

Introductory Google BigQuery with Gemini AI

Lab 1

1	BIGQUERY AND GEMINI REFERENCE / DOCUMENTATION	2
2	CREATING A NEW PROJECT AND DATASET AND IMPORTING A TABLE	2
3	BASIC SELECT	5
3.1	SELECT WITH EXPRESSIONS AND ALIASES.....	7
3.2	SELECT WITH FUNCTIONS (STRING, MATH AND DATE)	8
3.3	SELECT WITH DISTINCT AND COUNT	9
3.4	SELECT WITH LIMIT	10
4	USING GEMINI IN BIGQUERY	10
4.1	WORKING WITH PROMPTS DIRECTLY IN THE SQL EDITOR TAB	11
4.2	REFINING PROMPTS IN THE GEMINI DIALOG BOX.....	12
4.3	REFINING EXISTING QUERIES THROUGH FURTHER PROMPTS IN THE EDITOR TAB.....	14
4.4	DETAILED PROMPTS VS GENERIC PROMPTS AND PROMPT REFINEMENT	15
4.5	USING GEMINI CLOUD ASSIST	15
4.6	REPLICATING PREVIOUS QUERIES	17
4.7	USING OTHER AI TOOLS TO GENERATE THE QUERY (OPTIONAL)	17
4.8	VALIDATING PROMPT RESULTS (QUERY CORRECTNESS)	18
4.8.1	<i>Obtain detailed plain-language description of what a query does</i>	<i>18</i>
4.8.2	<i>Run query against a sample truth table</i>	<i>19</i>
4.8.3	<i>Compare against prompt results from other gen AI tools</i>	<i>20</i>
4.8.4	<i>Validate with a human expert</i>	<i>20</i>
5	SORTING ROWS WITH ORDER BY	20
6	SAVING QUERIES, QUERY RESULTS AND VIEWING QUERY HISTORY	22
7	FILTERING WITH WHERE	24
7.1	USING THE AND, OR AND NOT OPERATORS.....	26
7.2	USING BETWEEN FOR RANGE TESTS.....	27
7.3	USING IN TO CHECK FOR MATCHING WITH OTHER VALUES	28
7.4	USING LIKE TO MATCH STRING PATTERNS	29
8	USING CASE TO IMPLEMENT CONDITIONAL LOGIC TO ADD COLUMNS.....	29
9	AGGREGATE FUNCTIONS: COUNT, SUM, AVG, MIN, MAX	31
9.1	REFINING QUERIES WITH AGGREGATE FUNCTIONS FOR COLUMN DETAILS	31
9.2	AGGREGATE FUNCTIONS WITH CASE CLAUSE.....	33
10	AGGREGATING AND GROUPING WITH GROUP BY	34
10.1	GROUPING MULTIPLE COLUMNS.....	36
10.2	USING HAVING CLAUSE TO FILTER ON GROUPS	36
11	END	38

1 BigQuery and Gemini reference / documentation

Main [official documentation page](#) for BigQuery.

[Top level overview](#) of BigQuery

Basics of [organization of resources](#) in BigQuery, including [datasets](#) and [tables](#).

Basics of working with BigQuery [via the console UI](#).

Quick start guide for working with [loading and querying data](#) as well as [querying public datasets](#).

Official [reference for GoogleSQL](#), the official SQL dialect for Google BigQuery.

Basic guide to [using Gemini to assist writing queries](#),

2 Creating a new project and dataset and importing a table

You should have a Google company account which allows you full access to the BigQuery functionality for this workshop.

We will start off by creating a new project in which we will create one or more datasets that will hold the various tables and views that we will use in this lab session.

NOTE: Your ability to create this project and datasets **will depend on your authorization level** in your organization. If you are not able to create this project / dataset, you will be added to my (trainer's) project so you can access these datasets

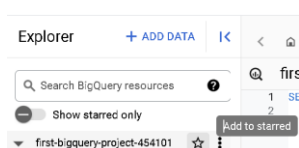
[Create a new project](#) with the name: `intro-bigquery-workshop`

Once you have received notification of the successful creation of the new project, you can directly [navigate to the BigQuery](#) main page.

You should see the newly created project name at the upper left of the top most bar – if it does not appear there, you can click on the project name that appears to switch to the project that you just created.



If you intend to create multiple projects later (or if you already have multiple projects in your account), you may optionally wish to star this latest new project so that you can easily locate it amongst the many other projects by switching on Show starred only.



We will [create a dataset](#) in this project with the name: `first_dataset`

You can skip this step if you wish, since you will access to the dataset that I create, but you can also choose to create your dataset if you wish.

IMPORTANT NOTE: If you are sharing the same project with me (the trainer), all the datasets that you create will be within in the same project. To ensure that the datasets you create are distinguishable from the datasets of other participants of this workshop, please precede the dataset with your name, for e.g. `peter_first_dataset`, `jane_first_dataset`, etc (make sure you use underscore and not dashes to separate the words).

We will use the `samplesales.csv` file in the `data` subfolder of the downloaded workshop resources to populate the contents of a new table in this new dataset. You can open `samplesales.csv` in Excel to quickly preview it first if you wish. This file contains dummy data for sales transactions over a period of time with a variety of relevant fields / columns such customer info, location of the customer, date of purchase, category of product, number of units purchased and the unit price.

[Create a table](#) in the newly created dataset with the following values in the dialog box that appears.

Create table from: Upload
Select file: `samplesales.csv`
File Format: CSV
Destination: `intro-bigquery-workshop`
Dataset: `first_dataset`
Table: `samplesales`
Table type: Native table

Tick Auto detect for Schema.

Schema

☒ Auto detect

 Schema will be automatically generated.

BigQuery will scan the contents of each column and infer the [data type](#) for each column as it imports them into the table that it will create.

In Advanced Options, type 1 for Header Rows to skip as the first row in our CSV file is essentially a header row containing the names of the columns/fields for the table we are creating. You can leave the other options as they are.

Advanced options ^

Write preference
Write if empty ▼

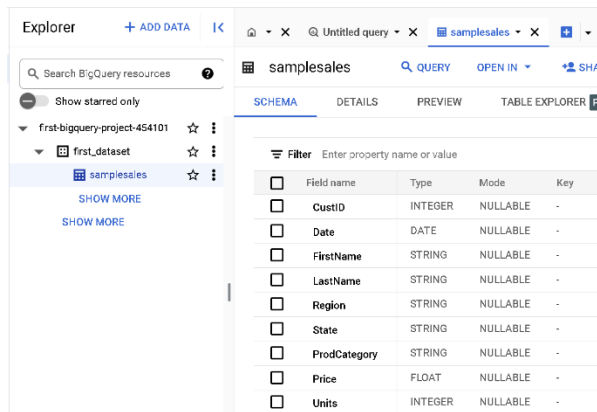
Number of errors allowed
0 ⓘ

Header rows to skip
1 ⓘ

Finally, click Create Table.

A message about load job running should appear followed by notification about successful creation of the `samplesales` table.

Selecting this table in the Explorer pane should show its Schema in the details pane, where you can see the [data types](#) that BigQuery has automatically assigned to each of the fields in the imported CSV file. The Nullable mode indicates this column can contain `null` values (to be covered in a later lab topic)



Click on Preview tab in the Details pane to view the first 50 rows in this table.

The screenshot shows the BigQuery interface with the Preview tab selected in the Details pane. It displays the first 7 rows of the `samplesales` table. The columns are Row, CustID, Date, and FirstName.

Row	CustID	Date	FirstName
1	18	2024-05-28	James
2	30	2023-10-16	Gabriel
3	46	2024-12-19	Benjamin
4	4	2023-01-02	Leo
5	10	2023-05-14	Cameron
6	14	2023-08-06	Noah
7	17	2022-07-22	Charles

Notice that the rows in this table (based on the CustID numbers) does not appear in the same sequence as the initial data in `samplesales.csv`

This is because the job executed by BigQuery to load the data from this file executes in parallel to populate the table with the data, resulting in rows appearing out of the original sequence.

The sequencing of the rows in the table will not affect the various queries that we will run next, and we can always perform a sort to display the rows based on ascending order of CustID, as we will see later.

3 Basic SELECT

One of the most **common tasks in basic data analytics is to retrieve data** from BigQuery tables using the SELECT statement. The SELECT statement is one of the most complex statements as it has many **optional clauses** that you can use within it to form flexible queries to retrieve data pertaining to any situation or condition. We will look at some of these clauses in more detail.

The basic syntax of the SELECT statement in GoogleSQL is as follows:

```
SELECT
    select_list
FROM
    table_name;
```

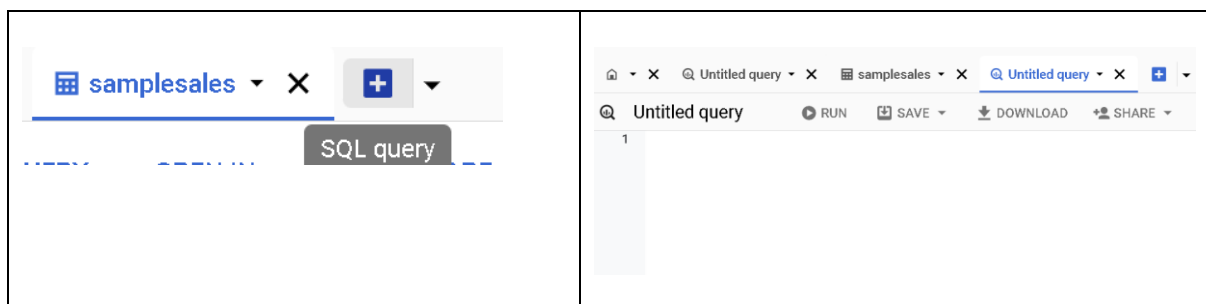
In this syntax, the `select_list` can be a column/field or a list of columns/fields in a table from which you want to retrieve data. If you specify a list of columns, you need to place a comma (,) between the columns to separate them. If you want to select data from all the columns of the table, you can use an asterisk (*) as shorthand instead of specifying all the column names.

