Databases — SQL2

Gordon Royle

School of Mathematics & Statistics
University of Western Australia

This lecture

This lecture introduces the fundamental concept of

- `SELECT` from multiple tables

In order to select from multiple tables, the tables must be *joined* — so this lecture is also about the various types of `JOIN`.

Multiple table selections

The real power (and complexity) of `SELECT` comes from the ability to rapidly extract data from *more than one* table.

A multiple table `SELECT` statement can become *very complex*, and (unfortunately) the syntax can often seem somewhat counterintuitive — this is largely because the lack of general programming constructs in SQL.

The key to mentally parsing SQL statements is to keep in mind the fundamental “row-processing loop”

- *Construct rows* according to the `FROM` statement
- *Filter rows* according to the `WHERE` statement
- *Extract columns* according to the `SELECT` statement

A sample schema

We use the following sample tables:

- `Student` – this stores student numbers and student names

```
CREATE TABLE Student(id CHAR(8), name VARCHAR(64));
```

- `Unit` – this stores unit codes and unit names

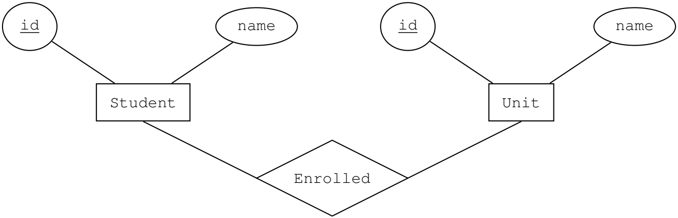
```
CREATE TABLE Unit(id CHAR(8), name VARCHAR(64));
```

- `Enrolled` – this stores enrolment information

```
CREATE TABLE Enrolled (sid CHAR(8), uid CHAR(8));
```

The *intention* of this set up is that the table `Enrolled` is meant to “connect” the other two tables — later we will see how to *enforce this* rule in SQL.

An ER diagram



A diagram like this is called an *entity-relationship* (or ER) diagram — it shows the *entities* being modelled and the *relationships* between them.

Student

```
mysql> SELECT * FROM Student;
+------+-------+
| id   | name  |
+------+-------+
| 1    | Amy   |
| 2    | Bob   |
| 3    | Chao  |
| 4    | Emily |
| 5    | Fan   |
+------+-------+
5 rows in set (0.00 sec)
```

There are a total of 5 students.

Unit

```
mysql> SELECT * FROM Unit;
+----------+-------------+
| id       | name        |
+----------+-------------+
| CITS1401 | Databases   |
| CITS1402 | Programming |
| MATH1001 | Maths 1     |
| MATH1002 | Maths 2     |
+----------+-------------+
4 rows in set (0.00 sec)
```

There are a total of 4 units.

Enrolled

```
mysql> SELECT * FROM Enrolled;
+------+----------+
| sid  | uid      |
+------+----------+
| 1    | CITS1401 |
| 2    | CITS1401 |
| 4    | CITS1401 |
| 2    | CITS1402 |
| 3    | CITS1402 |
| 4    | CITS1402 |
| 1    | MATH1001 |
| 2    | MATH1001 |
| 3    | MATH1001 |
+------+----------+
9 rows in set (0.00 sec)
```

There are a total of 9 enrolments.

A class list

With these tables, how can we find out who is taking `CITS1402`?

- The *enrolment information* is in `Enrolled`
- The *student name information* is in `Student`

Somehow we have to *combine* these tables to pull out the information.

The basic join

```
mysql> SELECT * FROM Student, Enrolled;
+------+-------+------+----------+
| id   | name  | sid  | uid      |
+------+-------+------+----------+
| 1    | Amy   | 1    | CITS1401 |
| 2    | Bob   | 1    | CITS1401 |
| 3    | Chao  | 1    | CITS1401 |
| 4    | Emily | 1    | CITS1401 |
| 5    | Fan   | 1    | CITS1401 |
| 1    | Amy   | 2    | CITS1401 |
| 2    | Bob   | 2    | CITS1401 |
| 3    | Chao  | 2    | CITS1401 |
| 4    | Emily | 2    | CITS1401 |
| 5    | Fan   | 2    | CITS1401 |
....
| 4    | Emily | 3    | MATH1001 |
| 5    | Fan   | 3    | MATH1001 |
+------+-------+------+----------+
45 rows in set (0.00 sec)
```

Yikes, why are there 45 rows in this table?

