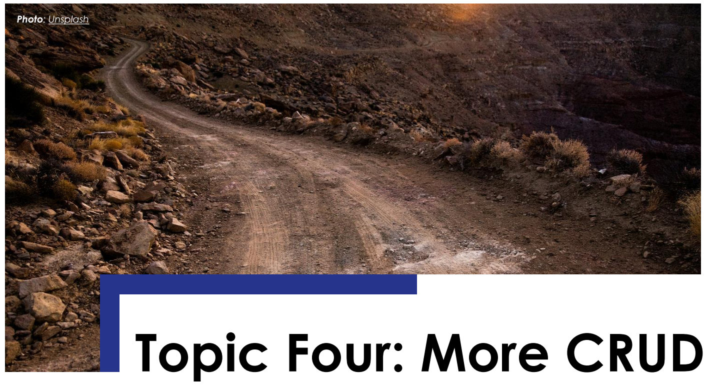


Topic Four: More CRUD

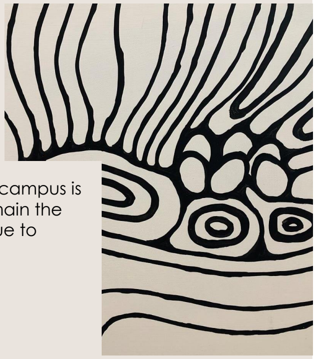
INMT5526: Business Intelligence

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

Running a 'script' in MySQL shell

- ***The following is a shortcut that may help us and save time with our analysis.***
- We often want to issue a lot of commands to effect something.
  - We also want to check the commands first before we issue them and also want to make sure that we don't enter them incorrectly – lest something go wrong!
- One solution to this is to use the "sourcing" function of MySQL shell.
  - We can run a text file full of (SQL) commands that we have authored in Notepad.
  - These will be effected immediately, one after another, top to bottom of the file.
- There are other considerations as well that we'll go over on the next slide.

Steps to running a 'script'

- Ignore whether or not at this point you consider this a script.
  - Some would argue whether or not it is (much like whether MySQL is a programming or scripting language, or neither) but we want to run a set of commands automatically.
- First, write the commands using Notepad (or a similar editor) and save the file with a `.txt` extension – or preferably, an `.sql` one.
  - Ensure that you have one command per line and each ends with a semicolon.
  - Execution will end if there is an error, at the command with an error.
  - Note the location upon where you have saved the text file.

Steps to running a 'script' (II)

- You will then need to open a new MySQL shell, but first recall the directory where you saved the `.txt` (or `.sql`) file – it will be easier if it doesn't have spaces in it.
  - Windows users should use Windows Explorer to find the path to the file they made;
  - Mac users should use Finder to find the path to the file they made.
- UniApps users (no matter which OS they run as 'host') should save their files in OneDrive.
  - This includes when you are running it on your own machine and if you are using the lab machines with UniApps as well. At least it will be synched between them!
  - This will mean your path will be something along the lines of `C:\Users\<StudentID>\OneDrive – UWA\` with wherever in your OneDrive the file is on the end of that and with your `<StudentID>` substituted in.

Steps to running a 'script' (III)

- The MySQL shell can then be launched from the command line.
  - Mac and Linux users can just run the command `mysqlsh` followed by the `\connect <StudentID>@db.tris.id.au` command in the Terminal.
  - Windows users may need to find the full path to MySQL shell, then follow with the `\connect <StudentID>@db.tris.id.au` command above. However, please try `mysqlsh` in Command Prompt first, before trying the full path.
- Once in the MySQL shell, you must enter SQL command mode using issuing the `\sql` command to the shell, as we will always do from now on!

Steps to running a 'script' (IV)

- Then, you can actually run the script by issuing the command as follows: `\. <filename>`, where `<filename>` is the name of the 'script' file to run.
  - This includes the full path to the file, as well as the filename!
  - You will need to surround the full path with quotes, if there is a space in it.
  - Note that in this case there is no semicolon at the end of the command.
  - Don't forget the `.txt` (or `.sql`) at the end of the filename.
  - Note the output in the terminal to see if it actually worked and if so, correctly and fully.
  - Alternatively, you can use `\source` instead of `\.` – no reason why you should, though!
  - You can use the command `\system ls` (macOS) or `\system dir` (Windows, including all UniApps systems) to see what current folder MySQL Shell is believed to be in.

Generated columns

- ***Now, on to some additional considerations with our CRUD operations.***
- A 'generated' attribute (or column), like the name would suggest, consists of a value that is not set directly for each observation.
  - Rather, it is generated from a mathematical function on other column(s).
  - Hence, if they are set (and not null!), the value for this column is set automatically.
- This can be useful for basic data summarisation and analysis.
  - For example – rather than storing 'redundant' data, the result of a function on two other columns can be calculated and utilised 'on the fly'.
  - We won't cover (today) applying a function on selection of column(s) as an 'extra attribute' at the end of a `CREATE TABLE` statement.