The `table_name` must at the minimum have the dataset name that it is contained in preceding it to qualify it, (for e.g. `dataset_name.table_name`) since there may be multiple tables with the same name in different datasets in a BigQuery project. The formal way however is to have the project name as well to fully qualify and identify the table correctly (for e.g. `project_name.dataset_name.table_name`). You don't have to do this if you don't wish to, but queries generated from Gemini prompts will use this convention (to be shown later).

The `SELECT` and `FROM` are SQL keywords. These are case-insensitive, which means that `SELECT` is equivalent to `select` or `Select`. By convention, SQL keywords are typically written in uppercase to mark them as such as and make the queries easier to read.

The SELECT statement can be written on a single line or broken across multiple lines and is usually terminated with a semicolon (which is however optional).

Click on SQL query icon to create a new tab to type in a new SQL query, which will initially be untitled.



To view all the columns in all the rows in the current table `samplesales`, type this query below and click Run.

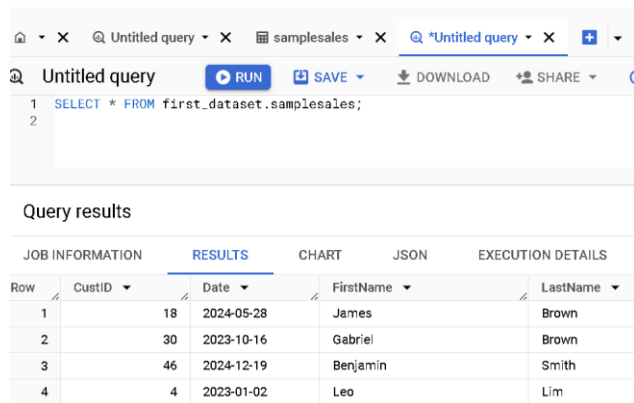
```
SELECT * FROM first_dataset.samplesales;
```

IMPORTANT NOTE: If you had created your own dataset earlier and created a table within it using the same CSV file as me and you wish to use that for this workshop instead, then remember to use your unique dataset name instead in all the remaining SELECT statements, for e.g.

```
SELECT * FROM peter_first_dataset.samplesales;
```

```
SELECT * FROM jane_first_dataset.samplesales;
```

The results should appear in the Query Results section.



The screenshot shows the BigQuery interface with a query editor and a results table. The query editor contains the following SQL:

```
1 SELECT * FROM first_dataset.samplesales;
2
```

Below the query editor, the 'Query results' section is active, displaying a table with 5 columns: Row, CustID, Date, FirstName, and LastName. The table contains 4 rows of data.

Row	CustID	Date	FirstName	LastName
1	18	2024-05-28	James	Brown
2	30	2023-10-16	Gabriel	Brown
3	46	2024-12-19	Benjamin	Smith
4	4	2023-01-02	Leo	Lim

You can open a few more new SQL query tabs in the same way as you did previously to type in the following new queries.

Most of the time when writing queries, we will only want to view values in specific columns, rather than all the columns in a table, since that may distract us from focusing on the specific values that we want to understand / study / analyze from the SQL retrieval operation:

To view only the CustID, FirstName and LastName column values from all rows in the table, type in a new SQL query tab and run:

```
SELECT CustID, FirstName, LastName FROM first_dataset.samplesales;
```

To view only the Price column values from all rows in the table, type in a new SQL query tab and run:

```
SELECT Price FROM first_dataset.samplesales;
```

You can specify the columns that you wish to view in any order you want independent of the order that they appear in the original table, for e.g. type in a new SQL query tab and run:

```
SELECT Units, ProdCategory, Region FROM first_dataset.samplesales;
```

Every query is executed through a job launched by BigQuery since the table that the query is executed on could be potentially very large (MB -> GB -> TB). When you are working with much larger tables, you can check on performance and job related issues in the Job information, Execution Details and Execution Graph tab.

--	--

Query results		Query results			
JOB INFORMATION	RESULTS	CHART	JSON	EXECUTION DETAILS	EXECUTION GRAPH
<p>For help debugging or optimizing your query, check our documentation. Learn more.</p>					
Elapsed time	189 ms	Slot time consumed	56 ms	Bytes shuffled	450 B
Job ID	first-bigquery-project-454101:US.bq				
User	Victor.Tan.33@gmail.com				
Location	US				
Creation time	Mar 18, 2025, 11:23:58 AM UTC+8				
Start time	Mar 18, 2025, 11:23:58 AM UTC+8				
End time	Mar 18, 2025, 11:23:59 AM UTC+8				

3.1 SELECT with expressions and aliases

The SELECT statement can be used to transform data retrieved from the table through expressions.

For e.g. the current total sale price is not available in the table, but we could calculate it using this formula:

total sale price = no. units sold x price per unit

To compute this value for all rows in the table, write and run the SQL statement below:

```
SELECT CustID, Price, Units, Price*Units
FROM first_dataset.samplesales;
```

Notice that the newly computed value for total sale price appears in a column with a temporary random name (such as f0_)

We can assign a meaningful column name for this newly computed value with a column alias using the keyword AS. Write and run the SQL expression below:

```
SELECT CustID, Price, Units, Price*Units AS Total_Price
FROM first_dataset.samplesales;
```

Besides the multiplication operator (*) you can use other mathematical operators such as addition (+), subtraction (-), and division (/).

If you perform a calculation on columns in the SELECT statement, these columns are often referred to as calculated columns.

You can also use expressions on string values, for e.g. if you wanted to view the customer first and last names as a single string in a single column (instead of in 2 separate columns as of now), you could use the || string concatenation operator on the values from both these columns. Write and run the SQL expression below:

```
SELECT CustID, FirstName || ' ' || LastName AS FullName
FROM first_dataset.samplesales;
```

Column aliases can also be used for any existing column, and not necessarily just calculated columns. This would be useful to rename existing columns to shorter or more meaningful names. Write and run the SQL expression below:

```
SELECT CustID AS CustomerID, ProdCategory AS Category
FROM first_dataset.samplesales;
```

3.2 SELECT with functions (String, Math and Date)

Google SQL has a [large number of functions](#). This serves the purpose of transforming, aggregating, and manipulating data during query execution that extends SQL beyond the basic data retrieval performed by a SELECT.

[String functions](#) are useful for working with columns in our table that contain alphanumeric data. For e.g. you may wish to list the FirstName column values so that they are all in lowercase and the LastName column values so that they are all in uppercase.

```
SELECT LOWER(FirstName) AS LowerFirstName,
UPPER(LastName) AS UpperLastName FROM first_dataset.samplesales;
```

You may wish to find the number of characters in the strings in the FirstName column.

```
SELECT FirstName, LENGTH(FirstName) AS LengthFirstName
FROM first_dataset.samplesales;
```

[Mathematical functions](#) are useful for working with columns in our table that contain numeric data.

For e.g. we have a Price column which contains numbers with fractional digits. For this, we have several possible functions:

[CEIL](#) – rounds up to the closest integer

[FLOOR](#) – rounds down to the closest integer

[TRUNC](#) – removes the fractional digit (has nearly the same effect as FLOOR)

[ROUND](#) – rounds to the nearest integer

To understand the difference between the effect of the 3 different functions (CEIL, FLOOR and ROUND), we use them on the Price column.

```
SELECT Price, CEIL(Price) as CeilingPrice, FLOOR(Price) as FloorPrice,
ROUND(Price) as RoundPrice FROM first_dataset.samplesales;
```

[Timestamp](#) functions are used to work with date and time values, an example of which is available in the Date column, which is from the [Date](#) data type. The Date data type represents a calendar date, independent of time zone, however timestamp functions work on [Timestamp](#) types, which include both date and time and optionally a time zone. We will work with Timestamp types in the exercise labs, but for now, we can use the EXTRACT timestamp function to determine the corresponding day, week, month and year portion corresponding to a given date.

```
SELECT Date, EXTRACT(YEAR FROM Date) AS year,
EXTRACT(MONTH FROM date) AS month,
EXTRACT (WEEK FROM Date) AS week,
EXTRACT(DAY FROM Date) AS day FROM first_dataset.samplesales;
```

The above are some very simple demonstration of functions. There are more complex functions in the different categories of functions which are particularly useful for specific uses cases, which we will see later in the exercises.

3.3 SELECT with DISTINCT and COUNT

Currently, when retrieving data from table, all values in a column for all rows are displayed, including duplicate values. For example, if two or more customers had the same last name, or are from the same regions, these values would have appeared multiple times.