The basic join

MySQL produces *every possible row* constructed by “gluing together” a row from `Student` and a row from `Enrolled`.

```
+------+-------+          +------+----------+
| id   | name  |          | sid  | uid      |
+------+-------+          +------+----------+
| 1    | Amy   |          | 1    | CITS1401 |
| 2    | Bob   |          | 2    | CITS1401 |
| 3    | Chao  |
...                       ...
```

This give us

```
+------+-------+------+----------+
| id   | name  | sid  | uid      |
+------+-------+------+----------+
| 1    | Amy   | 1    | CITS1401 |
| 2    | Bob   | 1    | CITS1401 |
| 3    | Chao  | 1    | CITS1401 |
...
```

Cartesian product

In fact, this command has computed the *entire Cartesian product*

$$\texttt{Student} \times \texttt{Enrolled}$$

The Cartesian product contains rows whose “first half” and “second half” relate to *different students*, but we want the join to compute *only* the valid rows.

In other words we want to “match up” the rows so that we only keep the ones where the `id` column matches the `sid` column.

Use WHERE

```
SELECT *
FROM Student, Enrolled
WHERE id = sid;
+------+-------+------+----------+
| id   | name  | sid  | uid      |
+------+-------+------+----------+
| 1    | Amy   | 1    | CITS1401 |
| 2    | Bob   | 2    | CITS1401 |
| 4    | Emily | 4    | CITS1401 |
| 2    | Bob   | 2    | CITS1402 |
| 3    | Chao  | 3    | CITS1402 |
| 4    | Emily | 4    | CITS1402 |
| 1    | Amy   | 1    | MATH1001 |
| 2    | Bob   | 2    | MATH1001 |
| 3    | Chao  | 3    | MATH1001 |
+------+-------+------+----------+
9 rows in set (0.00 sec)
```

The class list

We need to modify this in two ways — just print the *names* and only for the rows corresponding to `CITS1402`.

```
SELECT name
FROM Student, Enrolled
WHERE id = sid
      AND uid = ’CITS1402’;
+-------+
| name  |
+-------+
| Bob   |
| Chao  |
| Emily |
+-------+
3 rows in set (0.00 sec)
```

The second `WHERE` condition is playing a subtly different role to the first — the first condition is “*setting up the correct table*” while the second condition is “*selecting the rows we want*”.

Moving the join condition

We can separate out the join condition using a different construct that explicitly highlights the join — this is the `JOIN...ON` construction.

```
SELECT *
FROM Student JOIN Enrolled
    ON id = sid;
+------+--------+------+----------+
| id   | name   | sid  | uid      |
+------+--------+------+----------+
| 1    | Amy    | 1    | CITS1401 |
| 2    | Bob    | 2    | CITS1401 |
| 4    | Emily  | 4    | CITS1401 |
| 2    | Bob    | 2    | CITS1402 |
| 3    | Chao   | 3    | CITS1402 |
| 4    | Emily  | 4    | CITS1402 |
| 1    | Amy    | 1    | MATH1001 |
```

Put the WHERE conditions back

```
SELECT *
FROM Student JOIN Enrolled
ON id = sid
WHERE uid = ’CITS1402’;
+------+-------+------+----------+
| id   | name  | sid  | uid      |
+------+-------+------+----------+
| 2    | Bob   | 2    | CITS1402 |
| 3    | Chao  | 3    | CITS1402 |
| 4    | Emily | 4    | CITS1402 |
+------+-------+------+----------+
```

The phrase `INNER JOIN` can be used rather than `JOIN`, although they have exactly the same meaning.

Cartesian Products

There are (at least) *three other* ways to get the Cartesian product of two tables.

```
SELECT * FROM Student CROSS JOIN Enrolled;
SELECT * FROM Student CARTESIAN JOIN Enrolled;
SELECT * FROM Student JOIN Enrolled;
```

A three-table join

Suppose we want a class list containing the names of students taking Databases (i.e. this time we don't know that the right code is `CITS1402`.

