Photo: Unsplash

Topic Five: Sorting, Grouping & Aggregation

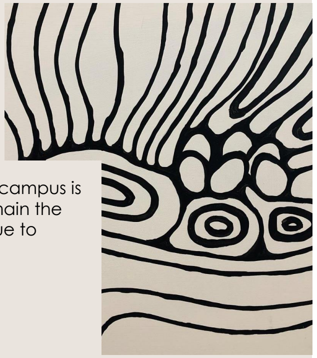
INMT5526: Business Intelligence

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

Today's context

- What have we covered so far in Business Intelligence?
  - Tools and techniques to help us make business decisions with data.
  - Basics and history of business intelligence and database systems.
  - How to CRUD our data models (tables) and data (observations) in a relational database.
  - Tips and tricks with database systems.
  - Selecting only certain observations (`WHERE` clauses).
- That leaves us with a few things to learn and do, in terms of databases:
  - Sorting data, grouping and aggregating data as well as learning some additional tips and tricks to help us interrogate our data.

Caveats of the `USE` command

- ***Some additional tips and tricks today to help you with your SQL!***
- We only need to issue the command once per session!
  - Once we close our session (i.e. `\quit`), we will have to issue the command again next time we connect to the database server (if we wish for it to apply);
  - If we wish to use a different database, we can issue another `USE` command;
  - Otherwise, the current database will continue to be used – there is no way to 'cancel'!
  - If we issue another `USE` command for the same database, nothing much will happen.
- Hence, make it a ritual to add it to the first line of your `.txt` file that you run, or the first thing that you do after going into `\sql` mode within the MySQL Shell!

Commenting your queries

- When writing a script file, it is often wise to be able to note information which shouldn't be executed (run) by the database server, but is instead for our benefit.
  - For example, a short English description of what the query does, especially if it is complicated – so you can go back to it and understand it later.
  - It also helps other people understand your queries (if you were to share it).
  - You may have a need to do so for an assessment – to explain what you have done and what question the query refers to, or a similar sort of thing.
- Thankfully, there are three ways we can add comments into our script file.

Comment formats for queries

- The three types of comments are as follows:
  - Starting with a **`#`** (hash symbol) - anything else until the next line becomes a comment.
  - Starting with a **`--`** (two dashes) - anything else until the next line becomes a comment.
  - Between **`/*`** and **`*/`** character sequences – anything between these two sequences (which can be over multiple lines) becomes a comment.
- A reminder: comments are not executed as queries!
  - So, this means they do not have to be SQL statements, although they can be.
- You are free to choose which format is appropriate and preferred.
  - It is suggested that you comment your work through the exercise, where relevant.

`SHOW CREATE TABLE` command

- We recall that the **`CREATE TABLE`** command is used to create our data model.
  - We specify the data type, modifiers and names of each of our attributes, alongside the name of our table (and the database it resides in).
  - We can use an **`ALTER TABLE`** command to add/change/remove these attributes.
- Through the use of a **`SHOW COLUMNS FROM`** command, we can see some of the above information, but not all of it – e.g. constraints are not shown.
  - This makes it hard to see what we have already done to our table!

The command itself

- What we can do instead, is bring back / view the **`CREATE TABLE`** command that would be required to build / generate the table the way it is now.
  - This allows us to see everything – including constraints!
- The command is as simple as: **`SHOW CREATE TABLE <TableName>;`**
  - Be careful – depending on the table, you may have a lot of data returned by this command that you will need to filter through – hence why we have the other way!

`SHOW CREATE TABLE` in action

```
CREATE TABLE `Transaction` (
    `id` int NOT NULL AUTO_INCREMENT,
    `date` date NOT NULL,
    `customerName` varchar(99) NOT NULL,
    `transactionItem` int DEFAULT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `TransactionIndex` (`id`),
    KEY `transactionItem` (`transactionItem`),
    CONSTRAINT `Transaction_ibfk_1` FOREIGN KEY (`transactionItem`)
    REFERENCES `TransactionItem` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

What can we see here?

`SHOW CREATE TABLE` in action (II)

```
CREATE TABLE `TransactionItem` (
    `id` int NOT NULL AUTO_INCREMENT,
    `description` varchar(255) NOT NULL,
    `quantity` int DEFAULT NULL,
    `unitPrice` float DEFAULT NULL,
    `hasGst` tinyint(1) DEFAULT '0',
    `totalPrice` float GENERATED ALWAYS AS ((`unitPrice` * `quantity`)) STORED,
    `taxAmount` float GENERATED ALWAYS AS ((`hasGst` * (`unitPrice` / 11))) STORED,
    PRIMARY KEY (`id`),
    UNIQUE KEY `TransactionItemIndex` (`id`),
    CONSTRAINT `qConstraint` CHECK ((`quantity` > 0)),
    CONSTRAINT `upConstraint` CHECK ((`unitPrice` > 0))
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

What *different things* can we see here?

