Lecture 13

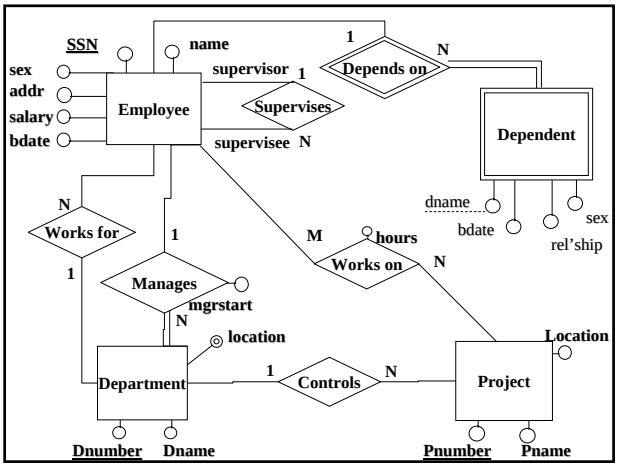
SQL as a Query Language

Week 6

SQL as a query language

Suppose that we have a schema, and a database full with data (already in the database). We use the COMPANY schema from the textbook as our running example.

We now *query* the database (we may not even have the access privilege to modify the content of the database!)



All queries start with...

SELECT <attribute list>
FROM <table list>

NB: ***not*** the same as relational select (σ) !!

the rest is optional, e.g.

SELECT <attribute list>
FROM <table list>
WHERE <selection condition>

Simplest query

Query 1. Select the birthdate and address of each employee whose name is 'James Brown'

SELECT bdate, address
FROM employee
WHERE name = 'James Brown'

(almost) equivalent relational algebra expression (*almost*, because duplicates are *not* automatically dropped in SQL):

$\Pi_{bdate,address}\ \sigma_{name='James\ Brown'}\ Employee$

The evaluation of an SQL query

(in reality implemented in more efficient way)

- Form a Cartesian product of the tables in the <table list>
- Select all the tuples from the result, that satisfy the <selection condition>
- Project onto the <attribute list>

(do not remove duplicate tuples unless specifically instructed to do so)

Example with more then one table

Query 2. Select the name and address of each employee who works for a department called 'Research'

```
SELECT E.name, E.address
FROM employee E, department D
WHERE D.dname = 'Research'
    AND E.dno = D.dnumber
```

Selection condition

Join condition

(almost) equivalent relational algebra expression:

$\Pi_{\text{name, address}}\ \text{Employee} *_{\text{dno=dnumber}} \Pi_{\text{dnumber}} \sigma_{\text{dname='Research'}}\ \text{Department}$

Tuple variables E and D are like pointers 'scanning' the tables employee and department

Query 2 (cont'd)

```
SELECT E.name, E.address
FROM employee E, department D
WHERE D.dname = 'Research'
    AND E.dno = D.dnumber
```

Notice in Query 2 the use of *tuple variables* E and D.
Note that some people call E and D an ***alias*** [to employee, and department respectively]

Some remarks

- Tuple variable names are not always necessary, but always a good idea
- It is possible to use table names instead of tuple variable names (unless the same table appears in the from clause more then one time) - i.e. one can say either **E.name** or **employee.name**.
- The WHERE clause is not always necessary, but the SELECT and FROM clauses must always be present.

When tuple variables are needed...

Query 3. Select the name of each employee and their supervisor's name

```
SELECT E.name, S.name
FROM employee E, S
WHERE E.superssn = S.ssn
```

- In case the same table appears more then once in the FROM clause, we need tuple ***variables*** to distinguish between references to attributes.

Note: If two tables in the FROM clause have attributes with identical names, to distinguish between these we can use *either* a tuple variable *or* the table name as a qualifier of the attribute.

Remark...

In Query 3:

```
SELECT E.name, S.name
FROM employee E, S
WHERE E.superssn = S.ssn
```

The result of this query will have two identically named columns - name and name. To be able to make a difference between these columns we need to rename them...

Remark (cont.)…

…therefore SQL allows you to rename the columns in the result, e.g.:

```
SELECT E.name AS employee_name , S.name
FROM employee E, employee S
WHERE E.superssn = S.ssn
```

NULL values in comparison

- NULL values fail all comparison tests. SQL has special tests for NULL values: 'IS NULL' and 'IS NOT NULL'

  e.g. if E.salary has a NULL value then

  E.salary >30000 -- false
  E.salary = 30000 -- false
  E.salary < 30000 -- false

  E.salary IS NULL -- true

Example for NULL test

Query 4: List the names of employees who have no supervisor

```
SELECT E.name
FROM employee E
WHERE E.superssn IS NULL
```


Examples

- To select all attributes of a relation

  ```
  SELECT * FROM employee
  ```

- To return distinct tuples (i.e. eliminate duplicates from the result)

  ```
  SELECT DISTINCT E.salary
  FROM employee E
  ```


Set Operations

Union, Intersect, Minus, Contains (⊇), in (∈), Exists, Not Exists

Originally System R (the first IBM research prototype implementation of a relational DBMS) had all set operators, but commercial systems do not implement the contains (⊇) operator and it is missing from SQL)

NB Tables must be union compatible!

