Forelæsning 9: Mere om SQL

30. marts 2006

Forelæser: Esben Rune Hansen
Today’s lecture

- Subqueries in SQL.
- Set operators in SQL.
- Security and authorization in SQL.
- Object orientation in databases and database modeling.
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 is allowed, we may put an SQL subquery computing a relation with one attribute and one tuple.
Subqueries in \texttt{FROM} clauses

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

\textbf{EXAMPLE:} \texttt{SELECT a FROM (SELECT a,b FROM (SELECT a,b,c FROM mytable));}

\textbf{ILLUSTRATION:}

\begin{align*}
\begin{array}{c|c|c}
0 & a & b \\
\end{array} & \leftarrow & \begin{array}{c|c|c|c}
0 & a & b & c \\
\end{array} \leftarrow \begin{array}{c|c|c|c}
0 & a & b & c \\
\end{array}
\end{align*}
Subqueries in WHERE clauses

We may also use SQL queries instead of relations in the FROM clause of SELECT–FROM–WHERE.

**EXAMPLE:** SELECT a, b FROM table1 WHERE a IN (SELECT a FROM table2 WHERE c > 0.2)

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

**EXAMPLE:** SELECT A.b, B.b FROM (SELECT b FROM table1 WHERE b = 5) A, (SELECT b, c FROM table2 WHERE c = 5) B

**Note:** Subqueries should always be surrounded by parentheses.
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.

Stated informally:

\[
\begin{array}{c|c}
\text{a} & 10 \\
\hline
10 & \\
12 & \\
\end{array}
\]

\( = 10 \) is true,

\[
\begin{array}{c|c}
\text{a} & 10 \\
\hline
10 & \\
12 & \\
\end{array}
\]

\( = 10 \) results in a run-time error.
Consider the table $t$:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>70</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>42</td>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

What is the result of each of the following:

- SELECT $a,b,c$ FROM (SELECT $c,b,a$ FROM $t$)
- SELECT $a,b,c$ FROM (SELECT * FROM $t$ where $1 = 2$)
- SELECT $a$ FROM $t$ where $a \geq$ (SELECT $c$ FROM $t$ where $b = 42$)
- SELECT $a,b,c$ FROM $t$ where $c >$ (SELECT $a$ FROM $t$ where $c = 42$)
- SELECT $a,b,c$ FROM $t$ where $c >$ (SELECT $a$ FROM $t$ where $c = 42$)
- SELECT * FROM (SELECT $a,b$ FROM $t$ where $a < b$) where $a > b$
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):

- **\( s > \text{ALL} \ R \)** is true if and only if \( s \) is greater than all values in \( R \).

- **\( s > \text{ANY} \ R \)** is true if and only if \( s \) is greater than some value in \( R \).

... and similarly for other comparison operators (\(<\), \(\geq\), \(\leq\), \(<\text{>}\)).
**Correlated subqueries**

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

**EXAMPLE:** SELECT a, b FROM t1 WHERE EXISTS (SELECT * FROM t2 WHERE t1.a > t2.a)

**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 E1, ..., Ek (which are subqueries or relations).

2. Form all possible combinations of tuples V1, ..., Vk, where V1 comes from E1, etc:
   (a) For each combination determine if <condition> is true.
       This may involve computing subqueries. If (attributes from) one of V1, ..., Vk 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 V1, ..., Vk.
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.
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. (Coming next.)
Set operators in SQL

SQL offers a number of set operators that can be used to combine results of several queries, if the queries return relations with compatible schemas.

- $R_1 \text{ UNION } R_2$ gives the relation with all tuples of $R_1$ and $R_2$.
- $R_1 \text{ INTERSECT } R_2$ gives the relation with all tuples that can be found in both $R_1$ and $R_2$.
- $R_1 \text{ EXCEPT } R_2$ gives the relation with all tuples that can be found in both $R_1$ but not in $R_2$.

For all operators, the result relation will have no duplicate tuples.
Relations in SQL are **bags** (or **multisets**), i.e., tuples may appear more than once.

The reason why bags are used is that *duplicate elimination* is relatively costly (requires time and memory), so it is generally an advantage to use it only when necessary.

We can force duplicate elimination in a SELECT-FROM-WHERE by adding the keyword DISTINCT.

The UNION operator involves duplicate elimination. To avoid duplicate elimination use $R_1 \text{ UNION ALL } R_2$, which corresponds to “putting one relation on top of the other”.
Next: Security and authorization in SQL.
Database security issues

- Data loss and unavailability – DBMSs provide good protection.
- Theft, fraud, sabotage – We won’t address this issue.
- Data integrity – database constraints and triggers may aid in this direction (cf. also lecture on transactions).
- Loss of privacy/confidentiality – SQL has an authorization mechanism.
Authorization in SQL

Because databases often have many users, not all of which are allowed to do any database operation, SQL has an authorization system.

Every user (called a module in case the user is a program) has certain rights, or privileges, to access and modify database elements. The basic privileges are:

   SELECT, INSERT, DELETE, UPDATE

It is possible to have privileges for certain attributes in a relation, e.g., a secretary might have the UPDATE(address, city) privilege for relation with customer information.
Basics of managing privileges:

- If a user defines a new schema, she has all possible privileges for the tables (and other database elements) of that schema.
- Users may grant ("copy") privileges of their own to other users.
- Being able to grant a privilege is a special privilege in itself that can be passed on.

Syntax for granting privileges:

```
GRANT <privilege list> ON <database element>
TO <user list> [WITH GRANT OPTION]
```
Revoking privileges

Granted privileges can be withdrawn (or revoked) by a user at any time.

