Databases - SQL 4

Gordon Royle

School of Mathematics & Statistics
University of Western Australia

This lecture

We continue our coverage of the fundamentals of SQL/MySQL with *nested queries*, also known as *subqueries*.

Nested Queries

A *nested query* is a query that involves *another query* as one of its component parts.

```
SELECT * FROM Score
WHERE event_id =
    (SELECT event_id FROM GradeEvent
     WHERE date = ’2014-09-09’);
```

Here we have a simple query that involves *two* `SELECT` statements.

Analysis

The *inner query* produces just the `event_id` of the test/quiz on 9th September 2014.

```
SELECT event_id
FROM   GradeEvent
WHERE  date = '2014-09-09';
+----------+
| event_id |
+----------+
|        3 |
+----------+
```

The *outer query* is then equivalent to

```
SELECT *
FROM   Score
WHERE  event_id = 3;
```

Types of subquery

How a subquery can be manipulated depends on the type of results that it produces:

- A *scalar* subquery produces a *single value* (that is, a table with one row and one column) as a result
- A *column* subquery produces a single column as a result
- A *row* subquery produces a single row as a result
- A *table* subquery produces an entire table as a result

There are special operators that can be used with each of these types of query.

Scalar subqueries

The result of a scalar subquery can be used essentially anywhere that a single value can be used, e.g. you can make comparisons with <, >, =, <> and so on.

Sometimes a scalar subquery is just used to find an unknown value from another table:

```
SELECT *
FROM City
WHERE CountryCode = (SELECT Code
                     FROM Country
                     WHERE name = 'Australia');
```

Equivalent to a join

A subquery like this equivalent to *a join*.

```
SELECT T.* FROM City T, Country C
WHERE T.CountryCode = C.code
AND C.name = 'Australia';
```

```
+-----+---------------+-------------+-----------------+------------+
| ID  | Name          | CountryCode | District        | Population |
+-----+---------------+-------------+-----------------+------------+
| 130 | Sydney        | AUS         | New South Wales |    3276207 |
| 131 | Melbourne     | AUS         | Victoria        |    2865329 |
| 132 | Brisbane      | AUS         | Queensland      |    1291117 |
| 133 | Perth         | AUS         | West Australia  |    1096829 |
| 134 | Adelaide      | AUS         | South Australia |     978100 |
```

Notice the use of `T.*` to get all of the fields from just the `City` part of the joined table.

Maximum population

An example of a subquery that cannot be replaced by a simple join is when the selection is based on the *result* of an *aggregate* operation.

```
SELECT name, population
FROM Country
WHERE population = (SELECT MAX(population)
                    FROM Country);
```

```
+-------+------------+
| name  | population |
+-------+------------+
| China | 1277558000 |
+-------+------------+
```

Analysis

This works as follows:

- The inner query uses the summary function `MAX` which can only produce a value after *every row* in the table has been scanned.
- The outer query then causes the table to be *re-scanned* to locate which actual row had that particular value.

We cannot do this in one operation — though here you may see an *imperative procedure* that could do better than using two scans

User Variables

You can also do such a query in *two steps* if you wish, because MySQL allows the user to define *user variables*. A user variable must begin with the `@` character and can be created within a `SELECT` statement.

```
SELECT @maxpop := MAX(population)
FROM Country;

SELECT name, population
FROM Country
WHERE population = @maxpop;
```

The first command creates a *variable* called `@maxpop` and assigns a value to it, while the second command *uses* that variable.

Relative comparisons

Which countries are between Germany and Indonesia according to population?

```
SELECT name, population FROM
Country
WHERE population <= (SELECT population
                     FROM Country
                     WHERE name = ’Indonesia’)
AND population >= (SELECT population
                   FROM Country
                   WHERE name = ’Germany’)
ORDER BY population DESC;
```

Which countries?

```
+--------------------+------------+
| name               | population |
+--------------------+------------+
| Indonesia          |  212107000 |
| Brazil             |  170115000 |
| Pakistan           |  156483000 |
| Russian Federation |  146934000 |
| Bangladesh         |  129155000 |
| Japan              |  126714000 |
| Nigeria            |  111506000 |
| Mexico             |   98881000 |
| Germany            |   82164700 |
+--------------------+------------+
```

Which countries have above average population density?

```
SELECT name, population/surfacearea AS density
FROM Country
WHERE population/surfacearea >
     (SELECT AVG(population/surfacearea) FROM Country)
ORDER BY density DESC;
```

```
+-------------------------------+--------------+
| name                          | density      |
+-------------------------------+--------------+
| Macao                         | 26277.777778 |
| Monaco                        | 22666.666667 |
| Hong Kong                     |  6308.837209 |
| Singapore                     |  5771.844660 |
| Gibraltar                     |  4166.666667 |
...
```

IN and NOT IN

If a subquery returns more than one value, then it can be treated as a *set of values* and the outer query can test whether values are IN or NOT IN this set.

For example, we can find out which sailors in the Sailor table have *not* reserved any boats.

```
SELECT * FROM Sailor
WHERE sid NOT IN (SELECT sid
                  FROM Reserves);
+-----+--------+------+
| sid | sname  | age  |
+-----+--------+------+
|  29 | Brutus |   33 |
|  32 | Andy   | 25.5 |
|  58 | Rusty  |   35 |
...
```

Analysis I

The *inner query* is

```
SELECT sid
FROM Reserves;
+-----+
| sid |
+-----+
|  22 |
|  22 |
|  22 |
|  22 |
|  31 |
...
```

which is a single-column table containing the ids of sailors who *have* reserved boats.