Note: duplicates are eliminated in set operations

Example for set operator

Query 5. List the numbers of projects who have an employee working on it such that the employee is called 'Smith', or the manager of the department controlling the project is called 'John Smith'

```
SELECT W.pno
FROM    works_on W, employee E
WHERE   W.essn = E.ssn and
        E.name = 'John Smith'
UNION
SELECT P.pnumber
FROM department D, employee E, project P
WHERE   P.dnum = D.dnumber and
        D.mgrssn = E.ssn and
        E.name = 'John Smith'
```


Nested queries (using the 'IN' operator)

Query 6. Select the name and ssn of employees who have a dependent

```
SELECT E.name, E.ssn
FROM employee E
WHERE E.ssn IN
    ( SELECT D.essn
      FROM dependent D)
```

This test looks at every tuple in the employee relation and tests if E.ssn for that tuple is in the set (or bag) returned by the nested (inner) query.

The nested query returns the ssn column of employee (with as many repetitions of each ssn as many dependents the given employee has)

Nested queries (cont.)

```
SELECT E.name, E.ssn
FROM employee E
WHERE E.ssn IN
    ( SELECT D.essn
      FROM dependent D)
```

Note that this nested query needs to be evaluated only once by the DMBS. The result of this evaluation is the set of employee ssn-s which appear in the table 'dependents'.

(The condition whether E.ssn is in this set is evaluated as many times as there are employees. However, the set itself needs to be determined only once).

Testing for empty set / non-empty set

Query 7. List the name and ssn of employees who have a dependent with the same name as the employee

```
SELECT E.name, E.ssn
FROM employee E
WHERE EXISTS
  (SELECT * FROM dependent D
   WHERE E.ssn = D.essn AND
         E.name = D.dname)
```

Tests whether the nested query returns a non-empty set (bag)

Note that for each employee E the nested query returns the set of dependents with the desired property

Query 8. List the names of those employees who have no dependent

```
SELECT E.name
FROM employee E
WHERE NOT EXISTS
  (SELECT * FROM dependent D
   WHERE E.ssn = D.essn)
```

Here, as opposed to **Query 6**, the nested query must be evaluated by the DBMS as many times as there are employees, because the result of the nested query depends on the tuple variable E).

Using nested queries to eliminate duplicate tuples

- Notice that the result of Query 6 will NOT have any repeating tuples, because the outer query merely tests for each employee in turn if the ssn is in the set (actually bag) of essn-s created by the nested (inner) query
- This is an effective way of controlling the appearence of duplicate tuples in the result

How are nested queries evaluated?

- The outer query is evaluated
- For each tuple that the outher query produces the WHERE clause is evaluated
- If the where clause contains a test that involves a nested query then the nested query is evaluated to be able to perform the test
  - If the nested query's result is independent from the tuple variables of the outer query, then the nested query needs to be evaluated only once (as in Query 6)
  - If the nested query's result depends on at least one of the tuple variables of the outer query, then the nested query must be repeateadly evaluated for each tested tuple (as in Queries 7 and 8)

Queries which involve the universal quantifier

Query 9. 'Select employees who work on all projects'

```
SELECT * from employee E
WHERE
    (SELECT W.pno from works_on W
     WHERE E.ssn = W.essn)
CONTAINS
    (SELECT P.pnumber from project P)
```

NOT IN SQL!

So what do we do?

Transforming a query with a universal quantifier

(using existential quantifier and nested queries instead)

"Select employees who work on all projects" is the same as "Select employees 'x' such that if 'p' is a project then 'x' works on 'p' "

With the universal quantifier this can be written as

$\forall x (employee(x) \wedge$
$\forall p (project(p) \rightarrow works\_on(x,p)))$

Further transforming this query

"Select employees for which there does not exist a project such that the employee does not work on it"

$\forall x (employee(x) \wedge$
$\forall p (project(p) \rightarrow works\_on(x,p)))$

$\equiv$

$\forall x (employee(x) \wedge$
$\neg\exists p (project(p) \wedge \neg works\_on(x,p)))$

Further transforming this query

'Select employees for which there does not exist a project such that the employee does not work on it'

$\forall x (employee(x) \wedge$
$\neg\exists p (project(p) \wedge \neg works\_on(x,p)))$

This transformation uses two facts of logic

$\forall x\, c(x) \equiv \neg\exists x \neg c(x)$
$\neg (a \rightarrow b) \equiv a \wedge \neg b$

And in SQL...

"Select employees for which there does not exist a project such that the employee does not work on it"

```
SELECT  * FROM EMPLOYEE E
WHERE   NOT EXISTS
  ( SELECT * FROM project P  // projects not worked on
        WHERE   NOT EXIST
          (SELECT * FROM works_on W
           WHERE W.pno = P.pnumber and E.ssn = W.essn))
```


To be continued in lecture 14...

- Aggregates, updates, views
- Don't forget to practice, try solving the more complex queries in the textbook

The end