Topic Six: Advanced Databases

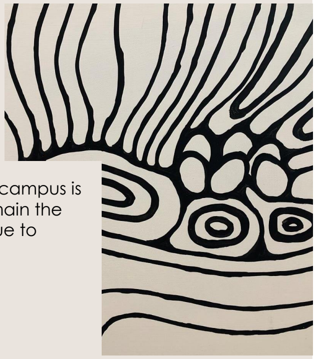
INMT5526: Business Intelligence

Semester One, 2024

Tristan W. Reed
UWA Business School

Acknowledgement of country

The University of Western Australia acknowledges that its campus is situated on Noongar land, and that Noongar people remain the spiritual and cultural custodians of their land, and continue to practise their values, languages, beliefs and knowledge.


Artist: Dr Richard Barry Walley OAM

Housekeeping

- **Quiz One:** well done, aim to get results back per the UWA Assessment Policy (~3wks);
- **Group Assignment:** you can now register into teams via LMS, specification out soon;
- **Timetable:** we now continue for seven weeks until the end of semester, ANZAC day.

Introduction to joining tables

- We know that we can easily relate multiple tables together.
  - Using foreign key relationships to refer to an observation (row) in one table from another.
  - This creates a 'reference' or connection between the two tables.
- This has the effect of 'extending' the row in one table.
  - This combined row in the first table logically contains the content of the row in the other (second) table as well as this initial row, combined.
  - Practically, we have to look up both rows separately – which can be a pain if we have anything other than a one-to-one relationship.

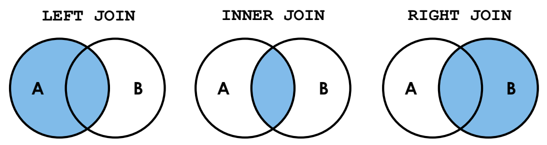
Logic of joining tables

- We can combine both tables together using a **`JOIN`** query (**`SELECT`**) that we write.
  - Based upon a specified equation that describes how the tables should be joined, the result is a single table with multiple rows derived from the two tables.
  - The equation specifies which attribute in the first table should be matched with which attribute in the second table.
  - While it logically makes sense for us to use a foreign key relationship, we can link any two attributes where there are matching values.
  - This can be of "any" type of relationship (many-to-many are more complex).

The types of joins

- There are three ways we can join a table, depending on which rows from which tables we wish to keep (as it won't be guaranteed there are always matches):
  - **`LEFT JOIN`**: all observations from the first table and matching from the second;
  - **`RIGHT JOIN`**: matching observations from the first table and all from the second;
  - **`INNER JOIN`**: matching observations that are within both tables;
- The most appropriate type of join to utilise depends on the context of the problem being solved – there is no 'one size' fits all.
  - **`INNER JOIN`**s are used quite often though;
- Other types of joins exist which we have not covered, as well as **`UNION`**s that allow us to see all observations in both tables.

Visually, the types of joins



Syntax format of joins

- The format for joining two tables (lets call them **`TableA`** and **`TableB`**) within MySQL is relatively straightforward:
  - **`SELECT * FROM TableA INNER JOIN TableB ON TableA.TheID = TableB.TheID;`**
- We can see it is the presence of the **`INNER JOIN`** keyword that differs.
  - **`<SecondTable> ON`** syntax specifies the 'other' table (**`<SecondTable>`**) we wish to join;
  - The equation to the right of it is very simple - 'this attribute in this table equals that one';
- Rarely can we use a **`*`** in this **`SELECT`** statement.
  - Note the prefix of the tables and which columns have been selected in the **`ON`** clause.
  - We may need to prefix attributes or use an **`AS`** keyword to rename conflicting ones.

Joins in practice

- Consider the following two tables:

TableOne

idOne	dataOne
1	Hello
2	What's
3	Happening

TableTwo

idTwo	dataTwo
2	World
4	Up
6	Now!

- Let us join **idOne** in **TableOne** (left) to **idTwo** in **TableTwo** (right), in various manners.