As an example, write and run the SQL expressions below:

```
SELECT LastName FROM first_dataset.samplesales;
```

```
SELECT Region FROM first_dataset.samplesales;
```

Often its useful to know what are all the possible unique values for a column rather than viewing all the values: for e.g. how many unique regions or states there are in the table. To eliminate duplicate values and obtain only unique entries, we use SELECT DISTINCT which removes duplicate rows from the result set returned and only retains one row for each group of duplicate values.

The SELECT DISTINCT clause can be applied to one or more columns in the select list of the SELECT statement.

Repeat the previous SQL expressions that you used earlier but this time with the SELECT DISTINCT keyword:

```
SELECT DISTINCT LastName FROM first_dataset.samplesales;
```

```
SELECT DISTINCT Region FROM first_dataset.samplesales;
```

If you specify multiple columns, the SELECT DISTINCT clause will evaluate unique values based on the combination of values in these columns. For example:

```
SELECT DISTINCT Region, State FROM first_dataset.samplesales;
```

Another common use case is where you just want to know how many unique values there are, instead of actually viewing all the unique values possible. For this purpose, we can use the COUNT aggregate function on the column of interest. This function counts the number of non-null values in the specified column. For example:

```
SELECT COUNT(DISTINCT Region) AS Total_Regions  
FROM first_dataset.samplesales;
```

There are also other aggregate functions such as SUM, AVG, MIN and MAX that work on a collection of values, which we will see later.

Another common use case for COUNT is just to count the total number of rows/records in the table:

```
SELECT COUNT(*) AS TotalRows  
FROM first_dataset.samplesales;
```

3.4 SELECT with LIMIT

So far, the SELECT statement query returns all the rows in the table. For large tables, often you only want to see a subset of the all the rows that returned by the query. The LIMIT statement used in this instance to constrain the number of rows returned from the query. This helps optimizing queries thus improving performance.

The simplest form for its usage:

```
SELECT
    select_list
FROM
    table_name
LIMIT
    row_count;
```

The simplest example of its usage here:

```
SELECT CustID, FirstName, LastName FROM first_dataset.samplesales
LIMIT 10;
```

This simply returns the first 10 rows/records from the query (which would normally return all 50 rows records).

Typically, we will combine the LIMIT clause with other clauses such as ORDER BY and WHERE to get the top ranking (or bottom ranking) records with a specific value in a column that fulfil certain conditions, as we will see later.

4 Using Gemini in BigQuery

[Gemini in BigQuery](#) offers AI-powered assistance for data analytics, in particular working with SQL queries, and needs 2 steps to be enabled. This is already performed for this particular project and in your company, this should be separately enabled for you by your admin:

- a) Enable necessary APIs and grant roles.
- b) Turn on Gemini in BigQuery features in the Google Cloud console.

[Gemini Cloud Assist](#) is also available to help with working with SQL queries, although it is primarily targeted to offer AI-powered assistance to help design, deploy, troubleshoot, and optimize apps that are tailored to help achieve specific efficiency, cost, reliability, and security goals.

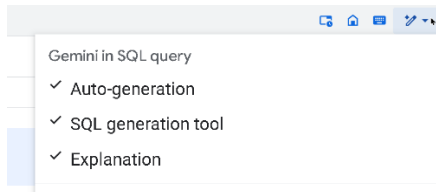
This also needs to [enabled before it](#) can be used. This is already performed for this particular project and in your company, this should be separately enabled for you by your admin.

When [using Gemini to assist writing queries](#), there are variety of features available:

- a) Generate a SQL query based on an existing table schema using a [prompt or an existing comment directly within the SQL editor table](#) (the most common use)
- b) Generate a SQL query based on an existing table schema [using a prompt in Gemini Code Assist](#) in the Cloud Assist side panel.

- c) Explain the [functionality of an existing query](#). This is useful for complex queries created by other members of your team that you cannot understand, or that you created yourself in the past but cannot understand clearly now.

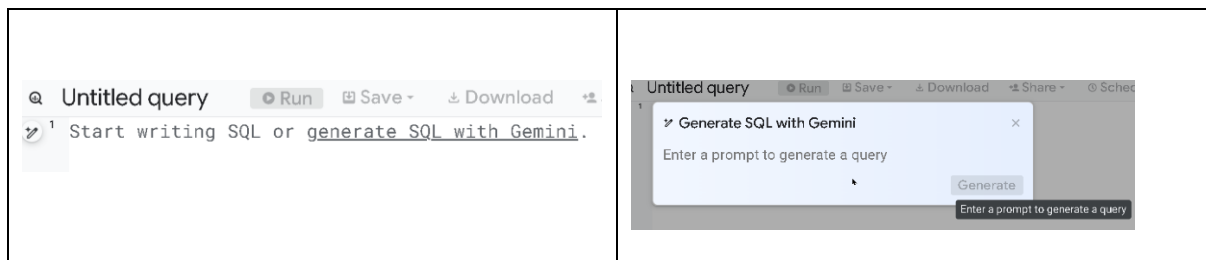
You should be able to see these features listed with a tick next to them (indicating them being enabled) in the drop down list when you click on the Gemini tool icon in the upper right hand corner.



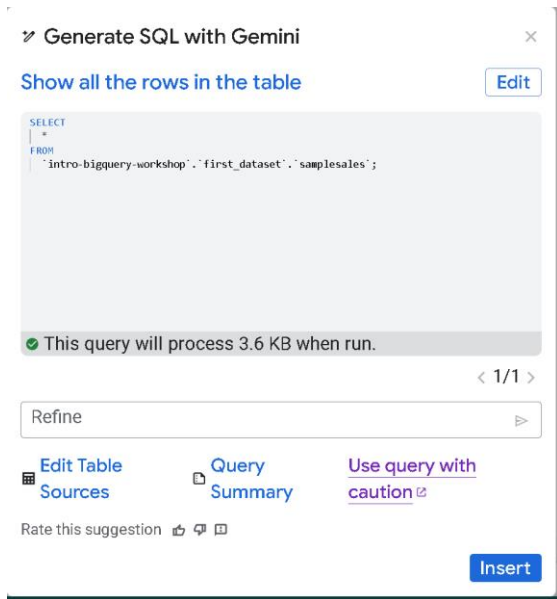
There are some [specific guidelines for prompts for queries](#) as well [general purpose guidelines](#) for writing prompts in Gemini that maximize the changes that your prompts successfully accomplish the specific purpose or functionality that they are intended to achieve.

4.1 Working with prompts directly in the SQL editor tab

This is the most common way you will work with prompting techniques. In a new query tab, click on Generate SQL with Gemini to open the prompt box to type in your prompt:



Type in a simple prompt: Show all the rows from the samplesales table and click Generate. You should get a response similar to the one below



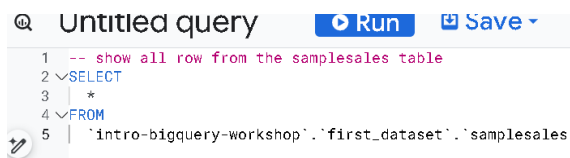
At this point, you have several options:

Edit Table Sources – this enables you to select the particular dataset and table that this query is to be executed on (the FROM portion of the query), in the event the selected dataset/table is incorrect. At the moment, you only have one active dataset / table and also you have explicitly named the table, so there should not be a problem here.

Notice there that the query includes the fully qualified name of the table, which is *project-name.data-set-name.table-name*.

Query Summary – Summarizes what the SQL query does in plain language. You can click this to see the explanation in the dialog box.

Click **Insert** to place the suggested SQL query into the editor table at which point you can run it and view the results. You will notice the prompt appears as a comment above the query, which is helpful to remind you that the query was probably generated from a prompt rather than human generated.



Let's try another prompt in new query tab:

Show only the Units, ProdCategory and Region columns from samplesales

You should get the same query that we had used earlier, and you can click to run as usual.

4.2 Refining prompts in the Gemini dialog box

We can further transform / refine a query returned from an initial prompt by several follow up prompts.

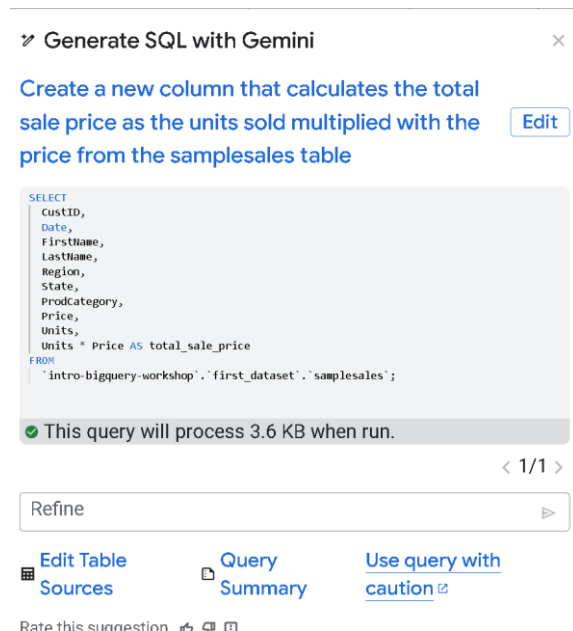
To demonstrate this, let's repeat an earlier use case where we wanted to calculate the current total sale price using this formula:

total sale price = no. units sold x price per unit

Let's try another prompt to do this in a new query tab:

Create a new column that calculates the total sale price as the units sold multiplied with the price from the samplesales table

You should get a query that looks similar to the one below:



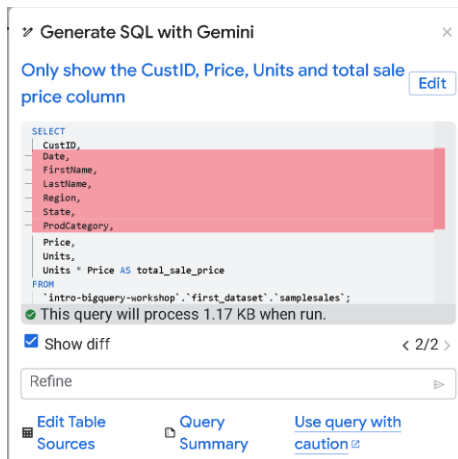
One of the key features with queries generated from most prompts are that they will include / specify all columns in the generated query.