Basics of revoking privileges:

- Privileges given without the GRANT OPTION can simply be removed.
- Otherwise we would like to revoke any privilege in the database that the revoked user has granted unless the privilege also are granted by another user that still has the GRANT privilege.
- What happens when revoking is independent on the order in which privileges were given.

Syntax for revoking privileges:

```
REVOKE <privilege list> ON <database element>
FROM <user list> CASCADE
```
Next: Object orientation in databases and database modeling.
Object orientation

Object orientation is a way of thinking about data and behavior.

Object oriented programming languages (e.g. Java) support object oriented concepts such as:

- Classes, objects
- Inheritance
- Methods
- Polymorphism

UML is an increasingly popular notation that can be used for object-oriented design and analysis, also for database systems.
Enhanced E-R model vs UML

Similar features:

- Entity type ≈ class.
- Entity instance ≈ object (instance).
- Relationship ≈ association.
- Associative entity ≈ association class.
- Super/subtypes ≈ inheritance

Some differences/additional features of UML:

- Implicit primary key in objects.
- Class-scope attributes.
- Notation for associating actions/behavior to objects.
- Notation for showing example object instances.
Object-oriented and object-relational databases

Around 15 years ago many people believed that object-oriented databases and query languages would take over the world.

Several data models and query languages have been proposed (see MDM ch. 15), but have found limited use.

Instead, backwards-compatible object-relational databases dominate the market, integrating essential parts of object-oriented languages into SQL:

- Ability to store code with data.
- Ability to define new, complex data types.

It seems likely that object-relational hybrids will continue to dominate.
The major DBMSs, and the SQL-99 standard, give a way of including functions and procedures in the database schema.

This means that tuples can be thought of as objects with methods.

It is not clear to what extent this will replace embedded and dynamic SQL in systems communicating with DBMSs.
Next: Illustrating Object-Relational DBMS by Oracle
(The important thing to grasp is not the notation but the OO-concepts)
What will be described

- Defining complex data types
- Making methods in the objects
- Making nested tables

The Oracle Syntax are only used as an example of the workings of a Object-Relational DBMS. The syntax and the capabilities of the DBMS in the Object-Oriented domain differs a lot from system to system.
CREATE TYPE CourseType AS OBJECT (  
coursename VARCHAR(20),
teacher VARCHAR(20),
startdate DATE,
enddate DATE,
);
/
/
CREATE TABLE CourseTable (  
id NUMBER,
course CourseType
);

<table>
<thead>
<tr>
<th>id</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Complex Data types (Records) 2 of 2

```sql
INSERT INTO Course VALUES (
    1,
    CourseType('DBS', 'Esben', '02feb06', '27apr06')
);

INSERT INTO Course VALUES (
    2,
    CourseType('ADBT', 'Rasmus', '30jan06', '24apr06')
);

SELECT c.course.coursename, c.course.teacher
FROM CourseTable c;
```

<table>
<thead>
<tr>
<th>id</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DBS</td>
</tr>
<tr>
<td></td>
<td>teacher: Esben</td>
</tr>
<tr>
<td></td>
<td>startdate: 02feb06</td>
</tr>
<tr>
<td></td>
<td>enddate: 27apr06</td>
</tr>
<tr>
<td>2</td>
<td>ADBT</td>
</tr>
<tr>
<td></td>
<td>teacher: Rasmus</td>
</tr>
<tr>
<td></td>
<td>startdate: 30jan06</td>
</tr>
<tr>
<td></td>
<td>enddate: 24apr06</td>
</tr>
</tbody>
</table>

**C.COURSE.COURSENAME**  **C.COURSE.TEACHER**

- DBS  Esben
- ADBT Rasmus
CREATE TYPE CourseType AS OBJECT (  
coursename VARCHAR(20),
teacher VARCHAR(20),
startdate DATE,
enddate DATE,
MEMBER FUNCTION weeks(vacation NUMBER) RETURN NUMBER
);
/
CREATE TYPE BODY CourseType AS

MEMBER FUNCTION weeks(vacation NUMBER) RETURN NUMBER IS
  BEGIN
    RETURN floor((enddate-startdate)/7)+1-vacation;
  END;
END;
/
Creating Functions in Oracle 2 of 2

```sql
SELECT c.course.coursename, c.course.weeks(1)
FROM CourseTable c;
```

<table>
<thead>
<tr>
<th>COURSE.COURSENAM</th>
<th>C.COURSE.WEEKS(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBS</td>
<td>12</td>
</tr>
<tr>
<td>ADBT</td>
<td>12</td>
</tr>
</tbody>
</table>
CREATE TYPE PartType AS OBJECT (  
  name VARCHAR(20),  
  email VARCHAR(20)  
);  
/
/
CREATE TYPE ParticipantsType AS TABLE OF PartType;  
/
/
CREATE TYPE NewCourseType AS OBJECT (  
  coursename VARCHAR(20),  
  teacher VARCHAR(20),  
  participants ParticipantsType  
);  
/
/
CREATE TABLE NewCourseTable (  
  id NUMBER,  
  course NewCourseType)  
  NESTED TABLE course.participants STORE AS ParticipantsTable;
### NewCourseTable:

<table>
<thead>
<tr>
<th>id</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### course:
- **coursename:**
- **teacher:**
  - name:
  - email:
- **participants:**
  - name:
  - email:

#### Nested Tables 2 of 2
As a minimum, you should after this week:

- Be able to understand and form SQL expressions using several levels of subqueries.
- Tell the difference between correlated and uncorrelated subqueries.
- Understand the mechanism for granting and revoking privileges in SQL.
- Have knowledge of the basic tools for implementing OO concepts in object-relational DBMSs.