Adding multiple values at once

- The syntax to add a new row to a database is quite verbose:
  - **`INSERT INTO <TableName> (<Columns>) VALUES (<Values>);`**
  - You can imagine that this is quite painful for multiple rows!
- We have a couple of speed-ups to make things easier!
  - No need to specify **`<Columns>`** if all columns are specified in **`<Values>`**;
  - We can separate multiple (**`<Values>`**) lists with a comma in a single command.
- No need to do anything – just remember this for when you need it!
  - We'll look at how to import spreadsheets next week.

Multiple values: an example

- **`INSERT INTO Transaction (date, customerName, transactionItem) VALUES ('2022-01-01', "Example Corporation", 4), ('2022-04-01', "Nikola Limited", 2), ('2023-04-01', "Pear Computers", 1), ('2023-07-31', "Western Mining", 3);`**
- What are the advantages and disadvantages of doing it this way?
  - *Pro*: Less work to write it out;
  - *Con*: Harder to diagnose errors.

What about special columns?

- Auto-incrementing and generated columns should be specified in the 'shorthand' manner, with their values detailed as **`DEFAULT`**:
  - **`INSERT INTO TransactionItem VALUES (DEFAULT, "Test", 10, 100.0, DEFAULT, 0, DEFAULT);`**
- Can you think of another situation where this would be useful?
  - When you've specified a default value for a field but still want to 'note' it.

Sorting values

- ***Now, moving on to sorting, grouping and aggregation…***
- There are many reasons we would want to sort values in our data.
  - What is the biggest (or smallest, or average…) value of X?
  - What is the most (or least) common value of Y?
  - How can I order products on my online store to aid with discovery?
- We can answer these questions through the use of sorting techniques within our database (and hence, its data within it).
  - We don't need to worry about the technical minutiae – however, we need to understand this can be very slow on very large datasets (tables)!

Sorting techniques

- Thankfully, this is another benefit of ensuring we have appropriate datatypes used for different types of data – such as numeric, date or text (character).
  - We can sort numeric (and date) types from smallest to largest or vice-versa;
  - We can sort text types from A to Z or vice-versa;
- We can sort on multiple columns if we wish!
  - Sort on column A, then sort on column B if both have an 'equal' value for column A.
  - Quite often, this is not necessary – the differences in column A mean no need for B.

Sorting Syntax

- To sort our data, we add an **`ORDER BY`** clause on the end of our **`SELECT`** query, before the semicolon but after a **`WHERE`** clause (if we have it).
  - After the words **`ORDER BY`**, we list (comma-separated) the attributes (columns) we wish to sort by, from first to sort by until last to sort by – it does not have to be all of them.
  - We can then state **`ASC`**(ending) or **`DESC`**(ending), depending on which direction that we'd like to sort by (for each attribute). What is the default behaviour, where it can be omitted?
- For example, a query to sort by one column in an ascending manner:

```
SELECT * FROM <SomeTable> WHERE id > 0 ORDER BY <SomeColumn> ASC;
```

Sorting in practice

- Consider our Transaction table from the Week 3 lab – dates and companies.
  - Company names in order: Example Corp, Nikola Ltd, Pear Computers, Western Mining.
  - This was driven by the order they were entered into the database.
- **`SELECT customerName from Transaction ORDER BY transactionItem ASC;`** Running the query above yields the following, from top-to-bottom:
  - Pear Computers, Nikola Limited, Western Mining, Example Corporation.
- Sorting on **`transactionItem`** sorted on the value of the foreign key – that's all.
  - Just because we sorted on it, doesn't mean we had to use (return) it.

Grouping and aggregation

- We can aggregate (group) data based upon shared values to understand the other properties of the rows (observations) of these groups.
  - Shared values, in this case, being the value of some attribute that is the same for more than one observation (row) – e.g. all items that have no GST in **`TransactionItem`**.
  - To do this, we must determine which columns we will group/aggregate by (i.e. which ones have the 'shared values'). Generally, we want some diversity here – multiple groups.
  - However, doing so means we consider all of the values that this column has throughout the table – as such, we may wish to filter first (**`WHERE`**) to ensure we only have some groups.