At this point, instead of executing the suggested query to check the result, if you are already quite familiar with GoogleSQL, you can straight away perform query refinement by typing further prompts to finetune the query in the Refine box below and clicking the arrow on refine.

Refine:

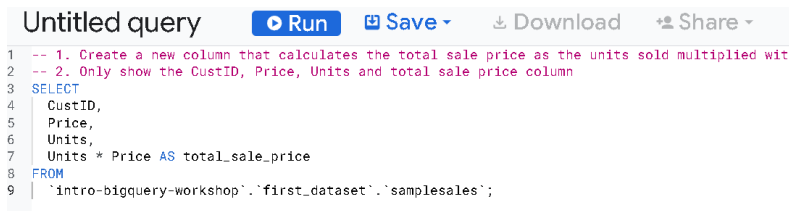
Only show the CustID, Price, Units and total sale price column

This will generate a new query and highlight the part of the original query that it will now change to obtain the latest query version (this is known as a diff). Right now, the area highlighted in red is the part that has being removed in the latest query iteration.



You can go ahead and click Insert and run as usual.

Both your original prompt and refinement prompt will show up as comments in the editor tab.



4.3 Refining existing queries through further prompts in the Editor tab

The previous approach of typing further refining prompts directly in the Gemini dialog box works well if you are already quite familiar with GoogleSQL and you can evaluate that the query from the original prompt will not produce the result you are looking for and hence you can immediately proceed to refine it.

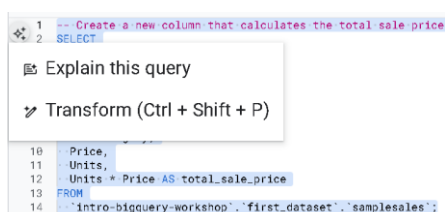
A more realistic approach (particularly when you are new to GoogleSQL) is to run the suggested query from the first prompt, check on the results generated, and then refine that query using further prompts.

To demonstrate this, open a new query tab and repeat the previous original prompt:

Create a new column that calculates the total sale price as the units sold multiplied with the price from the samplesales table

This time, immediately copy and paste the query result in the Editor pane without any further refining prompts. Then run it as usual to view the results.

Then highlight the entire query and click on Gemini icon in the numbers pane on the left to bring up the menu which provides two options (Explain this query and Transform).



Click on transform and repeat the previous refinement prompt:
Only show the CustID, Price, Units and total sale price column

This again brings up a very similar or identical query result box, for which you can then click Insert and run as usual.

With this approach, you get the chance of executing the query from the first prompt and seeing the results it produces, and then refining that query using a follow up prompt if the results are not what you are expecting or looking for.

4.4 Detailed prompts vs generic prompts and prompt refinement

The idea of prompt refinement is to provide multiple consecutive follow-up prompts to refine the result obtained from the initial query as just demonstrated. This is useful especially when you are unclear about the result likely to be produced from the first query.

However, if you are familiar with how GoogleSQL and Gemini AI prompt system works, you can type a more detailed prompt in a new Editor tab which specifies in greater detail the exact result you expect for e.g.:

```
Generate a query that creates a new column called OverallPrice that calculates the total sale price as the units sold multiplied with the price from the samplesales table. The query should only show the CustID, Price, Units and the OverllPrice column.
```

which provides you the query you are looking for in the first prompt response.

The better you get at writing **very detailed and very specific prompts**, the faster you will arrive at the query that gives you the exact results you are looking for.

This is the **key principle in effective prompting** and will take some time and practice to master adequately.

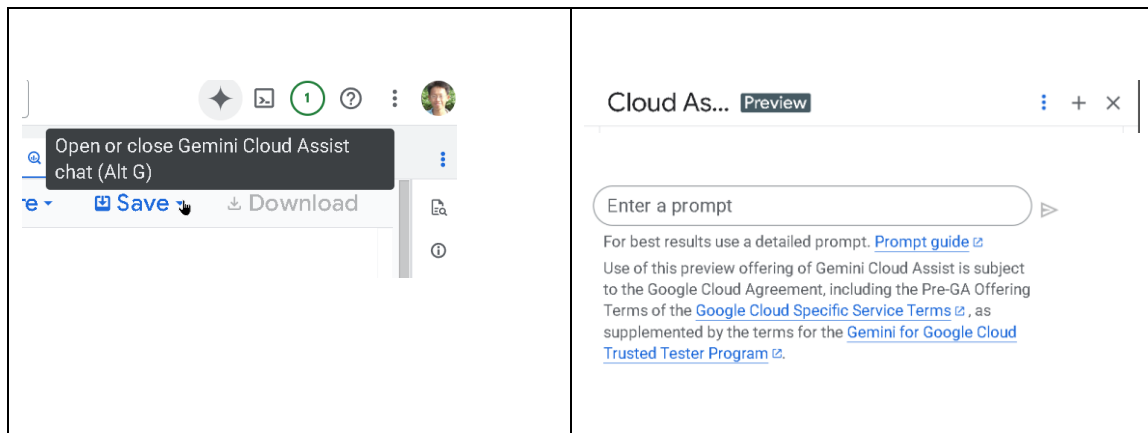
On the flip side of detailed, explicit prompts are excessively generic or universal prompts such as:

```
show me important rows that my boss needs to look at from the table
```

Here the term `important rows that my boss needs to look at` is so vague and has so many possible universal meanings and interpretations that whatever query that is returned will vary widely between different AI tools and even between different requests on the same AI tool. Try running this prompt multiple times on several SQL editor tabs to verify this.

4.5 Using Gemini Cloud Assist

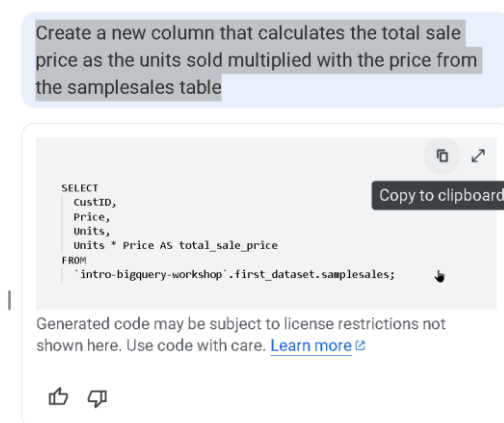
Another very similar alternative to typing prompts directly in the Query editor tab is to use the Gemini Cloud assist to type prompts. Click on Gemini Cloud Assist icon to bring this up.



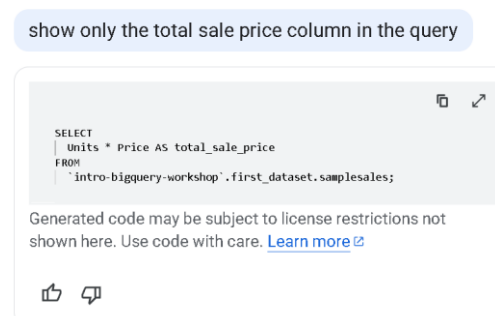
You can repeat the prompt that you used earlier just now

Create a new column that calculates the total sale price as the units sold multiplied with the price from the samplesales table

And you can copy the result to the clipboard and paste in the query editor.



Alternatively, you can continue to type further prompts to refine the query result returned, for e.g. show only the total sale price column in the query



whereby you can copy to the clipboard and paste again in your SQL shell.

In general, although we can use Gemini Cloud Assist to help with working with SQL queries, it is more of a general purpose AI tool that offers assistance to help design, deploy, troubleshoot, and optimize apps. Hence, it is best to stick with Gemini AI assistance embedded directly in BigQuery.

4.6 Replicating previous queries

We will try to replicate the previous queries that we demonstrated in the previous lab session through prompts.

Keep in mind that the SQL queries generated from your prompts may differ slightly between each other and between what is shown in this manual as the nature of LLMs is to give slightly different responses to the same prompt each time, but which essentially provide the same final desired result.

To speed up this lab session, you can compare the prompt results you obtain with your original queries to verify that they are nearly identical, and if you wish, you can further run them to validate that you get the same query result or else just simply discard the prompt result.

Useful tip: If you have already been running many queries and/or prompts related to a particular table, Gemini will keep track of this and you will no longer need to explicitly specify that table name in your prompt as you did earlier. If Gemini picks the wrong table, you can click on Edit Table Sources and pick the right one.