- We need the `Student` table for the *student name* information
- We need the `Unit` table for the *unit name* information
- We need the `Enrolled` table to "connect" the right students with the right units

Triple product

```
SELECT *  FROM Student, Enrolled, Unit;
+------+-------+------+----------+----------+-------------+
| id   | name  | sid  | uid      | id       | name        |
+------+-------+------+----------+----------+-------------+
| 1    | Amy   | 1    | CITS1401 | CITS1401 | Databases   |
| 1    | Amy   | 1    | CITS1401 | CITS1402 | Programming |
| 1    | Amy   | 1    | CITS1401 | MATH1001 | Maths 1     |
| 1    | Amy   | 1    | CITS1401 | MATH1002 | Maths 2     |
| 2    | Bob   | 1    | CITS1401 | CITS1401 | Databases   |
| 2    | Bob   | 1    | CITS1401 | CITS1402 | Programming |
...
...
| 5    | Fan   | 3    | MATH1001 | CITS1402 | Programming |
| 5    | Fan   | 3    | MATH1001 | MATH1001 | Maths 1     |
| 5    | Fan   | 3    | MATH1001 | MATH1002 | Maths 2     |
+------+-------+------+----------+----------+-------------+
180 rows in set (0.00 sec)
```

Doing the join

This produces the *triple* Cartesian product

$$\texttt{Student} \times \texttt{Enrolled} \times \texttt{Unit}$$

so what conditions are needed to ensure that the join makes sense?

- We need `id` = `sid` to correctly join `Student` and `Enrolled`
- We need `uid` = `id` to correctly join `Enrolled` and `Unit`

But we have *two columns* called `id`?

Disambiguation

```
SELECT *
FROM Student JOIN Enrolled JOIN Unit
ON id = sid AND uid = id;

ERROR 1052 (23000): Column ’id’ in on clause is ambiguous
```

The error message says it all — the column `id` is ambiguous, so we need to be able to specify “the `id` column that originally came from `Student`”.

Qualifying the columns

```
SELECT *
FROM Student JOIN Enrolled JOIN Unit
ON Student.id = sid AND Unit.id = uid;
+------+-------+------+----------+----------+-------------+
| id   | name  | sid  | uid      | id       | name        |
+------+-------+------+----------+----------+-------------+
| 1    | Amy   | 1    | CITS1401 | CITS1401 | Databases   |
| 2    | Bob   | 2    | CITS1401 | CITS1401 | Databases   |
| 4    | Emily | 4    | CITS1401 | CITS1401 | Databases   |
| 2    | Bob   | 2    | CITS1402 | CITS1402 | Programming |
| 3    | Chao  | 3    | CITS1402 | CITS1402 | Programming |
| 4    | Emily | 4    | CITS1402 | CITS1402 | Programming |
| 1    | Amy   | 1    | MATH1001 | MATH1001 | Maths 1     |
| 2    | Bob   | 2    | MATH1001 | MATH1001 | Maths 1     |
| 3    | Chao  | 3    | MATH1001 | MATH1001 | Maths 1     |
+------+-------+------+----------+----------+-------------+
9 rows in set (0.00 sec)
```

Aliases

```
SELECT *
FROM Student S JOIN Enrolled E JOIN Unit U
ON S.id = E.sid AND E.uid = U.id
WHERE U.name = 'Databases';
+------+-------+------+----------+----------+-----------+
| id   | name  | sid  | uid      | id       | name      |
+------+-------+------+----------+----------+-----------+
| 1    | Amy   | 1    | CITS1401 | CITS1401 | Databases |
| 2    | Bob   | 2    | CITS1401 | CITS1401 | Databases |
| 4    | Emily | 4    | CITS1401 | CITS1401 | Databases |
+------+-------+------+----------+----------+-----------+
```

The phrase `Student S` in the `FROM` clause means: “Use `S` as an alias for `Student` for this query”.

Natural Join

It is common for two tables to have columns with identical names because they refer to the same thing — for example, both `City` and `CountryLanguage` have a column `CountryCode` referring to the country.

```
SELECT Name, Language
FROM City C JOIN CountryLanguage L
ON C.CountryCode = L.CountryCode
WHERE Name = ’Perth’;
+-------+----------------+
| Name  | Language       |
+-------+----------------+
| Perth | Arabic         |
| Perth | Canton Chinese |
| Perth | English        |
| Perth | German         |
| Perth | Greek          |
...
```

Natural Join

The `NATURAL JOIN` operator joins tables by matching *all columns* with the *same names*:

```
SELECT Name, Language
FROM City NATURAL JOIN CountryLanguage
WHERE Name = ’Perth’;
+-------+----------------+
| Name  | Language       |
+-------+----------------+
| Perth | Arabic         |
| Perth | Canton Chinese |
| Perth | English        |
| Perth | German         |
...
```

Being careful

The NATURAL JOIN may have some unexpected consequences in terms of the *other columns* — if a new column gets added to one of the tables that happens to have the same name as a column in the other, then the behaviour will mysteriously change.

To be safe, it is better to always make joins explicit.

```
SELECT Name, Language
FROM City JOIN CountryLanguage
  USING (CountryCode)
WHERE Name = 'Perth';
```