Generated columns: the process

- The main keyword 'phrase' used to effect the creation of a generated column is **`GENERATED ALWAYS AS (<function>);`** which is used as a modifier on a column.
  - Where **`<function>`** involves one or more other columns.
  - We can add **`VIRTUAL`** or **`STORED`** before the semicolon to decide when to calculate it.
  - The **`CONCAT()`** function can be used to combine ('concatenate') one or more text fields (columns/attributes) – think of it like 'gluing' them together.
  - However, mathematical operators can also be used on numeric types (+-/*).
- Column names only need to be specified on their own.
  - The table and/or database names are not needed!

An example: generated column

- **`CREATE TABLE Example(id INT NOT NULL AUTO_INCREMENT, isTrue BOOLEAN, someQuantity INT, otherQuantity INT, combinedQuantity INT GENERATED ALWAYS AS (someQuantity * otherQuantity), PRIMARY KEY (id));`**
  - A table is created with some attributes, where one attribute is derived from the others.

Removing a primary key

- Unlike using a **`MODIFY`** command to remove other modifiers, we can't do the same with a **`PRIMARY KEY`**, as it is set differently.
  - We must instead **`DROP`** the primary key as it is considered a separate constraint.
- This is achieved with the command as follows:
  `ALTER TABLE <TableName> DROP PRIMARY KEY;`
  - Other constraints and modifiers should use the commands in last week's lecture.
  - We can then re-create the primary key using a command as follows:
    `ALTER TABLE <TableName> ADD PRIMARY KEY(<ColumnName>);`

Additional modifiers / constraints

- There are also some other modifiers to constrain values added to particular attributes, which we can do (apply) to various differing fields.
  - **`DEFAULT <defaultValue>`** to specify a default value of **`<defaultValue>`** for the field.
  - **`CHECK (<formula>)`** as an additional item in the attribute list to ensure that a constraint described by **`<formula>`** must be satisfied before a value is added (or changed!) for a particular column(s).
- We can name a constraint in the format **`CONSTRAINT <name> CHECK (<formula>)`** if we wish to do so for clarity or simplicity.

Some examples: modifiers

- **`ALTER TABLE Example MODIFY COLUMN isTrue BOOLEAN DEFAULT 1;`**
  - The value of the **`isTrue`** attribute does not need to be specified; will be **`TRUE`** (`1`) by default, each time that data is added to the table.
- **`ALTER TABLE Example MODIFY COLUMN someQuantity INT NOT NULL;`**
  - The column **`someQuantity`** must always have a value supplied.
- **`ALTER TABLE Example ADD CONSTRAINT sqCons CHECK (someQuantity > 0);`**
  - The value of **`someQuantity`** must always be greater than zero (i.e. positive);
  - Could be combined with the **`NOT NULL`** example above.

Constraints on tables

- If we wish to apply a constraint on an attribute that already exists in a table that already exists, we can do it as a **`ALTER TABLE <TableName> ADD CHECK (<formula>)`** command on the table itself.
  - Or alternatively in/as the **`ALTER TABLE <TableName> ADD CONSTRAINT <name> CHECK (<formula>);`** command format.
  - We can also do it as a **`MODIFY`** statement on the attribute, if we must.
- We consequentially can replace **`ADD`** with **`DROP`** to remove the constraint.
  - If we apply the constraints on the attribute, we can use a **`MODIFY`** column command.

Complex 'where' clauses

- Last week, we introduced the concept of a **`WHERE`** clause to a command.
  - This allows us to ensure a command only applies to a subset of rows/observations.
- However, we only looked at matching a column to a single value in our formulas that we define – the example being **`id = 1`**.
  - This is obvious quite limited in what is can achieve – maximum amount of rows matched!
  - To solve particular use cases, we often want more complex formula that we can apply.
- Two things we can use are *operators* and *keywords*.
  - Allow us to define more than one row/instance to be matched.

Complex 'Where' operators

- We can use the following operators (ignoring the colons) to compare values of a field (attribute) in a more complex way than just 'is equal to'.
  - **`=:`** is equal (the same) – won't work for floating point values;
  - **`>:`** greater than (numeric types) or **`<:`** less than;
  - **`>=:`** greater than or equal to or **`<=:`** less than or equal to;
  - **`<>:`** Not equal to (sometimes written as **`!=`**);
  - **`BETWEEN:`** between a certain range (specified – **`x AND y`**);
  - **`LIKE:`** basic pattern matching (using `%` as a wildcard);
  - **`IN:`** matches a value from a comma-separated list of values.