The prompt related to string functions:

```
Show the FirstName in lowercase and LastName in uppercase from samplesales
```

Notice here that the generated prompt uses the original column name as the alias column name after the AS clause:

```
SELECT LOWER(FirstName) AS LowerFirstName, UPPER(LastName) AS UpperLastName FROM first_dataset.samplesales;
```

Another prompt related to string functions:

```
Show the number of characters in the strings in the FirstName column
```

The prompt related to math functions:

```
Round up the numbers in the Price column to the closest integer value
```

The prompts related to DISTINCT clause:

```
Show all the unique or distinct values in LastName
```

```
How many unique or distinct regions are there ?
```

The prompt related to LIMIT clause:

```
Show the first 10 rows in the table
```

4.7 Using other AI tools to generate the query (Optional)

There are many AI tools that can generate queries in response to prompts in a similar manner to Gemini in BigQuery. Well known ones include ChatGPT, Copilot, Claude, Meta AI, Grok and DeepSeek.

Using Gemini in BigQuery has the advantage that the table schema is automatically picked up and interpreted in the query itself. If you wish to execute a prompt in another tool, you will have to **explicitly specify the table schema** that you want to generate the query for.

Selecting the table in the Explorer pane should show its Schema in the details pane, where you can see the [data types](#) that BigQuery has assigned to each of the fields. You will then need to incorporate this schema description into your prompt.

An example of a complete prompt to generate a query to count the number of unique / distinct regions:

```
I have created a table called samplesales in a dataset called first_dataset in Google BigQuery with the following schema and data types:
```

```
CustID: INTEGER
Date: DATE
FirstName: STRING
LastName: STRING
Region: STRING
State: STRING
ProdCategory: STRING
Price: FLOAT
Units: INTEGER
```

```
Create a GoogleSQL query that counts the number of unique or distinct values in the REGION column.
```

4.8 Validating prompt results (query correctness)

If you are writing extremely detailed prompts that produce very complex queries (which we will do later in some of the lab exercises), you may be uncertain about whether the query returns the exact result that you are looking for. The prompt will always return a valid query, so the query will always execute on the table without any error, but it may not necessarily give you the actual result you are looking for.

Remember: The more **detailed and specific your prompt is**, the more likely it is to generate query that gives you the exact result you are looking for.

Some practical beginner-level solutions that you can use to validate the correctness of the query:

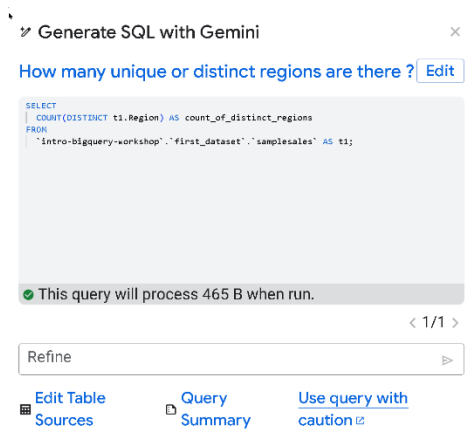
4.8.1 Obtain detailed plain-language description of what a query does

We can use Gemini in BigQuery to explain what the query does in great detail in plain language and validate whether this is what you actually want it to do based on the description alone.

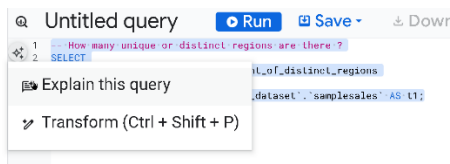
For e.g. for the prompt

```
How many unique or distinct regions are there ?
```

You can click on the Query Summary link in the Dialog box.



Alternatively, you can paste the query into the Editor Pane and use the Explain this Query option, which will pass this query to Gemini Cloud Assist to provide a more detailed explanation.



You can also type a prompt like this involving the generated query in Gemini Cloud Assist chat panel

Explain step by step what the query below does for each row and each column in the table samplesales
Sample query

For e.g.

Explain step by step what the query below does for each row and each column in the table samplesales
 SELECT COUNT(DISTINCT Region) AS Total_Regions
 FROM first_dataset.samplesales;

All these 3 different approaches provide different ways of explaining the same query. Use the approach that provides the explanation that is easiest for you to understand and verify whether it corresponds to what you really want to achieve.

4.8.2 Run query against a sample truth table

A truth table is a small table with dummy data (or a sampling of random rows with real life data from the original real life table) where you can manually determine what should be the correct result returned from your query. Then you can simply run the suggested query returned from the prompt and determine whether it provides back the exact result you determined manually. If so,

For e.g. you can manually count the number of distinct regions in this table from the column and compare it with the result returned from the previous query.

The problem with this approach is that your table with dummy data may not necessarily contain all combinations / permutations of data values possible to test your query. This means that the query may seem to work when you run it against a small table with dummy data consisting of 50 rows, but

when you run in on a real-life table with real life data consisting of 1 million rows, you might not get back exactly what you were hoping to accomplish.

4.8.3 Compare against prompt results from other gen AI tools

Run a similar prompt on another popular generative AI tool (ChatGPT, Grok, Copilot, etc) and compare the query that you get back. Different gen AI tools may produce different query results for the same prompt, and if all of them produce identical queries or nearly the same query for the same prompt, there is a good chance that this query is the correct one.

4.8.4 Validate with a human expert

This is the best and most reliable approach. Validate the correctness of your SQL query with an SQL expert who has domain expertise and vast experience in running queries against the kind of sample datasets that you work with.

5 Sorting rows with ORDER BY

The rows returned from the query are in an unspecified order. Often, particularly when the query result size is large, we want the rows returned to be sorted or ordered based on the value in one or more columns. We can use the ORDER BY clause for this purpose which has the general form:

```
SELECT
    select_list
FROM
    table_name
ORDER BY
    sort_expression1 [ASC | DESC],
    sort_expression2 [ASC | DESC],
    ...;
```

First, specify a sort expression, which can be a column or an expression, that you want to sort after the ORDER BY keyword. If you want to sort the result set based on multiple columns or expressions, you need to place a comma (,) between two columns or expressions to separate them.

Second, you use the ASC option to sort rows in ascending order and the DESC option to sort rows in descending order. If you omit the ASC or DESC option, the ORDER BY uses ASC by default.

To sort the rows returned based on ascending alphabetical order of the first name:

```
SELECT * FROM first_dataset.samplesales ORDER BY FirstName;
```

To sort the rows returned based on descending order of the price:

```
SELECT ProdCategory, Price FROM first_dataset.samplesales
ORDER BY price DESC;
```

WE can also sort on the values in 2 or more columns by specifying them in the ORDER BY.

For e.g. we can do a primary sort on the last name in ascending order, and if there are two or more rows with identical last names, then we perform a secondary sort in descending order

```
SELECT * FROM first_dataset.samplesales
ORDER BY LastName ASC, FirstName DESC;
```

Note: The default sort is ascending order, so we could have left out the ASC keyword in the statement above. Also the sort order is just randomly specified as ascending or descending for this example, we could have used any particular order for the first name / last name depending on our specific use case.

We can also use expressions and aliases for the columns that we wish to sort on. For example, we had earlier computed the total sale price for all rows using this query:

```
SELECT CustID, Price, Units, Price*Units AS Total_Price
FROM first_dataset.samplesales;
```

Now we can choose to sort on descending order to this total sale price with:

```
SELECT CustID, Price, Units, Price*Units AS Total_Price
FROM first_dataset.samplesales
ORDER BY Total_Price DESC;
```

If we have a table column with date/time related information, we can also choose to sort on that to find the latest / earliest transactions (depending on our sort order):

```
SELECT * FROM first_dataset.samplesales
ORDER BY Date;
```

Note that BigQuery [date type](#) is in the format YYYY-MM-DD

You can also combine ORDER BY with LIMIT in which case the command now that the format similar to the following:

```
SELECT
    select_list
FROM
    table_name
ORDER BY
    sort_expression
LIMIT
    row_count;
```

This is very useful when we want to find the top ranking (or bottom ranking) records with a specific value in a column.

For e.g. to find the top 5 transactions by quantity of units purchased we would use a query like this:

```
SELECT ProdCategory, Units
FROM first_dataset.samplesales
ORDER BY Units DESC
LIMIT 5;
```

To find the 10 lowest transactions in terms of total sale price, we would use a query like this:

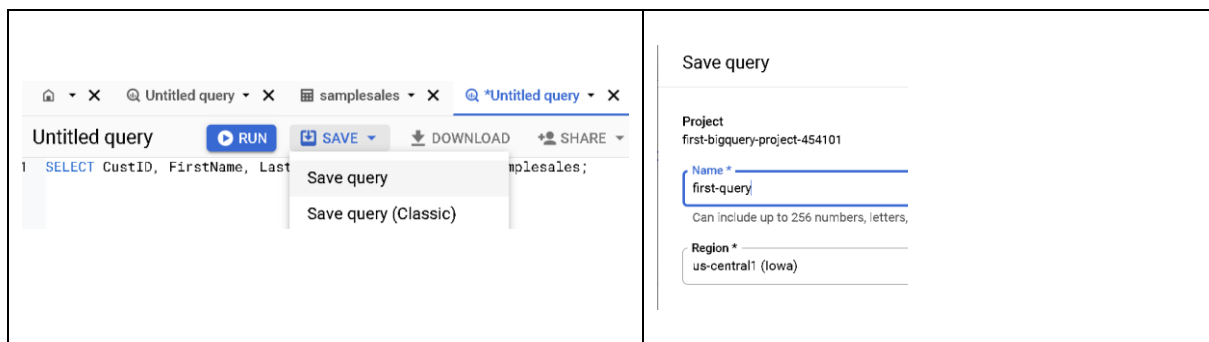
```
SELECT CustID, Price, Units, Price*Units AS Total_Price
FROM first_dataset.samplesales
```

```
ORDER BY Total_Price
LIMIT 10;
```

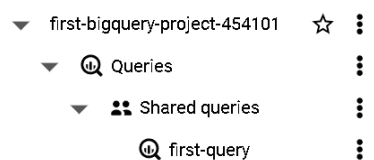
6 Saving queries, query results and viewing query history

At this point, you have a lot of untitled queries which are unsaved. You may wish to save any (or all) of them so that you can retrieve them and execute them on the table in the event that modifications have being made to the table (new rows added, existing rows modified or deleted) since the last time the query was run and you wish to see the results of the same query on the new table contents.

