Databases - SELECT

Gordon Royle

School of Mathematics & Statistics
University of Western Australia
The \textbf{SELECT} statement

This lecture reviews what we know about the \texttt{SELECT} statement, which is the cornerstone of database use.
The structure

This is the template for a SELECT statement.

```
SELECT columns (1)
FROM tables (2)
WHERE conditions (3)
GROUP BY group columns (4)
HAVING more conditions (5)
ORDER BY sort columns (6)
LIMIT number (7)
```
Conceptual execution plan

- SQL builds a “master-table” by joining the tables specified in the `FROM` clause (2)
- SQL processes each row, keeping only the rows that satisfy the `WHERE` clause (3)
- SQL forms the rows into groups according to the `GROUP BY` clause (4)
- SQL takes each group in turn, and produces one `summary row` per group, by choosing either `named` or `calculated columns` according to the `SELECT` clause (1)
- SQL processes each summary row, keeping the rows that satisfy the `HAVING` clause (5)
- The rows that have passed every test so far are then `sorted` according to the values specified in the `ORDER BY` clause
- This table is output, limited to the number of rows specified in the `LIMIT` clause
Columns (1) is a comma-separated list of items, each of which will contribute one column to the output table.

Each item is either

- A column name
  
  SELECT employeeNumber FROM...
  SELECT E.employeeNumber FROM...
  SELECT Employees.employeeNumber FROM...

- An expression involving column names
  
  SELECT unitPrice * quantity FROM...
  SELECT CONCAT(firstName, lastName) FROM...
- An *aggregate* function (usually with `GROUP BY`)
  
  `SELECT name, COUNT(*) FROM...`
  `SELECT name, MIN(mark) FROM...`

- A *value* that can be immediately evaluated
  
  `SELECT 2+3;`
  `SELECT SIN(1);`
  `SELECT POW(2,4);`

- Any of the above, *renamed*
  
  `SELECT unitPrice * quantity AS orderPrice FROM...`
The **FROM** clause defines a table — conceptually, this is the “master table” from which *everything else is calculated*. This clause can be

- The name of an actual table
  
  ```
  ...FROM employees...
  ...
  ```

- A **JOIN** of two or more actual tables
  
  ```
  ...FROM Student, Enrolled ...
  ...
  ```
A **derived table**

...FROM

(SELECT * FROM Store
    WHERE postCode = 6009) AS localStore ...

All derived table must be given an *alias*, even if it is never used.

This definition is *recursive* in that a derived table may itself use another derived table (and so on).
The WHERE clause is a *boolean expression* (that is, a true/false expression) that is applied to every row of the “master-table” in turn. Only the rows for which the expression is true are kept.

The WHERE clause can be

- A test for equality
  
  ```
  ...WHERE employeeNumber = 1002...
  ...WHERE gender = 'M'...
  ...WHERE DAYOFWEEK(salesDate) = 0...
  ```

- A test for inequality

  ```
  ...WHERE employeeNumber <> 1002...
  ...WHERE countryCode <> 'GBR'...
  ```
WHERE (3) - cont

- A comparison

  ...WHERE csMark < mathMark...
  ...WHERE YEAR(dateOfBirth) < 1995...

- A compound boolean expression

  ... WHERE csMark < mathMark
  AND csMark > 50...

  ... WHERE NOT (csMark < 50 OR mathMark < 50)...

The operators are

- AND, & & Logical AND
- OR, || Logical OR
- NOT, ! Logical Negation
- XOR Logical exclusive-OR
WHERE (3) - cont

- Membership or non-membership

  ...WHERE id IN (1,5,8,12)...
  ...WHERE id NOT IN (SELECT id FROM... )

- Existence or non-existence\(^1\)
  (This can only be illustrated with a complete statement)

  ```sql
  FROM Country C
  WHERE NOT EXISTS (SELECT *
                  FROM Country C1
                  WHERE C1.population > C.population);
  ```

\(^1\) We have not covered this
The **GROUP BY** statement forms the surviving rows into groups in such a way the rows in each group have the same value on *all of* the named columns.

- One or more columns
  
  ```sql
  ...GROUP BY gender...
  ...GROUP BY region...
  ...GROUP BY unitCode, gender...
  ```
Some data from Country grouped by region

<table>
<thead>
<tr>
<th>Country</th>
<th>Population</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Australia</td>
<td>18886000</td>
<td>Australia and New Zealand</td>
</tr>
<tr>
<td>Cocos (Keeling) Islands</td>
<td>600</td>
<td>Australia and New Zealand</td>
</tr>
<tr>
<td>Christmas Island</td>
<td>2500</td>
<td>Australia and New Zealand</td>
</tr>
<tr>
<td>Norfolk Island</td>
<td>2000</td>
<td>Australia and New Zealand</td>
</tr>
<tr>
<td>New Zealand</td>
<td>3862000</td>
<td>Australia and New Zealand</td>
</tr>
<tr>
<td>Latvia</td>
<td>2424200</td>
<td>Baltic Countries</td>
</tr>
<tr>
<td>Lithuania</td>
<td>3698500</td>
<td>Baltic Countries</td>
</tr>
<tr>
<td>Estonia</td>
<td>1439200</td>
<td>Baltic Countries</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>59623400</td>
<td>British Islands</td>
</tr>
<tr>
<td>Ireland</td>
<td>3775100</td>
<td>British Islands</td>
</tr>
</tbody>
</table>

GROUP BY - cont
The expressions in the `SELECT` statement are then evaluated over each group, producing *one summary row* per group.

```
SELECT region, SUM(POPULATION) FROM...
```

SQL then takes from each group:
- The value for `region` from the first row
- The sum of the `population` values from each group
The **HAVING** clause is another round of selection, but this time on the summary rows produced by the previous step.

The additional conditions are based on the column names as determined by the SELECT statement.

```
SELECT region, 
    SUM(population)
FROM Country
GROUP BY region
HAVING SUM(population) > 100000000;
```

<table>
<thead>
<tr>
<th>region</th>
<th>sum(population)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central America</td>
<td>135221000</td>
</tr>
<tr>
<td>Eastern Africa</td>
<td>246999000</td>
</tr>
<tr>
<td>Eastern Asia</td>
<td>1507328000</td>
</tr>
</tbody>
</table>
ORDER BY

The final (optional) step is to sort the rows into a sensible order if desired.

Unless instructed to do so, MySQL will not sort the rows into any particular order — this is because sorting is a computationally expensive operation.

```sql
SELECT region, 
    Sum(population)
FROM country
GROUP BY region
HAVING Sum(population) > 100000000
ORDER BY Sum(population) DESC;
```

ORDER BY sorts the data in ascending order (i.e. from low to high) according to the values in the specified column. Specifying DESC reverses the order so that the rows are sorted in descending order.