Joins in practice (II)

- **INNER JOIN**: only the ID values that are in common (value of 2 matched):

idOne	dataOne	idTwo	dataTwo
2	What's	2	World

- **LEFT JOIN**: only the ID values in the left (**TableOne**) are kept, with only one match:

idOne	dataOne	idTwo	dataTwo
1	Hello	*Null*	*Null*
2	What's	2	World
3	Happening	*Null*	*Null*

Joins in practice (III)

- **RIGHT JOIN**: only the ID values in the right (**TableTwo**) are kept, with only one match, which is no different to the match for **LEFT JOIN**:

idOne	dataOne	idTwo	dataTwo
2	What's	2	World
Null	*Null*	4	Up
Null	*Null*	6	Now!

Joins in practice (IV)

- How would we get all ID values - but only one is matched still?

idOne	dataOne	idTwo	dataTwo
1	Hello	*Null*	*Null*
2	What's	2	World
3	Happening	Null	Null
Null	*Null*	4	Up
Null	*Null*	6	Now!

- We can do a **UNION** of two queries – one **LEFT JOIN** and one **RIGHT JOIN**.
  - Do not worry about the practicalities of that (for this unit).

Creating views

- Consider our generated columns, where we can have **`VIRTUAL`** or **`STORED`** values that are either calculated on-the-fly or stored.
- We can apply the same principle to an entire table – that is, we can generate a 'new' table from the result of a **`SELECT`** query.
  - This could be a reduced version of a single table (i.e. filtered with **`WHERE`**);
  - Alternatively, it could also be the result of a **`JOIN`** query on two tables.
- We can then access this table as we would with any other – plainly!
  - **`SELECT * FROM ViewName;`** and so forth…

Syntax of creating a view

- It is very simple to create a view with MySQL:
  - **`CREATE VIEW <ViewName> AS SELECT`**…
  - The rest of the standard **`SELECT`** query follows this above, including any clauses.
- This view then effectively becomes a new table and can be queried normally.
- We can do a few special things with views if we wish:
  - **`CREATE OR REPLACE VIEW <ViewName> AS SELECT`**… if we want to update our view;
  - **`DROP VIEW <ViewName>;`** to remove the view;
  - **`SHOW CREATE VIEW <ViewName>;`** is also a thing – much like for tables.

Examples of views

- All of the following are valid for creating a view, let us assume the table **`ExampleTable`** actually exists with attributes, preferably with data in it…
  - **`CREATE VIEW ExampleView AS SELECT * FROM ExampleTable;`**
  - **`CREATE VIEW ExampleViewTwo AS SELECT AttributeOne FROM ExampleTable;`**