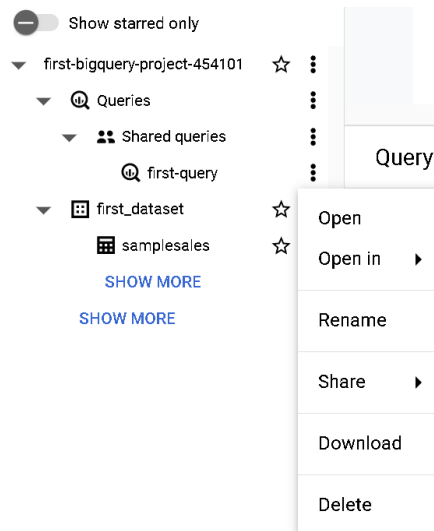
To save a query, ensure the tab for that query is active in the query editor pane, click on Save and give it a suitable name, for e.g.: `first-query`. You can accept whatever default region is suggested as the location to save this query to.



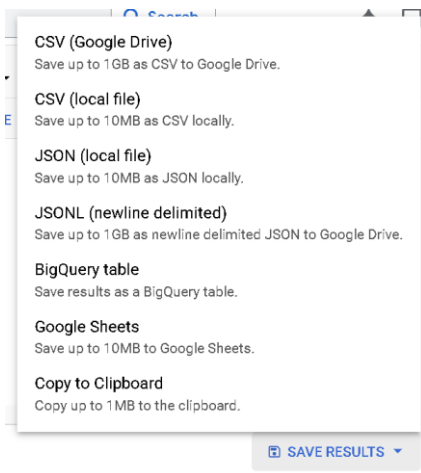
Once it is saved, you should be able to view it in the Queries section of the project in the Explorer pane. You can then close the Query tab and double click on this entry in the Explorer Pane to access it again.



There are a variety of other options available for Query accessible from the ellipsis menu icon, such as renaming, sharing with other valid users on the GCP, downloading or deleting.



In addition to the query itself, you can save the results returned from the query in a variety of forms easily from the Save Results menu in the upper right corner of the pane for the query results. This include downloading the result, saving the results to a CSV or Google Sheet in your Google Drive account and even creating a new table to store the results.









You can check on the history of your queries (listed as Job History, since queries are executed as jobs in BigQuery) in the lower bottom pane area of the main editor pane. This area can be maximized or minimized accordingly to facilitate access.

Job history

PERSONAL HISTORYPROJECT HISTORY

Filter Enter property name or value

Job ID	Creation time	Owner	Type	Summary
 bquxjob_6af7c08c_195a7c35690	March 18, 2025 1:39 PM	Victor.Ta...	QUERY	SELECT COUNT(DISTINCT Region) AS Tot...
 bquxjob_62fe8fd2_195a7ba8d6f	March 18, 2025 1:29 PM	Victor.Ta...	QUERY	SELECT * FROM first_dataset.samplesales;...
 bquxjob_45a1702d_195a7b87cb6	March 18, 2025 1:27 PM	Victor.Ta...	QUERY	SELECT CustID, Price, Units, Price*Units FR...
 bquxjob_7170f9a4_195a7abbad8	March 18, 2025 1:13 PM	Victor.Ta...	QUERY	SELECT DISTINCT Region FROM first_data...
 bquxjob_2b836e04_195a7ab818d	March 18, 2025 1:13 PM	Victor.Ta...	QUERY	SELECT DISTINCT LastName FROM first_d...
 bquxjob_3445d6f5_195a7a9f4e6	March 18, 2025 1:11 PM	Victor.Ta...	QUERY	SELECT Region FROM first_dataset.sample...

7 Filtering with WHERE

An extremely common use case is to retrieve rows from a very large table that meet one or more specific conditions (rather than every single row). The WHERE clause is used for this purpose and has the general form:

```
SELECT
    select_list
FROM
    table_name
WHERE
    condition;
```

The `condition` is an expression that evaluates to the [Boolean type](#) (true, false, or Null) and is used to filter the rows returned from the query.

To form the condition in the WHERE clause, you use comparison and logical operators:

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<> or !=	Not equal
AND	Logical operator AND
OR	Logical operator OR
IN	Return true if a value matches any value in a list
BETWEEN	Return true if a value is between a range of values
LIKE	Return true if a value matches a pattern
IS NULL	Return true if a value is NULL
NOT	Negate the result of other operators

To identify all the customers whose last name is Lim, we use this query:

```
SELECT CustID, FirstName, LastName
FROM first_dataset.samplesales
WHERE LastName = 'Lim';
```

If we are only interested in counting the total number of customers with the last name of Lim (rather than examining the first names for each of these customers), we can use COUNT to summarize – the name of the field to apply COUNT to is irrelevant.

```
SELECT COUNT(*) AS TotalLims
FROM first_dataset.samplesales
WHERE LastName = 'Lim';
```

To identify all the sales that have being made by customers in Kedah, we use this query:


```
SELECT CustID, FirstName, LastName, State
FROM first_dataset.samplesales
WHERE State = 'Kedah';
```

Again, if we are only interested in counting the number of sales made in Kedah (rather than examining specific individual field values for each of these transactions), we can use COUNT to summarize:

```
SELECT COUNT(CustID) AS TotalKedahSales
FROM first_dataset.samplesales
WHERE State = 'Kedah';
```

To identify all sales where the number of units purchased are more than 20.

```
SELECT CustID, FirstName, LastName, Units
FROM first_dataset.samplesales
WHERE Units > 20;
```

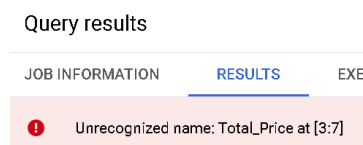
To identify all sales where the total sale price is less than 100.

```
SELECT CustID, Price, Units, Price*Units AS Total_Price
FROM first_dataset.samplesales
WHERE Price*Units < 100;
```

If we had attempted to write the statement above in this form below:

```
SELECT CustID, Price, Units, Price*Units AS Total_Price
FROM first_dataset.samplesales
WHERE Total_Price < 100;
```

We would obtain an error as follows:



This is because the order of evaluation is: **FROM -> WHERE -> SELECT**

The Total_Price is only used in the SELECT, so it cannot be referenced in the WHERE, which can only reference the actual columns in the table referenced by FROM (in this case Price and Units)

This illustrates a subtle but important issue about understanding order of evaluation of keywords when writing complex SQL queries.

To identify sales in all regions **EXCEPT** the Southern region:

```
SELECT CustID, Region, Price, Units
FROM first_dataset.samplesales
WHERE Region <> 'Southern';
```

You can also include the ORDER BY and LIMIT clauses as well if you wish in which case it has the more complex form of:

```

SELECT
    select_list
FROM
    table name
WHERE
    condition
ORDER BY
    sort_expression
LIMIT
    row_count;

```

In a complex query like this, the order of evaluation is: **FROM -> WHERE -> SELECT -> ORDER BY -> LIMIT**

An example might be to find the top 5 sales transactions from the Northern region of Malaysia based on the number of units purchased:

```

SELECT CustID, Region, Units
FROM first_dataset.samplesales
WHERE Region = 'Northern'
ORDER BY Units DESC
LIMIT 5;

```

7.1 Using the AND, OR and NOT operators

We use the AND and OR operators to combine the Boolean values returned from two (or more conditional expressions). The results of the evaluation using these operators is shown in the truth table:

X	Y	X and Y	X or Y	not(X)	not(Y)
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

To find all the sales transactions that fulfil BOTH of these 2 conditions below:

- Sales were in the state of Kedah
- Sales were for products whose price was above 20

```

SELECT CustID, State, Price
FROM first_dataset.samplesales
WHERE State = 'Kedah'
AND Price > 20;

```

To find all the sales transactions that fulfil EITHER of these 2 conditions below:

- Sales for products categorized as Computers
- Sales to customers with last name of Lim

```
SELECT CustID, ProdCategory, LastName
FROM first_dataset.samplesales
WHERE ProdCategory = 'Computers'
OR LastName = 'Lim';
```

To identify sales in all regions **EXCEPT** the Southern region:

```
SELECT CustID, Region, Price, Units
FROM first_dataset.samplesales
WHERE NOT Region = 'Southern';
```

Note that this is functionally identical to a previous query we used before:

```
SELECT CustID, Region, Price, Units
FROM first_dataset.samplesales
WHERE Region <> 'Southern';
```

7.2 Using BETWEEN for range tests

Sometimes we want to test that a value falls between a certain range. This can be done using the AND operator but can be simplified using BETWEEN.

