CSCI 403: Databases
6 - Subqueries

Subqueries

Also known as *nested queries*, subqueries allow you to use the results from nested SELECT queries in the SELECT, FROM, and WHERE clauses of another SELECT query, in the WHERE clause of a DELETE query, and in the WHERE and SET clauses of an UPDATE query.

Evaluating a Subquery

Recall that a SELECT query returns a relation, i.e., a set of tuples. In the case of a subquery we need to separate the possible results into those with no tuples, those with exactly one tuple, and those with multiple tuples. Some uses of subqueries require zero or one result, whereas others can handle multiple results.

Another way of looking at this is that a subquery may result in:

- A relation
- A tuple
- A (scalar) value
- NULL

Subqueries in WHERE

Most applications of subqueries tend to be in the WHERE clause of some outer query. Note that WHERE clauses occur in SELECT, DELETE, and UPDATE queries, and thus we can use subqueries in all of those cases.

IN and NOT IN

If we view the result of our subquery as being a relation (e.g., a set of tuples of any size), then the primary use of the subquery is with the operator IN. IN tests the tuples of a subquery result for a value match with some attribute or expression in the main query, resulting in a Boolean TRUE if a match is found, and FALSE otherwise. The expression being tested can be a scalar or a tuple, and the rows in the subquery result must match the expression in degree and type.

Example:

```
SELECT course_id FROM mines_courses
WHERE instructor IN
(SELECT name FROM mines_eecs_faculty);
```

The subquery is given in parentheses, and in this instance, returns rows with a single attribute. If we wish to compare tuple values instead, the query might look like (assuming we had separate last and first names in our tables):

```
SELECT course_id FROM mines_courses
WHERE (instr_first, instr_last) IN
(SELECT first, last FROM
mines_eecs_faculty);
```

Here we must enclose the attributes being compared in parentheses to mark them off as a tuple.

Note that the above query can equivalently be expressed using a join:

```
SELECT course_id
FROM mines_courses AS mc,
mines_eecs_faculty AS mef
WHERE mef.name = mc.instructor;,
```

although it is equivalent in this case only because each choice of instructor can match at most one row in mines_eecs_faculty (otherwise the two queries would produce the same values, but in different multiplicities). Adding DISTINCT to both queries would ensure their equivalence, and in general, subqueries in the WHERE clause can often be rewritten using joins.

If we change the query to use NOT IN instead of IN, we select the set of rows where the expression does *not* match any results from the subquery. (This can be equivalently expressed using

an OUTER JOIN and an IS NULL test (see next lecture for outer joins), but the NOT IN query may be easier to write and understand.)

Comparison with Single Tuple or Scalar Result

If a subquery can be guaranteed to return zero or one rows, then a number of additional options become available. In particular, the result of such a query can be used with comparison operators, particularly $=$ and $<>$. For example, to obtain faculty information about EECS faculty who do not teach CSCI 403, we can do:

```
SELECT * FROM mines_eecs_faculty
WHERE name <>
(SELECT instructor FROM mines_courses
WHERE course_id = 'CSCI403');
```

When the returned value is scalar and comparable, we can also use comparison operators such as $<$, $<=$, $>$, and $>=$.

Note that our query will fail if the subquery returns more than one row; the SQL engine will report an error. In the above query, we can produce the error by replacing 'CSCI403' with 'CSCI262', which has two rows in the mines_courses table.

When the subquery returns 0 rows, then the return values are interpreted as NULL, and thus will cause any comparison to be false.

Correlated Subqueries

In a correlated subquery, rows in the nested query are selected using some value from the outer query in the nested query's WHERE clause. This results in a correlation between the inner query and the tuples selected in the outer query. For example:

```
SELECT DISTINCT instructor, course_id
FROM mines_courses AS mc1
WHERE course_id IN
 (SELECT course_id
 FROM mines_courses AS mc2
 WHERE mc2.course_id = mc1.course_id
 AND mc2.instructor <>
mc1.instructor);
```

To understand what this query is doing, we must first understand how the SQL engine processes a correlated subquery (at least conceptually). Conceptually, you can think of the correlated subquery being executed once for each row of the outer query. So we iterate over every row in mines_courses and see if the course_id is in the result set from the nested query. Inside the nested query, we have access to the values of attributes in the outer query for the current row. In this case, we look for rows in mines_courses where the course_id matches the course_id in the currently iterated row, but where the instructor does not match the instructor in the currently iterated row. That is, we only take rows in the outer query when there exists a row in mines_courses which matches in course_id but not in instructor. In effect, this query is finding out the instructors and courses where different instructors teach different sections.

Once again, this query could be expressed using a join of mines_courses to itself. Which you use may depend on a variety of factors; which is most clear to understand, which one is easier to express in application software, etc.

EXISTS and UNIQUE

In the above query, we didn't really need to check to see if course_id was in the result set for the correlated subquery; it would suffice for there to be any result whatsoever (1 or more rows). The EXISTS operator tests a subquery for a non-zero number of rows in the result (NOT EXISTS tests for zero rows). The above query could be rewritten using EXISTS as:

```
SELECT DISTINCT instructor, course_id
FROM mines_courses AS mc1
WHERE EXISTS
 (SELECT course_id
 FROM mines_courses AS mc2
 WHERE mc2.course_id = mc1.course_id
 AND mc2.instructor <>
mc1.instructor);
```

Yet another operator that can be applied to a subquery is UNIQUE. This operator simply tests to see if the result tuples from a query are all unique (e.g., that the result is a set rather than a multiset).

Subqueries in FROM

Since the result of a SELECT query is simply a relation, we can treat a subquery as if it were an unnamed table, and select from it. For a trivial

example,
SELECT course_id FROM
(SELECT course_id, instructor
FROM mines_courses) AS mc
WHERE mc.instructor LIKE 'Painter%';

Subqueries in SELECT and SET

You can use a subquery returning a single (scalar) value in other situations where a scalar expression is appropriate. For instance, you can use a subquery to obtain a value inside the SELECT clause of a query, for instance:

```
SELECT instructor, course_id,
 (SELECT office
 FROM mines_eecs_faculty AS mef
 WHERE mef.name = instructor) AS
office
FROM mines_courses;
```

It is also occasionally useful to use a subquery to retrieve a value that you want to set some column to in an update query (typically using a correlated subquery):

```
UPDATE table1 AS T1
SET attribute =
 (SELECT val FROM table2 AS T2
 WHERE condition on attributes of T1,
T2)
WHERE condition on T1;
```

Note that the above is uniquely useful in that it acts kind of like a join inside an UPDATE query.