Lecture 14

Data Manipulation in SQL (cont'd) Aggregates, Updates and Views

Week 8

Aggregate functions

Query 10. List the sum, average, minimum and maximum of salary values of employees

SELECT
SUM(E.salary), **AVG**(E.salary),
MIN(E.salary), **MAX**(E.salary)
FROM employee E

Simple Aggregate Query Evaluation by the DBMS

Step 1. Evaluate query without aggregate function to determine columns that participate in the aggregate

employee

ssn	salary	
1234567	70,000	
5670349	72,000	…
7562057	83,000	
6594774	47,500	

1

salary
70,000
72,000
83,000
47,500

2

Step 2. Apply aggregate function

SUM(sal)	**AVG(sal)**	**MIN(sal)**	**MAX(sal)**
272,500	68,125	47,500	83,000

Aggregate Query Evaluation (cont.)

Note that the result of evaluating Query 10 (or any other simple aggregate query) contains <u>only one</u> tuple!

Consider

SELECT

~~E.ssn~~, SUM(E.salary), AVG(E.salary), MIN(E.salary), MAX(E.salary)

FROM employee E

This query is wrong (has no meaning) - we can not decide which employee’s ssn to include in the result

ssn	**SUM(sal)**	**AVG(sal)**	**MIN(sal)**	**MAX(sal)**
????	272,500	68,125	47,500	83,000

Aggregate of Groups
(the GROUP BY clause)

Query 11. List the department number and the average salary for **each** department

```
SELECT D.dnumber, AVG(E.salary)
FROM department D, employee E
WHERE E.dno = D.dnumber
GROUP BY D.dnumber
```

GROUP BY Query evaluation

Step 1. Evaluate query without aggregate function (including where conditions!) creating a group of tables

Employee X Department

Dnumber	Salary
1	70,000
1	72,000
2	83,000
2	47,500

…

Create a table for each value of D.dnumber

Employee X Department

Dnumber	Salary
1	70,000
1	72,000

Dnumber	Salary
2	83,000
2	47,500

…

Step 2. Apply aggregate function to each such table

Dnumber	**AVG (Salary)**
1	**71,000**
2	**65,250**

NB Only GROUP BY and aggregate attributes can be present in the SELECT clause!

GROUP BY Query evaluation (cont.)

Step 1. Evaluate query without aggregate function (including where conditions!) creating a group of tables

Create a table for each value of D.dnumber

Employee X Department

Dnumber	Salary
1	70,000
1	72,000
2	83,000
2	47,500

…

Employee X Department

Dnumber	Salary
1	70,000
1	72,000

Dnumber	Salary
2	83,000
2	47,500

…

NOTE

The <u>where condition can not refer to aggregate attributes</u>, since the aggregation has not been been carried out yet

Conditions on the result of the grouped aggregate (HAVING clause)

Query 12. List the average salary of those departments where the average salary is greater then 70,000

```
SELECT D.dnumber AVG(E.sal)
FROM department D, employee E
WHERE E.dno = D.dnumber
GROUP BY D.dnumber
HAVING AVG(E.sal) > 70,000
```

Employee X Department

Dnumber	Salary
1	70,000
1	72,000

Dnumber	Salary
2	83,000
2	47,500

…

Step 2. Apply aggregate function and apply the condition of the HAVING clause

Dnumber	AVG(Salary)
1	**71,000**

Important! ‘where’ vs. ‘having’

- The WHERE clause eliminates tuples from the cross product (of tables in the FROM clause) *before* the GROUP BY clause is applied
- The HAVING clause eliminates tuples from the table produced *after* the GROUP BY clause is applied

Two Implementations of One Query

Query 13. List department names and the number of employees who earn >40,000 in each department, provided that the department has > 5 employees

```
SELECT D.dname, count(*)
FROM department D, employee E
WHERE D.dnumber = E.dno AND
          E.salary > 40000

GROUP BY D.dname

HAVING count(*) >5
```

This query eliminates employees from each department if the employee earns less then 40,000

After this the GROUP BY statement will only see employees in each department who earn >40,000

Thus, count(*) will count the number of employees in the department who earn >40,000 and the HAVING clause will eliminate those from the result where the number of employees earning >40,000 is not >5. Thus this query is WRONG

Two Implementations of One Query

Second (and correct) implementation.

```
SELECT D.dname, count(*)
FROM department D, employee E
WHERE D.dnumber = E.dno AND
          E.salary >40000 AND
          E.dno IN
           (SELECT EM.dno
            FROM employee EM
            GROUP BY EM.dno
            HAVING count(*) > 5)
GROUP BY D.dname
```

The nested query ensures that only those employees are considered, who work for a department which has >5 employees.

ORDER BY clause

- It is possible to order the returned tuples by any columns(s). E.g.:

  SELECT E.name, E.salary, E.address
  FROM employee E
  ORDER BY E.name **ASC**, E.salary **DESC**

  orders result by E.name and for employees with the same name they are ordered by E.salary

ASC: ascending (default, can be omitted)
DESC: descending

Insert / Delete and Update

Example: Insert a new employee into the employee table:

```
INSERT INTO employee
VALUES
   ('John Smith', 12345678, null, null, null, null)
```

- the tuple must have the same order of attributes as defined in the schema
- attributes not supplied are set to NULL (SQL2)
- attributes not supplied must be declared as NULL (Oracle SQL*Plus)

Inserting a set of Tuples

- Insertion is a set-based operation
- Suppose we have a schema:

department_sal (dno, salary_average, date)

We can insert into this table the result of a query:

```
INSERT INTO department_sal
    SELECT E.dno, AVG(E.salary), "2 FEB 2005"
    FROM employee E
    GROUP BY E.dno
```

Deletion

Deletion is a set-based operation. It must be defined *which* tuples to delete

Example. Delete all employees of department number 5

```
DELETE FROM employee E
WHERE E.dno = 5
```

To delete a Unique Tuple...

