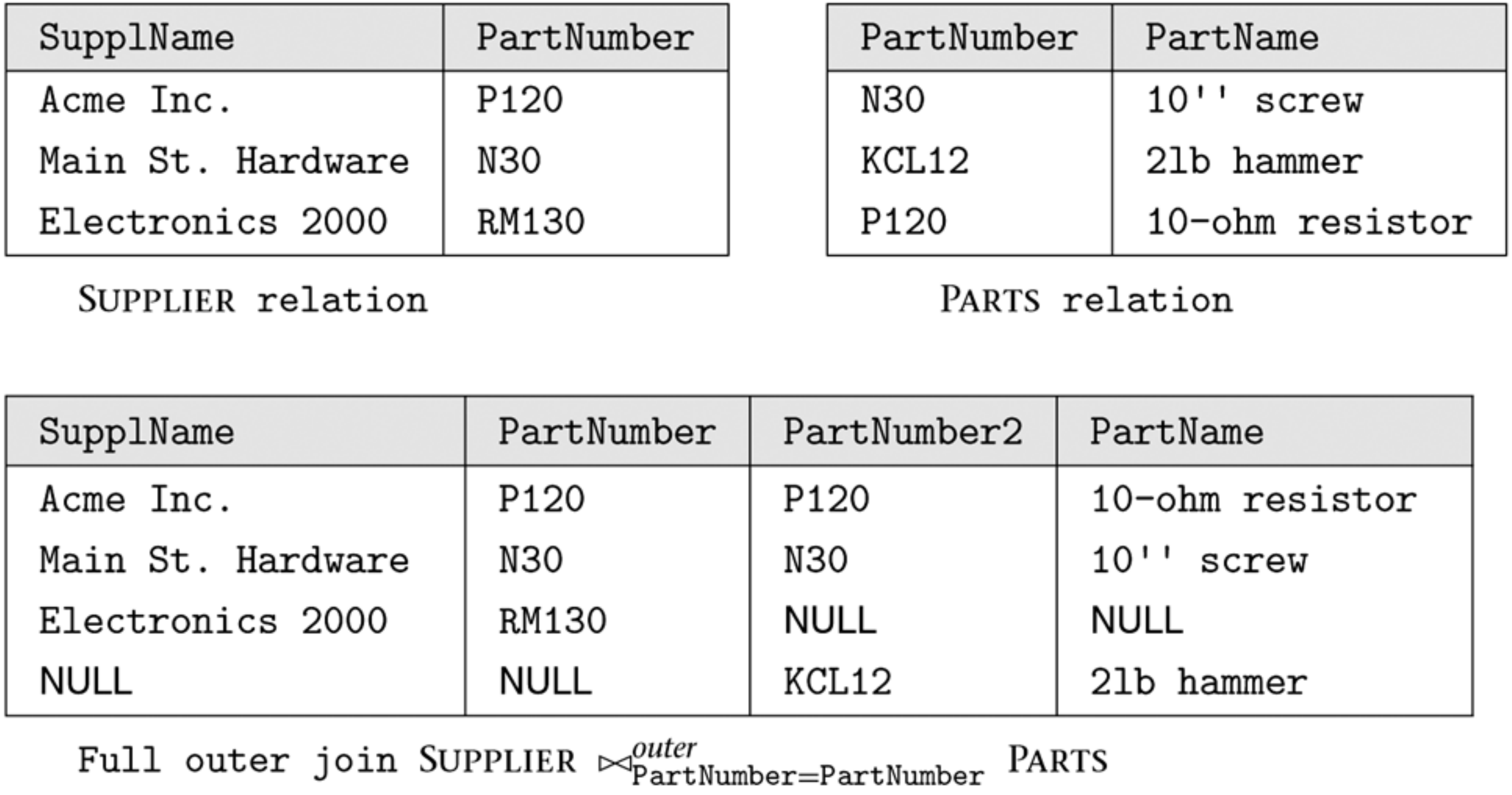
```
SELECT C.Regnr
FROM Car C, Owner O
WHERE C.Ownerid=O.Id AND O.Lastname=‘Sørensen’
```

Reverse example

```
SELECT C.Regnr
FROM Car C
WHERE C.Ownerid NOT IN
     (SELECT O.Id
      FROM Owner O
      WHERE O.Lastname = ‘Sørensen’)
```

Not expressible as a standard join!
(Assume Owner.id is a candidate key.)

(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 $\bowtie^{outer}_{\text{PartNumber}=\text{PartNumber}}$ 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

Subquery to define a value

A subquery producing a single value can be used as any other value (constant or attribute):

```
SELECT T.Regnr
FROM Cartax T
WHERE T.Amount =
     (SELECT T2.Amount
      FROM Cartax T2
      WHERE T2.Regnr='AB12345')
```

Note that the same table is used twice with two different tuple variables

If the subquery returns more than one tuple, a *runtime error* results.