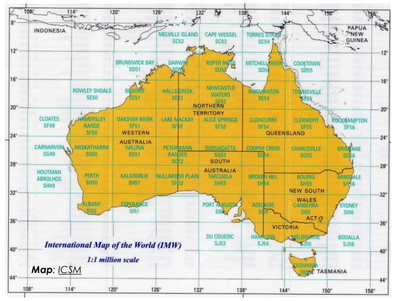
  -
    ```
    CREATE VIEW ExampleViewThree AS SELECT AttributeThree, MAX(AttributeTwo)
    FROM ExampleTable WHERE AttributeThree < 9000 GROUP BY AttributeThree
    ORDER BY AttributeThree DESC;
    ```
- Yes, we can have views on **`SELECT`** queries that **`JOIN`** tables together – I say again!

Spatial data

- Spatial data allows us to add new attributes which describe the physical location of each observation within the real world.
  - We can specify the spatial reference system to describe how the location is described.
  - Generally, we use latitude and longitude to refer to the location of phenomena.
- We can store a variety of different types of data that describe the world.
  - These primarily fall into the groups of lines, points and polygons.
- We will focus only on creating and analysing point data in this exercise.

Longitude and Latitude

- Think of longitude as going “a**long**” and latitude as “the other one”.



Map: ICSM

Creating Point data

- In MySQL, we simply provide the spatial data type when we define an attribute in a **`CREATE TABLE`** or **`ALTER TABLE`** statement, i.e.:
  - **`CREATE TABLE GeometryTable (geometry POINT);`**
- We can then **`INSERT`**, or use in other ways, the spatial data itself using the following format as our value (there are other methods as well):
  - **`ST_GeomFromText(‘POINT(<Long> <Lat>)’)`** where **`<Long>`** is the longitude and **`<Lat>`** is the latitude – note this format very specifically!
- We must ensure that the data that is inserted is valid – what makes it so?

Spatial queries

- Due to the nature of spatial data, we are able to write more complex queries that take advantage of this information.
  - For example, finding locations within X kilometres from a specified location.
- A very simple (or is it really that simple?) query we can write is to filter for locations within 10 kilometres of a given point:
  - **`SELECT * FROM <TableName> WHERE ST_Distance_Sphere(<GeomField>, ST_GeomFromText(<Location>)) <= 10 * 1000 ORDER BY name;`**

Viewing geometries as text

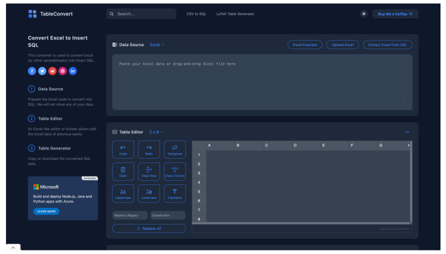
- For some reason, MySQL Shell “ruins” looking at geometry fields.
  - Rather than seeing (for example) **`POINT(0 0)`** in a geometry field, we instead see a sea of random characters when we **`SELECT`** a geometry field.
  - We can instead use the syntax **`ST_AsText(<FieldName>)`** in a **`SELECT`** statement column list to force MySQL Shell to retrieve this in text format (as it should!).
- Hence, whenever we **`SELECT`** a geometry field, we should do it this way.
  - Keep in mind that we will probably want to also use other **`ST_`** functions at the same time, as seen in the previous slide – and that’s okay.

Loading Spreadsheets

- Quite often we will want to translate a spreadsheet into a database.
  - We know the similarities and the benefits of a database over a spreadsheet.
  - We also know the effort required to craft **`INSERT`** queries.
- We must consider if there a quicker way to load a spreadsheet in.
  - Could we automatically generate the **`INSERT`** queries based upon the content of a spreadsheet, using an (automated) system to generate the queries?
  - The answer is "yes", and sadly this is the best way we can 'accelerate' it – it will always boil down to a set of queries!

Spreadsheet to Database

- There exists many tools, one is: https://tableconvert.com/excel-to-sql.



Convert Excel to Insert SQL

This converter is used to convert Excel (or other spreadsheets) into Insert SQL.

1 Data Source

Prepare the Excel code to convert into SQL. We will not store any of your data.

2 Table Editor

An Excel-like editor or builder allows edit the Excel data of previous easily.

3 Table Generator

Copy or download the converted SQL data.

Paste your Excel data or drag-and-drop Excel file here

Build and deploy Node.js, Java and Python apps with Azure.

Alternate method: loading in data

- There is an alternate method we can look at compared to using online tools to generate **`INSERT`** statements to load data from a table.
- Alternatively, we can also use a **`LOAD TABLE`** command instead.
  - **`LOAD DATA INFILE '<FileName>' INTO TABLE <TableName>;`**
  - This is an example and you would replace **`<FileName>`** and **`<TableName>`**.
  - Must specify the full file path, or have 'moved' into the correct folder first – a la 'script'.
- You must **`CREATE TABLE`** before you can use this!
  - Therefore, the table schema (attributes, datatypes etc.) must at least somewhat match the format of this file!

Modifiers for loading in data

- There are also additional options that can be supplied as additional clauses:
  - **`IGNORE <X> LINES`**: to ignore the first **`<X>`** lines of the file;
  - A standard 'column list' like in an **`INSERT`** statement to load for only some columns;
  - **`SET <ColumnName> = <Value>`**: to directly set **`<ColumnName>`** to **`<Value>`**;
  - **`@var<X>`**: refers to column number **`<X>`** from the input file.
- We won't be using this, but it is another tool for your collection of knowledge.
  - More info at https://dev.mysql.com/doc/refman/8.0/en/load-data.html.

Any leftover database work?

- This concludes our relational database content in INMT5526 (Business Intelligence).
  - What did we cover? Quite a lot but not quite everything!
  - Hopefully, it was at least somewhat enjoyable!
  - In the labs, we will have some time to spend for revising any content you wish to or to cover any content you have missed throughout the last few weeks.
  - Alternatively, you can spend some more time practising!
- If all goes to plan, this lecture should finish a little early too – more time for questions!

The End: Thank You

Any Questions? Ask now or via email (tristan.reed@uwa.edu.au)