This lecture

We continue our coverage of the fundamentals of SQL/MySQL with nested queries, or 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 grade_event
   WHERE date = '2004-09-09');
```

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 2004.

```
SELECT event_id FROM grade_event
   WHERE date = '2004-09-09';
```

| +----------+ |
| event_id  |
| +----------+ |
| 3         |
```

Thus the outer query is simply 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

A scalar subquery produces a one-row, one-column table that is treated as a single value (that is, a *scalar*), and can be used essentially anywhere that a single value can be used – for example, you can make comparisons with $<$, $>$, $=$, $<$>, and so on.

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

```
SELECT * FROM city
WHERE countrycode = (SELECT code
FROM country
WHERE name = 'Australia');
```

Equivalent to a join

A subquery like this is just an alternative to a join.

```
SELECT T.* FROM city T, country C
WHERE T.countrycode = C.code
AND C.name = 'Australia';
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>CountryCode</th>
<th>District</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
<td>Sydney</td>
<td>AUS</td>
<td>New South Wales</td>
<td>3276207</td>
</tr>
<tr>
<td>131</td>
<td>Melbourne</td>
<td>AUS</td>
<td>Victoria</td>
<td>2865329</td>
</tr>
<tr>
<td>132</td>
<td>Brisbane</td>
<td>AUS</td>
<td>Queensland</td>
<td>1291117</td>
</tr>
<tr>
<td>133</td>
<td>Perth</td>
<td>AUS</td>
<td>West Australia</td>
<td>1096829</td>
</tr>
<tr>
<td>134</td>
<td>Adelaide</td>
<td>AUS</td>
<td>South Australia</td>
<td>978100</td>
</tr>
</tbody>
</table>

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

Maximum population

More significant uses that cannot be replaced by simple joins occur when a selection is based on the result of an *aggregate operation*.

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

<table>
<thead>
<tr>
<th>name</th>
<th>population</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>1277558000</td>
</tr>
</tbody>
</table>
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.

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

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

This uses a variable @maxpop to store the maximum population value to be used in the second query.

Relative comparisons

Which countries are between Germany and Indonesia according to population?

```sql
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 have above average population density?

```sql
SELECT name, population/surfacearea as density
FROM country
WHERE population/surfacearea > (SELECT AVG(population/surfacearea) FROM country)
ORDER BY density DESC;
```

<table>
<thead>
<tr>
<th>name</th>
<th>density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macao</td>
<td>26277.777778</td>
</tr>
<tr>
<td>Monaco</td>
<td>22666.666667</td>
</tr>
<tr>
<td>Hong Kong</td>
<td>6308.837209</td>
</tr>
<tr>
<td>Singapore</td>
<td>5771.844660</td>
</tr>
<tr>
<td>Gibraltar</td>
<td>4166.666667</td>
</tr>
</tbody>
</table>

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.

```sql
SELECT * FROM sailor
WHERE sid NOT IN (SELECT sid FROM reserves);
```

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>Brutus</td>
<td>33</td>
</tr>
<tr>
<td>32</td>
<td>Andy</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>35</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Further examples

Which students are not enrolled in any classes?

```sql
SELECT S.sname
FROM student S
WHERE S.snum NOT IN
  (SELECT snum FROM enrolled);
```

<table>
<thead>
<tr>
<th>sname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maria White</td>
</tr>
<tr>
<td>Charles Harris</td>
</tr>
<tr>
<td>Angela Martinez</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

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 population easily enough.

```sql
SELECT C.region, MAX(C.population) as maxpop
FROM country C
GROUP BY region;
```

<table>
<thead>
<tr>
<th>region</th>
<th>maxpop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antarctica</td>
<td>0</td>
</tr>
<tr>
<td>Australia and New Zealand</td>
<td>18886000</td>
</tr>
<tr>
<td>Baltic Countries</td>
<td>3698500</td>
</tr>
<tr>
<td>British Islands</td>
<td>59623400</td>
</tr>
<tr>
<td>Caribbean</td>
<td>11201000</td>
</tr>
</tbody>
</table>

This tells us that, for example, that some country in the Baltic Countries has a population of 3.7 million, but does not tell us which country.

---

### Incorrect approach

An obvious, but unfortunately incorrect approach would be to try

```sql
SELECT C.region, C.name, MAX(C.population) as maxpop
FROM country C
GROUP BY region;
```

<table>
<thead>
<tr>
<th>region</th>
<th>name</th>
<th>maxpop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baltics</td>
<td>Latvia</td>
<td>2424200</td>
</tr>
<tr>
<td>Baltics</td>
<td>Estonia</td>
<td>1439200</td>
</tr>
<tr>
<td>Baltics</td>
<td>Lithuania</td>
<td>3698500</td>
</tr>
<tr>
<td>Caribbean</td>
<td>Netherlands Antilles</td>
<td>11201000</td>
</tr>
</tbody>
</table>

This is incorrect because after the “grouping by region”, the group for the Baltic countries looks like this:

<table>
<thead>
<tr>
<th>region</th>
<th>name</th>
<th>maxpop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baltic Countries</td>
<td>Latvia</td>
<td>2424200</td>
</tr>
<tr>
<td>Baltic Countries</td>
<td>Estonia</td>
<td>1439200</td>
</tr>
<tr>
<td>Baltic Countries</td>
<td>Lithuania</td>
<td>3698500</td>
</tr>
</tbody>
</table>

The presence of the aggregate function `MAX` indicates that each group should be aggregated into a single row.

There is no ambiguity in the value of `C.region` because it is constant over the group, and nor is there any ambiguity in the value `MAX(population)`. However the value `name` is an individual value, and not a property of the group. Hence SQL simply takes the first value — possibly a warning should be issued if a column is named in the `SELECT` but not used in the `GROUP BY` clause.
Correct Approach 1

One correct approach would be to use a subquery 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);
```

The inner is run once and produces a list of (region, maxpop) pairs, while the outer query essentially checks the values for each country to see if they match a row in this table.

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 this schema from the textbook:

- Suppliers (sid: integer, sname: string, address: string)
- Parts (pid: integer, pname: string, colour: string)
- Catalogue (sid: integer, pid: integer, price: real)

Here, suppliers and parts are entity sets that are related by the relationship set (which has the name catalogue) indicating which suppliers sell which parts.

The price variable is a relationship attribute of this relationship set, because the price is an attribute of the combination of a particular seller and a particular part.
**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);
```

(GF Royle 2006-8, N Spadaccini 2008) SQL III 21 / 24

**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);
```

(GF Royle 2006-8, N Spadaccini 2008) SQL III 22 / 24

**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 S.sid = C.sid));
```

(GF Royle 2006-8, N Spadaccini 2008) SQL III 23 / 24

**Unique suppliers**

*Find the names of the parts supplied only by Acme Widget Suppliers.*

By now the general principle is becoming apparent — find the parts that are supplied by Acme Widget Suppliers, but for which there are no **different** suppliers.

```
SELECT P.pname
FROM parts P, catalogue C, suppliers S
WHERE P.pid = C.pid
AND C.sid = S.sid
AND S.sname = 'Acme Widget Suppliers'
AND NOT EXISTS (SELECT *
    FROM catalogue C2
    WHERE C2.sid <> S.sid
    AND C2.pid = P.pid);
```

This correlated subquery uses two of the “outer” tables.

(GF Royle 2006-8, N Spadaccini 2008) SQL III 24 / 24