How we group

- The **`GROUP BY`** clause sits between the **`WHERE`** clause and the **`ORDER BY`** clause in our **`SELECT`** query (like sorting, it only makes sense for this type of query).
  - One or more columns are then specified which are the ones that will form the grouping.
  - The values for the first column are grouped, then the second and so forth.
  - The same caveats apply as they did with the sorting (**`ORDER BY`**) clause – consider a relevant number of groups that are meaningful to sort the problem.
  - You can see how this would be most useful for text, then date, then whole (integer) numeric and not very useful for floating point numeric.

Aggregating functions

- We must describe how we want these other values within the observations that make up each group to be summarised.
  - To do this, we have a set of aggregating functions that allow us to specify how we wish to summarise all the values for a particular attribute within the group.
  - As such, in a 'grouped' or 'aggregating' query, we must only include columns which we are grouping or summarising – otherwise it is meaningless and can cause errors.
- These functions will be listed on the next slide…

The actual aggregating functions

- The five most common aggregating functions that will be used are:
  - **`AVG(<ColumnName>)`** to get the mean of the values in **`<ColumnName>`**;
  - **`MIN(<ColumnName>)`** to get the smallest of the values in **`<ColumnName>`**;
  - **`MAX(<ColumnName>)`** to get the largest of the values in **`<ColumnName>`**;
  - **`SUM(<ColumnName>)`** to get the total of the values in **`<ColumnName>`**;
  - **`COUNT(<ColumnName>)`** to get the count of the values in **`<ColumnName>`**;
- It can be seen that these would only be appropriate for particular data types.
  - Which functions for which data types? It is not a 1:1 relationship.
- Other aggregating functions do exist, that we won't cover here.

Date functions

- Within our **`GROUP BY`** clause, we may only want to group on part of a date:
  - **`MONTH(<ColumnName>)`** to group by the month only;
  - **`YEAR(<ColumnName>)`** to group by the year only;
- We also have **`NOW()`** as a shortcut for the current date/time (useful for **`WHERE`**).
- There exists many more date arithmetic functions that you can also use.
  - You can read about them within the MySQL documentation, alongside the other aggregating functions that we did not cover – in this weeks' reading!
- These date functions can also be used in the **`SELECT`** side of the statement/query.

Example of a grouping query

- **`SELECT hasGst, MAX(unitPrice) FROM TransactionItem GROUP BY hasGst;`**
  - For a **`hasGst`** value of **`1`** (has tax), the maximium **`unitPrice`** is **`123.45`**;
  - For a **`hasGst`** value of **`0`** (no tax), the maximum unitPrice is **`100`** (ignoring the change);
- Why did we mention **`hasGst`** twice?
  - How would we know what grouped (maximum) value belongs to what group?

Putting it all together

- Consider a generic example for a table that does not exist:

```
SELECT department, MONTH(date), YEAR(date), COUNT(id), MIN(salePrice)
  FROM ExampleTable
  WHERE department LIKE 'SALES %'
  GROUP BY department, MONTH(date), YEAR(date)
  ORDER BY MIN(salePrice) DESC;
```

- A lot happening in this example – allows us to see a vast combination of possibilities.
  - Would this be 'too specific' in practice to be useful? Depends on the data.

Other relevant bits and pieces

- We can use AS to simplify the output of aggregating functions to simpler names.
  - **`MIN(someValue) AS minValue`** in the **`SELECT`** statement.
- Using 'rollup' allows us to generate a grand total of all groups, by specifying **`WITH ROLLUP`** at the end of a **`GROUP BY`** clause in our query.
  - For example – calculating a total for each year and then for all years, by adding it.
- Generally, aggregation functions ignore **`NULL`** values.
  - Be aware when you are grouping values that this is the case.
- There is a lot you can do with date functions in MySQL (and SQL generally).
  - It really does pay to read the readings, using the link on LMS.

Next week

- Next week is a tuition-free week, although we do have Quiz One.
  - After this, we will have our last week of relational database content!
  - We look to be covering joins, views, spatial data and spreadsheet loading.
- We'll then spend a week looking at NoSQL and graph databases.
  - We'll set up a NoSQL database up during the seminar/lab and then look briefly into how to query a graph database using their own query language (which builds upon SQL).
  - After that, we'll be looking at Power BI – so confirm that it (still) works on your system!

Quiz One

- The first Quiz will be held on Friday, 5 April 2024 between 9:59AM and 9:59PM.
  - You have a twelve hour window to complete these eight short answer-questions;
  - This should take you about an hour to do the work – everyone is different, though;
  - Answers will be written in a word-processor document and uploaded to Turnitin;
  - Hence, you can do this on your own PC at an appropriate time for you;
  - Topics covered will be everything in the first five *weeks* of learning activities.
- Remember the process for special consideration if this is required for you.
  - All the best and good luck with it – I will still be working, so do get in contact (if needed).

The End: Thank You

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