Analysis 2

The *outer query* then asks for any ids that *are not in* the set of ids produced by the inner query. It is equivalent to

```
SELECT *
FROM   Sailor
WHERE  sid NOT IN ( 22, 31, 64, 74 );
```

Further examples

Which students are not enrolled in any classes?

```
SELECT S.sname
FROM Student S
WHERE S.snum NOT IN
     (SELECT snum FROM Enrolled);
+-----------------+
| sname           |
+-----------------+
| Maria White     |
| Charles Harris  |
| Angela Martinez |
...
```

This uses the same idea as the previous example.

Most populous country in each region

Suppose we want to find the most heavily-populated country in each of the world's regions. We know how to find the *maximum population* easily enough.

```
SELECT C.region,
       Max(C.population) AS maxpop
FROM   Country C
GROUP  BY region;
+---------------------------+------------+
| region                    | maxpop     |
+---------------------------+------------+
| Antarctica                |          0 |
| Australia and New Zealand |   18886000 |
| Baltic Countries          |    3698500 |
| British Islands           |   59623400 |
```

This tells us that, for example, that the biggest country in the Baltic Countries has a population of 3698500, but not *which country*

Incorrect approach

An obvious, but unfortunately incorrect, approach would be to try

```
SELECT C.region,
       C.name,
       Max(C.population) AS maxpop
FROM   Country C
GROUP  BY C.region
+------------------+----------------------------+------------+
| region           | name                       | maxpop     |
+------------------+----------------------------+------------+
| Antarctica       | Antarctica                 |          0 |
| Australia and NZ | Australia                  |   18886000 |
| Baltic Countries | Latvia                     |    3698500 |
| British Islands  | United Kingdom             |   59623400 |
| Caribbean        | Netherlands Antilles       |   11201000
```

Why is this incorrect?

This is such a common error that it is *very important* to understand why it is not correct.

The issue is that

- The `region` field is one of the `GROUP BY` fields and so has the same value for all the rows in each group
- The `name` field is *not* one of the `GROUP BY` fields and so the rows in each group can have different values for this field.

Why is this incorrect?

So after the groups have been formed (internally, by MySQL) the group for the Baltic countries looks like this:

```
| Baltic Countries   | Latvia         |    2424200 |
| Baltic Countries   | Estonia        |    1439200 |
| Baltic Countries   | Lithuania      |    3698500 |
```

The presence of the summary function `MAX` indicates that each group should be summarised into a single row containing a `region`, a `name` and a `MAX` value.

Correct Approach 1

One correct approach would be to use an *inner query* that first determines the maximum population for each region, and then an *outer query* that “attaches” the correct country name to that pair.

```
SELECT C.region,
       C.name,
       C.population
FROM   Country C
WHERE  ( C.region, C.population ) IN (SELECT C2.region,
                                             MAX(C2.population)
                                      FROM   Country C2
                                      GROUP  BY region);
```

Correct Approach 2

The second correct approach uses a *correlated subquery* which is where the subquery refers to a table from the *outer query*.

```
SELECT C.region,
       C.name,
       C.population
FROM   Country C
WHERE  C.population = (SELECT MAX(population)
                       FROM   Country C2
                       WHERE  C2.region = C.region);
```

This subquery is called *correlated* because it involves a value (`C.region`) that comes from a table in the outer query.

Visualizing correlated subqueries

Conceptually, we imagine a correlated subquery as being run once for each row of the table that it refers to.

For the query on the previous slide, we imagine `C` being set equal to each row of the table `Country` in turn:

```
| Afghanistan         | Southern and Central Asia |22720000 |
| Netherlands         | Western Europe            |15864000 |
| Netherlands Antilles| Caribbean                 |  217000 |
...
```

Then each time through, the maximum population of the region `C.region` is computed and compared to the actual population of `C`.

Example schema

We will use a schema regarding industrial parts (spanners, wrenches etc), suppliers of those parts, and a catalogue that indicates who is supplying which part at what price.

```
CREATE TABLE Suppliers (
sid INT PRIMARY KEY,
sname VARCHAR(64),
address VARCHAR(512) );

CREATE TABLE Parts (
pid INT PRIMARY KEY,
pname VARCHAR(64),
colour VARCHAR(16) );
```

The catalogue

```
CREATE TABLE Catalogue (
     sid   INT,
     pid   INT,
     price DECIMAL(10, 2));
```

So `Catalogue` is a *relationship* between parts and suppliers.

(Probably it would be better named `Supplies` to stick to the *entity-noun*, *relationship-verb* model.)

EXISTS and NOT EXISTS

The clauses EXISTS and NOT EXISTS can be used in conjunction with a subquery simply to see if that subquery returns *any results*.

This kind of construct can be useful when answering *all or none* questions in relational tables. For example, consider the question:

Which suppliers do not supply any parts?

```
SELECT S.sname
FROM   Suppliers S
WHERE  NOT EXISTS (SELECT *
                   FROM   Catalogue C
                   WHERE  S.sid = C.sid);
```

Who supplies *every* part

To find out who supplies *every* part in the catalogue requires a bit of linguistic contortion.

First let's find out which parts a supplier with id `sid` does *not* supply — notice that this is not a fully-formed query because `sid` is not qualified.

```
SELECT P.pid
FROM   Parts P
WHERE  NOT EXISTS (SELECT *
                   FROM   Catalogue C
                   WHERE  C.pid = P.pid
                          AND C.sid = sid);
```

Double negative

Now a supplier supplies *every* part if we *cannot find* a part that the supplier *does not supply*.

```
SELECT S.sname
FROM   Suppliers S
WHERE  NOT EXISTS (SELECT P.pid
                   FROM   Parts P
                   WHERE  NOT EXISTS (SELECT *
                                      FROM   Catalogue C
                                      WHERE  C.pid = P.pid
                                             AND C.sid = S.sid));
```