Correlated subqueries

A subquery is said to be **correlated** when a variable in the outer query is used in the subquery:

```
SELECT R.Studid, P.Id, R.CrsCode
FROM TRANSCRIPT T, PROFESSOR P
WHERE R.CrsCode IN
      (SELECT T1.CrsCode
       FROM TEACHIN T1
       WHERE T1.ProfId=P.Id AND T1.Semester='S2009')
```

The inner query is evaluated **for each** P.Id.

Often **expensive** to evaluate correlated subqueries.

NOT EXISTS

```
SELECT O.Id
FROM Owner O
WHERE NOT EXISTS
     (SELECT C.Regnr
      FROM Car C
      WHERE C.Color LIKE ‘%green%’ AND
               C.Ownerid=O.Id)
```

True when subquery returns an empty relation

O is a **global variable** for the entire query, C is a **local variable** for the subquery.

Subquery “is” evaluated for each value of O.Id.

Problem session

What does the following query compute?

```
SELECT C1.Color, AVG(T.Amount)
FROM (SELECT O.Id AS Id
      FROM  Owner O, Car C2
      WHERE O.Id=C2.Ownerid
      GROUP BY O.Id
      HAVING COUNT(*)>8) AS Bigshots,
     Cartax T,
     Car C1
WHERE T.Regnr=C1.Regnr AND
      C1.Ownerid=Bigshots.Id
GROUP BY C1.Color
```

Beware of NULLs!

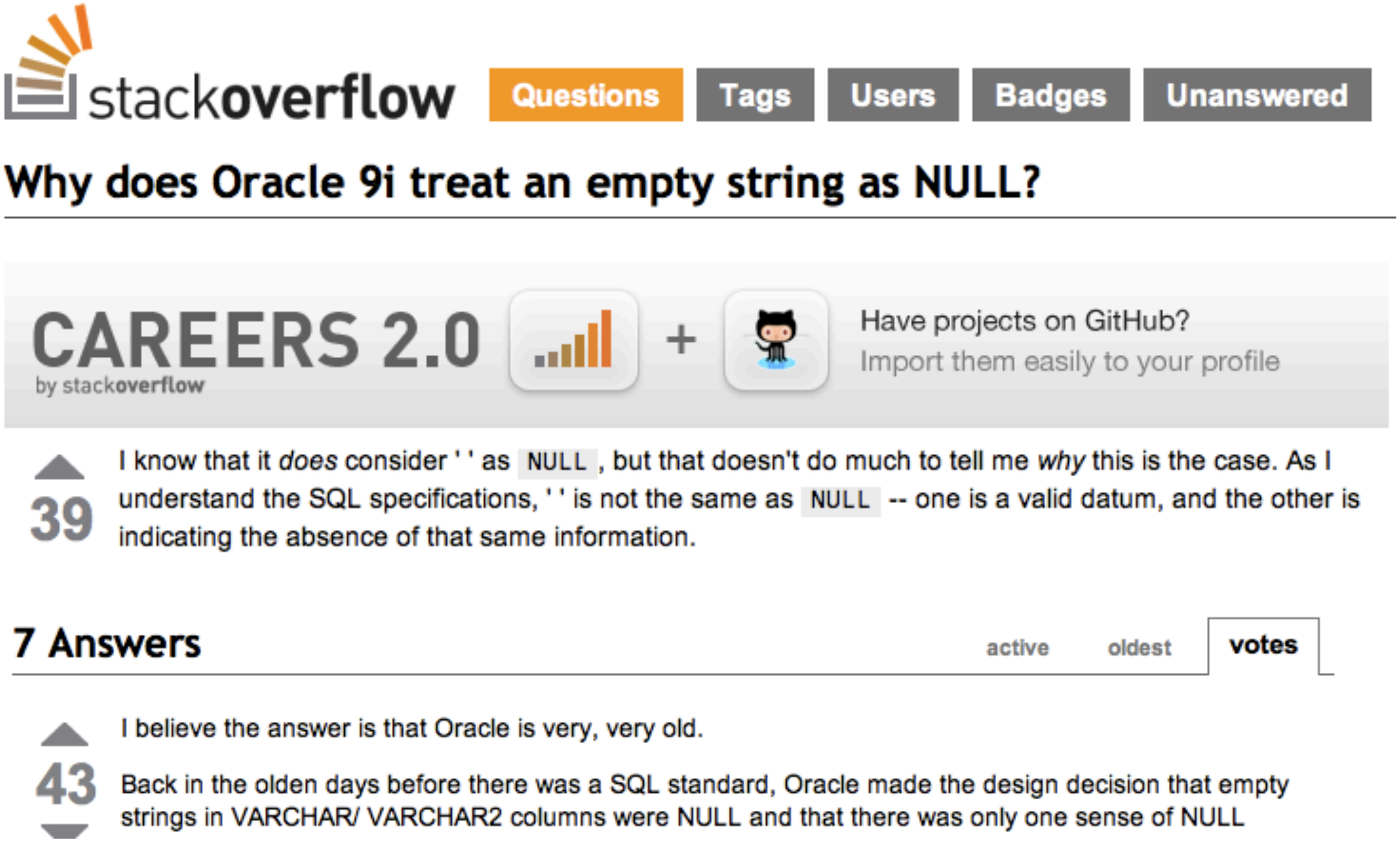
- Things are not always what they appear.
  - Aggregates treat nulls differently
  - Logic is different.
  - Different DBMSs handle NULLs differently…

- Demo:

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

Different behavior for NULL /empty string…

Beware of NULLs, cont.