We must identify the tuple (e.g. by a key value).

Example. Delete employee with name ‘J.S’ and ssn 1234567

```
DELETE FROM employee E
WHERE E.name = ‘J.S’ AND
          E.ssn = 1234567
```

Updating a Relation

```
UPDATE employee E
SET E.salary = E.salary*1.1
```

gives a salary raise to all employees

```
UPDATE employee E
SET E.salary = E.salary*1.1 AND E.dno = 6
WHERE E.dno = 5
```

gives a salary raise to employees in department 5, and transfers them to department 6

Update vs. Delete + Insert

- It is possible to simulate the effects of an update statement with a deletion (delete the old tuples) followed by and insertition (add tuples with the new values)

- The two solutions are not entirely equivalent, because the database state between the delete and insert *may* be visible to others and/or may be inconsistent. As opposed to this the update is always done in one transaction.

SQL Views

- An SQL view is a form of external schema
- An SQL view does not describe a *real* table, it is a *virtual* table, or *derived* relation
- For the user of the database (at least for querying purposes) a view is just like a table. (Views can be used to define external schemas for applications)
- The data in a view may or may not explicitly exist in the database (e.g. aggregate values)

To Define a View….

```
CREATE VIEW
    department_sal_view (dname, salary_average)
AS
    SELECT E.dno, AVG(E.salary)
    FROM employee E
    GROUP BY E.dno
```

Note: A view definition may refer to other, already defined views

Executing a Query on a View

SELECT * FROM department_sal_view

- The DBMS translates this query, *based on the view definition* into a, equivalent query that only mentions base tables
  SELECT E.dno, AVG(E.salary) FROM employee E GROUP BY E.dno

- If the base tables change the result of the queries on views automatically change (compare this with the base table department_sal into which we inserted the current result of the query - see example at insertion)

- The department_sal table is sometimes called a ‘materialised view’

Materialised Views

- Materialised views need to be updated each time a base table on which they depend is updated
- Whether to use views or materialised views depends on the application and is mostly an efficiency question
- Views are easier to maintain than materialised views

Deleting a View Definition

```
DROP VIEW department_sal_view
```

Updating Views

Normally difficult, only special cases are possible (meaningful).

For example, let us define a view:

```
CREATE VIEW works_on_view (emp, proj, hrs)
AS
  SELECT E.name, P.pname, W.hours
  FROM employee E, project P, works_on W
  WHERE E.ssn = W.essn AND
                P.pnumber = W.pno
```

The View Update Problem

Views are difficult to update, because of the ***ambiguity*** that arises from such attempt.
E.g. Suppose that at this moment 'John Smith' works on 'Project X'.
Try to execute:

```
UPDATE works_on_view W
SET W.pname = 'ProjectY'
WHERE W.emp = 'John Smith'
```

What could be the intended meaning of this?

The View Update Problem (cont.)

- Potential meaning **#1**:
  We want 'John Smith' to work on 'Project Y' instead of 'Project X'.

  We have to update the underlying Works-on base relation, changing the project number attribute to reflect that J.S. now works for a different project

Ambiguity: we can not decide what was the *intended* meaning

- Potential meaning **#2**
  We want the name of the project 'Project X' to be changed to 'Project Y'

The View Update Problem (cont.)

Similar ambiguity arises if the view contains aggregate attributes (e.g. average salary). For example:

What does it mean that we increase the average salary by 10,000? Give an ‘across the board’ raise or give a raise to some employees?

Answer: it is not decidable.

Solution 1 to the View Updates

We don’t allow it, unless

1. The view is defined on a single base table
2. The view does not contain aggregate attributes

Theoretically, the class of permissible (unambiguous) updates on views is somewhat larger, everywhere where it is possible to prove that the update can only effect at most one base relation tuple in each base relation involved.

Solution 2 to view updates…

- If we want to allow view updates where there is ambiguity, then we must define what is the *intended* operation on the base tables
- It is possible to do this in SQL, using *triggers*

Example:

Similar triggers can be written for 'instead of insert' and 'instead of delete'

```
CREATE TRIGGER works_on_view_update_trigger
INSTEAD OF UPDATE ON works_on_view
FOR EACH ROW
BEGIN
    <SQL update statement comes here to update the underlying works_on table>
END
```

NB The treatment of triggers is not part of this introductory course, but it is logical to at least mention them here

Summary of SQL Query Evaluation

- Create cross product of tables in the FROM clause
- Evaluate the WHERE clause
- On the remaining relation create groups according to the GROUP BY clause
- Evaluate aggregates
- Test tuples in the result if they satisfy the HAVING clause
- Eliminate duplicates if prescribed by DISTINCT, and print the result according to the SELECT clause
- Order result according to ORDER BY clause

The end