
Introduction to Databases, Fall 2005
IT University of Copenhagen

Lecture 6, part I: More on SQL

October 10, 2005

Lecturer: Rasmus Pagh

— Today's lecture —

Group hand-in 2: Presentation and questions.

Part I: More on SQL

- Subqueries in SQL.
- Views in SQL.

Part II: OLAP and data cubes (next slide set)

- Information integration (e.g. data warehousing).
- OLAP.
- Data cubes and Relational OLAP.

— What you should remember from previously —

In this lecture I will assume that you remember:

- The SQL concepts from the first two lectures:
 - Projection and selection using `SELECT-FROM-WHERE`.
 - `SELECT-FROM-WHERE` involving multiple relations.

Subqueries

Until now, you have seen SQL queries of the form

```
SELECT <list of attributes>  
FROM <list of relations>  
WHERE <condition>
```

What we haven't used is that:

- In any place where a relation is allowed, we may put an SQL query (a **subquery**) computing a relation.
- In any place where an “atomic value” (e.g. string or integer) is allowed, we may put an SQL subquery computing a relation with one value of this type (i.e., having one attribute and one tuple).

— Subqueries in FROM clauses —

Instead of just relations, we may use SQL queries in the FROM clause of SELECT-FROM-WHERE.

If we need a name for referring to the relation computed by the subquery, a tuple variable is used.

Subqueries should always be surrounded by parentheses.

— Subqueries producing scalar values —

When a query produces a relation with one attribute and one tuple, it can be used in any place where we can put an atomic (or **scalar**) value.

Semantics:

In places where an atomic value is expected, SQL regards a relation instance containing one atomic value x to be the same as the value x .

If such a subquery does not result in exactly one tuple, it is a **run-time error**, and the SQL query cannot be completed.

— Subqueries in conditions —

One common use of subqueries is in the WHERE part of SELECT-FROM-WHERE. There are several operators in SQL that apply to a relation R and produce a boolean result:

- EXISTS R is true if and only if R is not empty.
- s IN R is true if and only if s is a tuple in R .

If R is **unary** (has just one attribute):

- s > ALL R is true if and only if s is greater than *all* values in R .
- s > ANY R is true if and only if s is greater than *some* value in R .

... and similarly for other comparison operators (<, >=, <=, <>).

— Correlated subqueries —

Sometimes a subquery depends on (is **correlated** with) tuple variables/relations of the surrounding SELECT-FROM-WHERE.

A correlated subquery *would not make sense on its own*, typically because it refers to attributes not in the relations of the FROM part of the query.

Semantics:

The query is evaluated once for each **binding** of tuple variables in the surrounding SELECT-FROM-WHEREs.

Scoping rule:

In case several tuple variables/relations have the same name x , an occurrence of x refers to the *closest* such tuple variable/relation.

Semantics of SELECT-FROM-WHERE

```
SELECT *  
FROM E1 V1, E2 V2, ..., Ek Vk  
WHERE <condition>
```

1. Determine the values of E_1, \dots, E_k (which are subqueries or relations).
2. Form all possible combinations of tuples V_1, \dots, V_k , where V_1 comes from E_1 , etc:
 - (a) For each combination determine if $\langle \text{condition} \rangle$ is true.
This may involve computing subqueries. If (attributes from) one of V_1, \dots, V_k is referred to in a subquery and is *not* a tuple variable in the subquery, we substitute in the appropriate value.
 - (b) If the condition is true, we output the combination of V_1, \dots, V_k .

— Problem session (5 minutes) —

What does the below SQL query compute?

```
SELECT title, year
FROM Movie
WHERE EXISTS (SELECT *
              FROM Movie M2
              WHERE Movie.year = year + 1
              AND EXISTS (SELECT *
                        FROM Movie M3
                        WHERE M2.year = year + 1));
```

Tip: Read from inside out.

— Subqueries and efficiency —

Generally speaking, queries involving subqueries are more difficult to deal with for a DBMS.

A good DBMS will execute most such queries efficiently, but there may be performance problems, especially for correlated subqueries.

Some, but not all, queries can be rewritten as a `SELECT-FROM-WHERE` with no subquery. This will sometimes improve efficiency.

Example: “Return all students that do not yet have any registered grade.”

In some cases, such as the above, a correlated subquery can be avoided by using SQL’s set operators. (After the fall break.)

Next: Views in SQL

Views

Views are queries that have been given a name.

Syntax for declaring a view:

```
CREATE VIEW <name of view> AS <SQL query>
```

We may use the name of a view in SQL expressions, as a *shorthand* for the corresponding query.

— Properties of views —

- Views are elements of the database schema, just like relation schemas.
- Privileges to access a view are handled just like privileges for relations.
- The privileges to perform the query must be held by the user who *defines* the view, but not necessarily by users accessing the view.
- Sufficiently simple views can be modified, meaning that the the modifications are passed on to the underlying relations.

— Materialized views —

Materialized views are views that are physically stored, i.e. stored relations that are results of queries.

Syntax for declaring a materialized view in Oracle:

```
CREATE MATERIALIZED VIEW <name of view>  
AS <SQL query>
```

Differences from an ordinary view:

- Allows faster access, as the query result is always computed.
- Needs to be updated when the underlying relations change.

— Most important points in this lecture —

As a minimum, you should after this week:

- Be able to understand and form SQL expressions using several levels of subqueries.
- Be able to define and use views in SQL.
- Understand the mechanism for granting and revoking privileges in SQL.