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

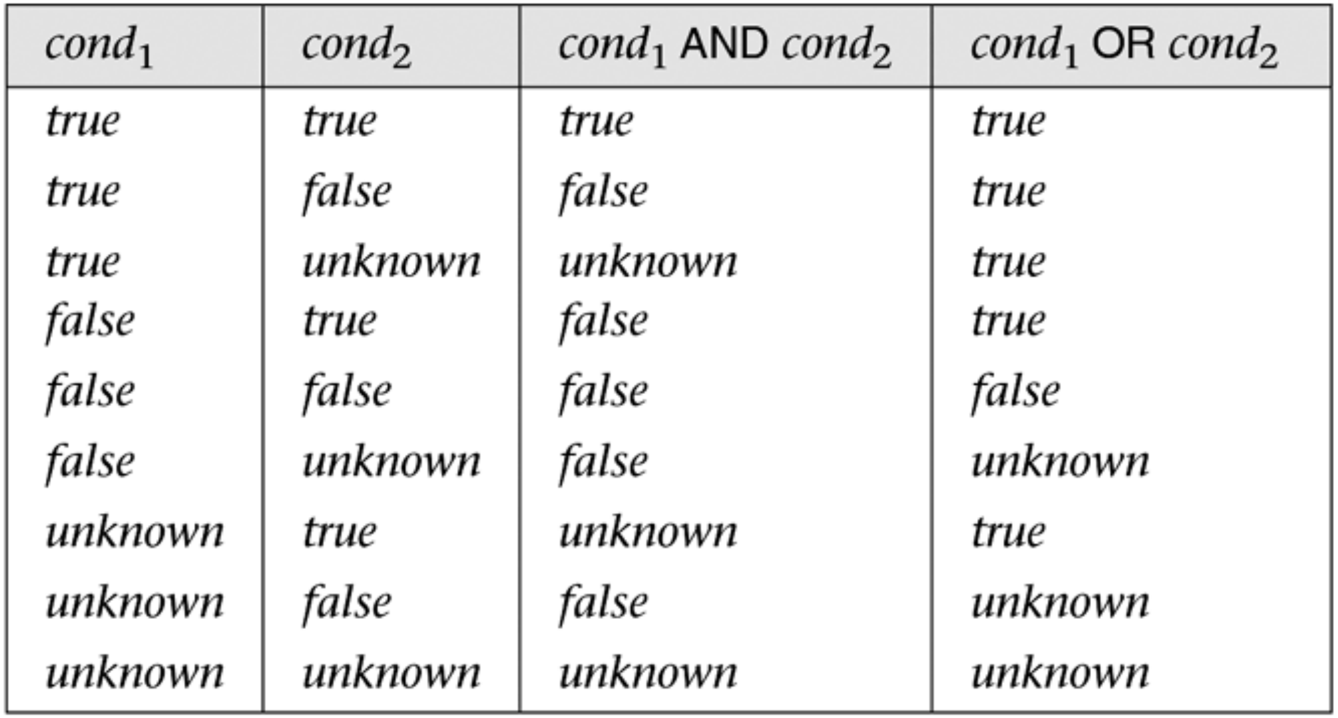
I know that it *does* consider ' ' as `NULL`, but that doesn't do much to tell me *why* this is the case. As I understand the SQL specifications, ' ' is not the same as `NULL` -- one is a valid datum, and the other is indicating the absence of that same information.

7 Answers

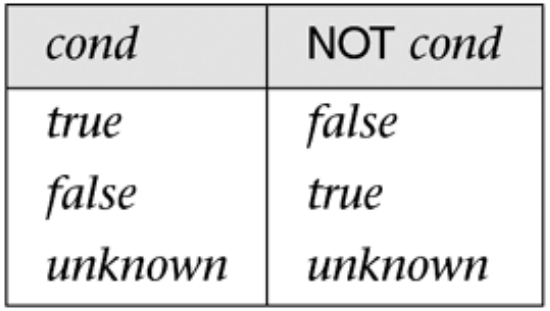
I believe the answer is that Oracle is very, very old.

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 ProfNameDept(Name,DeptId) AS
SELECT P.Name, P.DeptId
FROM Professor P
```

What are the results of the following 2 updates?

```
INSERT INTO ProfNameDept
VALUES (Hansen, ‘CS’)

DELETE FROM ProfNameDept
WHERE Name=Hansen and DeptId=‘CS’
```

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!

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:

- express simple relational expressions using the relational algebra operators select, project, join, intersection, union, set difference, and cartesian product.
- write SQL queries, involving multiple relations, compound conditions, grouping, aggregation, and subqueries.