To find all sales whose price is between 20 and 40, we could write:

```
SELECT CustID, Price
FROM first_dataset.samplesales
WHERE Price >= 20
AND Price <= 40;
```

The equivalent shorter form using BETWEEN is:

```
SELECT CustID, Price
FROM first_dataset.samplesales
WHERE Price BETWEEN 20 AND 40;
```

To find all sales whose price is either below 20 or above 40, we could write:

```
SELECT CustID, Price
FROM first_dataset.samplesales
WHERE Price < 20
OR Price > 40;
```

Notice that if you had used an AND in the above statement instead of an OR you would not have obtained any results at all since the condition will never be true. This illustrates the importance of ensuring you understand clearly the logic when combining multiple conditions using these 2 operators.

We can accomplish identical query logic using BETWEEN by noting that the condition for this latest query is simply an inversion of that for the initial query and repeat the initial BETWEEN query with the inversion operator NOT.

```
SELECT CustID, Price
FROM first_dataset.samplesales
WHERE Price NOT BETWEEN 20 AND 40;
```

BETWEEN can also be used for date ranges. To find all sales within the year 2023, alone we can use this statement:

```
SELECT CustID, DATE
FROM first_dataset.samplesales
WHERE DATE BETWEEN '2023-01-01' AND '2024-01-01';
```

7.3 Using IN to check for matching with other values

The IN operator allows you to check whether a value matches any value in a given list of values.

To get list of all sales in Sabah, Terengganu and Penang, we could use a statement with OR to combine reference to all these values:

```
SELECT CustID, State
FROM first_dataset.samplesales
WHERE State = 'Sabah'
OR State = 'Terengganu'
OR State = 'Penang';
```

Alternatively, we can use the IN operator to simplify the statement and accomplish the same logic with:

```
SELECT CustID, State
FROM first_dataset.samplesales
WHERE State IN ('Sabah', 'Terengganu', 'Penang');
```

We can also use IN with a list of integer numbers and dates, for e.g:

```
SELECT CustID, Units
FROM first_dataset.samplesales
WHERE Units IN (10, 20, 25, 38);
```

Notice that it is perfectly fine if there is no matching row for any one or more items in the list (for e.g. 38); as long as there is a row whose designated column value matches any one of the items in the list, it is returned in the query result.

```
SELECT CustID, Date
FROM first_dataset.samplesales
WHERE Date IN ('2022-10-25', '2023-10-15', '2024-6-6');
```

Note that BigQuery [date type](#) is in the format YYYY-MM-DD, so the strings should be in this format too as well.

7.4 Using LIKE to match string patterns

This is useful for when you are able to identify specific string patterns to be matched in a field. For e.g. a name that contains a particular sequence of characters or that starts or ends with a particular sequence of characters.

There are two common wildcard characters that are used with LIKE:

- Percent sign (%) matches any sequence of zero or more characters.
- Underscore sign (_) matches any single character.

To select all last names which contain the letter I in them, regardless of whether its at the start, in the middle or at the end, we can use:

```
SELECT LastName
FROM first_dataset.samplesales
WHERE LastName LIKE '%i%';
```

To select all last names which contain the letter I in them at exactly the 3rd position (and not anywhere else), we can use:

```
SELECT LastName
FROM first_dataset.samplesales
WHERE LastName LIKE '__i%';
```

8 Using CASE to implement conditional logic to add columns

The CASE expression allows implementation of conditional logic within the query itself. It is conceptually similar to if-else statement in other programming languages.

The general syntax

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  WHEN condition3 THEN result3
  ...
  ELSE final_result
END
```

Each condition (`condition1`, `condition2`, `condition3` ...) is a Boolean expression that returns true or false. The CASE expression evaluates conditions in sequence from top to bottom.

If a condition is true, the CASE expression returns the corresponding result in that follow the THEN and stops evaluating subsequent conditions.

If none of the listed conditions match, then the `final_result` is returned.

The typical use case for the CASE expression is to add an additional column to the original table which we can subsequently operate on by sorting, grouping or executing an aggregate function either directly or via subqueries and / or materialized views (to be covered in a subsequent lab topic)

Let assume we want to provide an additional column to further categorize the number of units for an order based on the table below.

Units purchased	Category
1 – 10	Low
11 – 20	Medium
21 - 25	High
26 and above	Awesome

The statement to achieve this would be:

```
SELECT CustID, Date, Price, Units,
CASE
    WHEN Units >=1 AND Units <=10 THEN 'Low'
    WHEN Units >= 11 AND Units <= 20 THEN 'Medium'
    WHEN Units >= 21 AND Units <= 25 THEN 'High'
    ELSE 'Awesome'
END AS Category
FROM first_dataset.samplesales;
```

We might have some custom classification logic to perform which is a bit more complex than the first example, which can easily be accomplished in the same way through appropriate conditional expressions in the CASE expression.

For e.g. we might identify that certain products in a product category that is sold in specific regions should have certain numeric ranking associated with them:

```
SELECT CustID, Region, ProdCategory,
CASE
    WHEN Region = 'East' AND ProdCategory = 'Computers' THEN 1
    WHEN Region = 'Northern' AND ProdCategory = 'Books' THEN 2
    WHEN Region = 'Southern' AND ProdCategory = 'Electronics' THEN 3
    WHEN Region = 'EastMalaysia' AND ProdCategory = 'Books' THEN 4
    ELSE 5
END Ranking
FROM first_dataset.samplesales;
```

Since we have assigned a numeric ranking to all the products, we can then sort on this ranking if we wish by just a minor extension to the previous statement:

```
SELECT CustID, Region, ProdCategory,
CASE
    WHEN Region = 'East' AND ProdCategory = 'Computers' THEN 1
    WHEN Region = 'Northern' AND ProdCategory = 'Books' THEN 2
    WHEN Region = 'Southern' AND ProdCategory = 'Electronics' THEN 3
    WHEN Region = 'EastMalaysia' AND ProdCategory = 'Books' THEN 4
    ELSE 5
END Ranking
FROM first_dataset.samplesales
ORDER BY Ranking;
```

9 Aggregate functions: COUNT, SUM, AVG, MIN, MAX

We have seen some earlier examples of the use of the COUNT aggregate command – to simply count the number of rows returned by a specific query involving other keywords such as WHERE, BETWEEN, DISTINCT, etc.

We can use the other aggregate commands in a similar manner.

For e.g. to find the total sum, average, largest and smallest value in the Units column we can use the following queries:

```
SELECT SUM(Units) AS TOTAL_UNITS
FROM first_dataset.samplesales;
```

```
SELECT AVG(Units) AS AVG_UNITS
FROM first_dataset.samplesales;
```

```
SELECT MAX(Units) AS LARGEST_UNIT
FROM first_dataset.samplesales;
```

```
SELECT MIN(Units) AS SMALLEST_UNIT
FROM first_dataset.samplesales;
```

Just like the case of COUNT, we can use any of these other aggregation commands in conjunction with the set of rows returned from query using WHERE to filter on the original table content.

For e.g. to find the average number of units sold in the Northern region, we can type:

```
SELECT AVG(Units) AS AVG_UNITS
FROM first_dataset.samplesales
WHERE Region = 'Northern';
```

To find the order with the highest price in Kedah, we can type:

```
SELECT Max(Price) AS HighestPrice
FROM first_dataset.samplesales
WHERE State = 'Kedah';
```

To find the order with the lowest number of units in the year 2024, we can type:

```
SELECT Min(Units) AS SmallestOrder
FROM first_dataset.samplesales
WHERE DATE BETWEEN '2024-01-01' AND '2025-01-01';
```

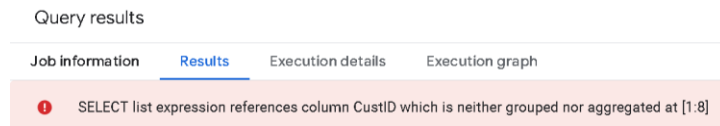
9.1 Refining queries with aggregate functions for column details

Occasionally, we may want additional details about the particular order that has the highest price or lowest number of units in the previous two queries. For e.g. we might want to know the first name, last name for the customer or date for these two specific orders.

We cannot directly use these fields in the SELECT statement and mix them with the aggregate command as it violates the SQL syntax rules, for e.g:

```
SELECT CustID, FirstName, LastName, Max(Price) AS HighestPrice
FROM first_dataset.samplesales
WHERE State = 'Kedah';
```

This query above results in an error.



To work around this issue, we can just simply use the result returned from the initial query involving the aggregation command in a subsequent query using WHERE to locate the row containing that particular value.

For e.g. if the previous query to find the order with the highest price in Kedah returns 59.5, we can subsequently use this in a follow up statement:

```
SELECT * FROM first_dataset.samplesales
WHERE State = 'Kedah'
AND Price = 59.5;
```

to obtain all the values for all the fields for this particular record.

We can also obtain the same effect directly using a subquery (or nested query) as below:

```
SELECT * FROM first_dataset.samplesales
WHERE State = 'Kedah'
AND Price = (
    SELECT Max(Price) FROM first_dataset.samplesales
    WHERE State = 'Kedah'
);
```

We will see more examples of subqueries in a subsequent lab topic.

Another way to confirm the correctness of the result obtained from queries performed using aggregation commands such as MIN or MAX is to sort the results (either in descending order or ascending order) based on the same WHERE filter used with the MIN or MAX command.

For e.g. we can list all the orders in Kedah and sort them in descending order based on price:

```
SELECT * FROM first_dataset.samplesales
WHERE State = 'Kedah'
ORDER BY Price DESC;
```

And we can confirm that the record at the top of the results returned has the price of 59.5

Similarly this query returns the orders with 3 lowest number of units in the year 2024:


```
SELECT * FROM first_dataset.samplesales
WHERE DATE BETWEEN '2024-01-01' AND '2025-01-01'
ORDER BY Units ASC LIMIT 3;
```

From which we can quickly confirm that the order with the lowest number of units is the one with 3 units.

9.2 Aggregate functions with CASE clause

Earlier we saw that the typical use case for the CASE expression is to add an additional column to the original table.

For e.g. assume we want to provide an additional column to further categorize the number of units for an order based on the table below.

Units purchased	Category
1 – 10	Low
11 – 20	Medium
21 - 25	High
26 and above	Awesome

The statement to achieve this would be:

```
SELECT CustID, Date, Price, Units,
CASE
    WHEN Units >=1 AND Units <=10 THEN 'Low'
    WHEN Units >= 11 AND Units <= 20 THEN 'Medium'
    WHEN Units >= 21 AND Units <= 25 THEN 'High'
    ELSE 'Awesome'
END AS Category
FROM first_dataset.samplesales;
```

Lets assume we now want to take this a step further to find the total number of units in each of the new categories we have created using the CASE statement. To do this, we can write:

```
SELECT
SUM (
    CASE WHEN Units >=1 AND Units <=10
    THEN 1 ELSE 0 END
) AS Low,
SUM (
    CASE WHEN Units >= 11 AND Units <= 20
    THEN 1 ELSE 0 END
) AS Medium,
SUM (
    CASE WHEN Units >= 21 AND Units <= 25
    THEN 1 ELSE 0 END
) AS High,
SUM (
    CASE WHEN Units > 25
    THEN 1 ELSE 0 END
) AS Awesome
FROM first_dataset.samplesales;
```

Similarly, if we wish to find the largest Units value in each of these separate categories, we could rewrite the statement in a slightly different way:

```
SELECT
  MAX (
    CASE WHEN Units >=1 AND Units <=10
      THEN Units ELSE 0 END
  ) AS Low,
  MAX (
    CASE WHEN Units >= 11 AND Units <= 20
      THEN Units ELSE 0 END
  ) AS Medium,
  MAX (
    CASE WHEN Units >= 21 AND Units <= 25
      THEN Units ELSE 0 END
  ) AS High,
  MAX (
    CASE WHEN Units > 25
      THEN Units ELSE 0 END
  ) AS Awesome
FROM first_dataset.samplesales;
```

The logic for statements written in this way is quite convoluted: we will see how later we can make it clearer by nesting queries using subqueries or using views to represent virtual tables.

10 Aggregating and grouping with GROUP BY

The GROUP BY clause divides the rows returned from the SELECT statement into groups. For each group, you can apply the pervious aggregate functions (COUNT, SUM, AVG, MIN, MAX) to specific column items.

The format for using this clause is:

```
SELECT
  column_1,
  column_2,
  ...,
  aggregate_function(column_3)
FROM
  table_name
GROUP BY
  column_1,
  column_2,
  ...;
```

The GROUP BY clause divides the rows by the values in the columns specified in the GROUP BY clause and calculates a value for each group.

The columns specified in the GROUP BY clauses must also appear in the SELECT clause. You'll encounter an error if you specify a column in the SELECT clause that does not appear in the GROUP BY clause, unless there is an aggregation function applied to that COLUMN.

It's possible to use other clauses of the SELECT statement with the GROUP BY clause: such as WHERE, DISTINCT, ORDER BY, LIMIT, etc

In this case, the order of evaluation is as follows:

WHERE -> GROUP BY -> SELECT -> DISTINCT -> ORDER BY -> LIMIT

If we use the ORDER BY clauses by itself without any aggregation function, the result is functionally identical to the SELECT DISTINCT clause.

For e.g. these 2 queries below return identical result, which is the name of the 5 different distinct regions

```
SELECT Region FROM first_dataset.samplesales
GROUP BY Region;
```

```
SELECT DISTINCT Region FROM first_dataset.samplesales;
```

We can use the COUNT aggregation function to determine the number of orders in each particular region with:

```
SELECT Region, COUNT(Region) AS NumOrders
FROM first_dataset.samplesales
GROUP BY Region;
```

We can find the average number of units for orders in each particular region with:

```
SELECT Region, AVG(Units) AS AvgUnits
FROM first_dataset.samplesales
GROUP BY Region;
```

By the same logic, we could find the order with the highest price in each state with:

```
SELECT State, MAX(Price) AS HighestPrice
FROM first_dataset.samplesales
GROUP BY State;
```

To find the total of all units ordered for each product category, we could use:

```
SELECT ProdCategory, SUM(Units) AS TotalUnits
FROM first_dataset.samplesales
GROUP BY ProdCategory;
```

We can extend the previous queries we have just demonstrated by including other clauses (such as WHERE, ORDER BY and LIMIT) to further refine the results returned.

For e.g. to obtain a listing of the top 3 regions based on the average units sold in each particular region, we could use:

```
SELECT Region, AVG(Units) AS AvgUnits
FROM first_dataset.samplesales
GROUP BY Region
ORDER BY AvgUnits DESC
LIMIT 3;
```

To obtain listing of all the orders with the highest price in each state for the year 2022, we could type:

```
SELECT State, MAX(Price) AS HighestPrice
FROM first_dataset.samplesales
WHERE DATE BETWEEN '2022-01-01' AND '2023-01-01'
GROUP BY State
ORDER BY HighestPrice DESC;
```

Notice that the DATE filter applied using the WHERE clauses results in a set of different results from the original GROUP BY query that we had applied earlier. There are only 9 states returned in this result (rather than 14) since there are some states which did not have any orders in the year 2022.

10.1 Grouping multiple columns

We can include more than 1 column in the GROUP BY and also apply an aggregation function to a designated column.

For e.g. to find the number of customers with a specific last name in all the 5 distinct regions, we could type:

```
SELECT Region, LastName, COUNT(LastName) as NumCust
FROM first_dataset.samplesales
GROUP BY Region, LastName;
```

For e.g. to find the highest price of a product in a specific region and state, we could type:

```
SELECT Region, State, Max(Price) as HighestPrice
FROM first_dataset.samplesales
GROUP BY Region, State;
```

10.2 Using HAVING clause to filter on groups

The HAVING clause is used to specify a condition to filter groups returned by the GROUP BY clause.

```
SELECT
    column1,
    aggregate_function (column2)
FROM
    table_name
GROUP BY
    column1
HAVING
    condition;
```

You can also include any of the other clauses we have seen together with the HAVING clause.

In this case, the order of evaluation is as follows:

WHERE -> GROUP BY -> HAVING -> SELECT -> DISTINCT -> ORDER BY -> LIMIT

The HAVING and the WHERE clause both perform filtering. The fundamental difference is:

- WHERE clause filters on all rows from the original table and is performed before GROUP BY
- HAVING clause filters on groups of rows returned by GROUP BY

Let's return to previous query we executed which returns the average number of units for orders in all regions with:

```
SELECT Region, AVG(Units) AS AvgUnits
FROM first_dataset.samplesales
GROUP BY Region;
```

We can further refine this to only include regions where the average number of units for order in that region is higher than a specified number using the HAVING clause:

```
SELECT Region, AVG(Units) AS AvgUnits
FROM first_dataset.samplesales
GROUP BY Region
HAVING AvgUnits > 15;
```

Now consider this query which uses a WHERE without a HAVING:

```
SELECT Region, AVG(Units) AS AvgUnits
FROM first_dataset.samplesales
WHERE Units > 15
GROUP BY Region;
```

Here, we execute the WHERE first, whereby we filter the original set of rows from the table to only include rows where the Units value > 15. Next, we group these filtered rows based on their region and compute the average of Units for all orders in each particular region. Since we initially performed a filter to only retain the rows where Units value > 15, you can see that the average of units for each region is now higher than the original query where no initial WHERE filtering was performed.

We can further refine the regions returned by the query above by now including a HAVING clause to the previous query:

```
SELECT Region, AVG(Units) AS AvgUnits
FROM first_dataset.samplesales
WHERE Units > 15
GROUP BY Region
HAVING AvgUnits > 22;
```

Here the final HAVING filter ensures we only show regions whose average units is above 22.

We could further add in some of the other clauses as we have already seen such as:

```
SELECT Region, AVG(Units) AS AvgUnits
FROM first_dataset.samplesales
WHERE Units > 15
GROUP BY Region
HAVING AvgUnits > 22
ORDER BY AvgUnits DESC
LIMIT 2;
```

We can repeat this exercise again with one of our previous queries:

```
SELECT State, MAX(Price) AS HighestPrice  
FROM first_dataset.samplesales  
GROUP BY State;
```

Now we add an additional HAVING clause to it to only filter the results from the GROUP BY in order to return specific states (rather than all the states) which fulfil a specific condition:

```
SELECT State, MAX(Price) AS HighestPrice  
FROM first_dataset.samplesales  
GROUP BY State  
HAVING HighestPrice > 60;
```

11 END