Complex ‘Where’ keywords

- To make things even more finer-grained (albeit complicated), we can combine formula by using the following commands:
  - **AND**: both the left and right side must be true to match;
  - **OR**: either (or both of) the left and right side must be true to match;
  - **NOT**: only applied to one formula, gets the opposite result (what is true is false).
- **NOT** can be combined with the other two.
  - However, in either case – use plenty of brackets for clarity!

An example: complex ‘Where’

- An example of a ‘compound’ (complex) **WHERE** statement in a query:
  - **SELECT (date, customerName) WHERE (transactionItem > 2) AND (date BETWEEN ‘2023-01-01’ AND ‘2023-12-31’);**

id	date	customerName	transactionItem
1	1/1/2022	Example Corporation	4
2	1/4/2022	Nikola Limited	2
3	1/4/2023	Pear Computers	1
4	31/7/2023	Western Mining	3

Foreign key relationships

- Much in the manner we define primary keys, we can also define foreign keys.
  - The ‘other’ table and its primary key must first already exist before a foreign key relationship can be made to it from a different table.
- We add in the format of an ‘extra’ attribute, much in the same manner as a PK: **FOREIGN KEY (<AttributeName>) REFERENCES <TableName>(<ColumnName>)**.
  - We can also name it through instead describing this as **CONSTRAINT <ConstraintName> FOREIGN KEY <AttributeName> REFERENCES <TableName>(<ColumnName>)**.
  - This alternate method allows us to define multiple-attributed foreign keys as well!

Foreign key relationships

- It is important to note *we must also define the foreign key as an attribute*.
  - For example, if **OrderID** in another table is a foreign key and of type **int**, this table must define an attribute of type **int** (of any name – but **OrderID** is a good one!).
- We can also remove a foreign key or add it to an existing table in the same manner as a primary key (self-explanatory at this point!)
  - **ALTER TABLE <TableName> ADD FOREIGN KEY**…
  - **ALTER TABLE <TableName> ADD CONSTRAINT**…
  - **ALTER TABLE <TableName> DROP FOREIGN KEY**…

Foreign key constraints

- We can add constraints to **`FOREIGN KEY`**s to describe what happens when the relationship is broken (i.e. generally if the row referred to is deleted or updated).
  - After **`REFERENCES`**, we add **`ON DELETE <method>`** and/or **`ON UPDATE <method>`**.
- The 'referential actions' we can mention in the constraint are as follows:
  - **`CASCADE`**: delete (or, depending on design, update) the row referred to;
  - **`SET NULL`**: set the value referred to as **`NULL`** ('empty'/blank) – ensure possible;
  - **`RESTRICT`**: prohibit the action from happening (related is **`NO ACTION`**);
  - **`SET DEFAULT`**: not actually usable in practice with our setup.

An example: foreign key

- Assume we have two tables: **`Customer`** and **`Invoice`**;
  - Customer is referenced in Invoice – i.e. Foreign Key on **`CustomerID`** in **`Invoice`**.
- We would hence define a foreign key in terms of the following query at creation:
  - **`CREATE TABLE Invoice (id INT AUTO_INCREMENT, customerID INT, …, FOREIGN KEY (customerID) REFERENCES 21234567_db.Customer(id) ON DELETE CASCADE);`**
- Looking at the above, is **`CASCADE`** the most appropriate choice here?

Utilising indexes

- An index can be created on a primary key (or any other attribute) to speed up data retrieval – this uses processing power and storage space but is invisible to the user.
  - The above is on creation, on use, indexed attributes are quicker to query.
- This is achieved by running an additional query after creating the table:
  - **`CREATE UNIQUE INDEX <IndexName> ON <TableName>(<AttributeName>);`**
  - **`CREATE INDEX <IndexName> ON <TableName>(<AttributeName>);`**
  - **`CREATE INDEX <IndexName> ON <TableName>(<AttrOne>, <AttrTwo>);`**
- It should hopefully be self-explanatory what the 'fill in the blanks' are above.
  - **`<IndexName>`** is the self-chosen name for the index, **`<TableName>`** is the table's name, **`<AttrOne>`**, **`<AttrTwo>`** and **`<AttrName>`** are all names of attributes.

Utilising indexes (II)

- More options for indexes are available, but beyond the scope of this course.
  - Can specify technical options as to how the index is created, as well as whether or not the index is modifiable (or 'fixed' and unchangeable).
- To remove an index, we **`DROP`** it much like we would **`DROP`** a table:
  `DROP INDEX <IndexName>;`
  - Thankfully, this just **`DROP`**s the index, not the whole table and its data!

THE UNIVERSITY OF WESTERN AUSTRALIA

The End: Thank You

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