# Google BigQuery
# Lab 1

# 1 BigQuery 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 BigQuery SQL query syntax

# 2 Lab setup with BigQuery Sandbox

You can use the BigQuery sandbox to explore limited BigQuery capabilities without providing a credit card or creating a billing account for your project. If you already created a billing account, you can still use BigQuery at no cost in the free usage tier.

The BigQuery sandbox is subject to the following limits:

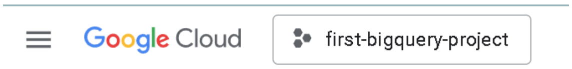
a) All BigQuery quotas and limits apply.
b) You are granted the same free usage limits as the BigQuery free tier, including 10 GB of active storage and 1 TB of processed query data each month.
c) All BigQuery datasets have a default table expiration time, and all tables, views, and partitions automatically expire after 60 days.
d) The BigQuery sandbox does not support several BigQuery features, including the following:
    - Streaming data
    - Data manipulation language (DML) statements
    - BigQuery Data Transfer Service

# 3 Creating a new project and dataset and importing a table

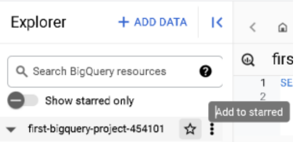
We will start off by creating a new project in which we will create one or more datasets that will hold the various tables and views that we will use in this lab session.

Create a new project with the name: `first-bigquery-project`
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.



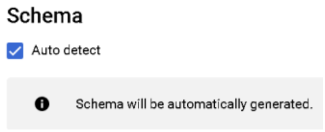
We will create a dataset in this project with the name: `first_dataset`

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: `first-bigquery-project-xxxxx`
Dataset: `first_dataset`
Table: `samplesales`
Table type: `Native table`

Tick Auto detect for Schema.



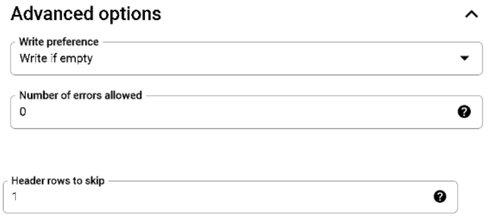
Schema

☑ 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

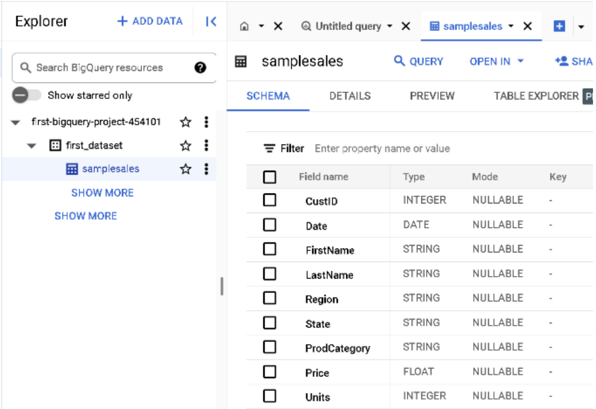
Number of errors allowed: 0

Header rows to skip: 1

Finally, click Create Table.

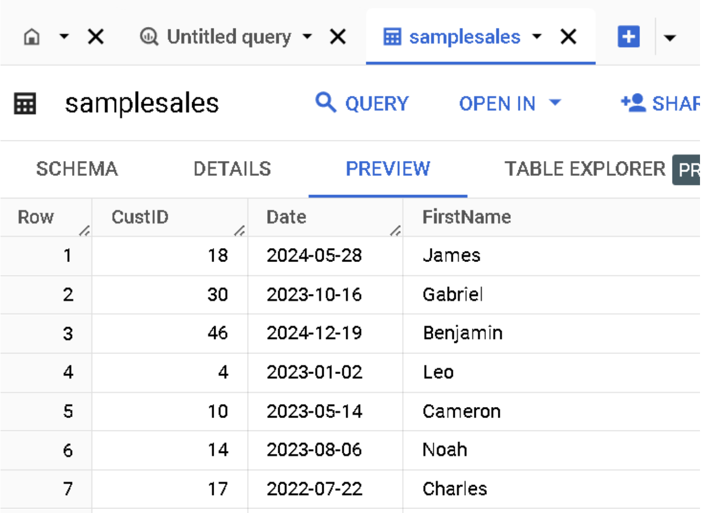
A message about load job running should appear followed by notification about successful creation of the `samplesales` table.

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)



| | Field name | Type | Mode | Key |
|---|---|---|---|---|
| □ | CustID | INTEGER | NULLABLE | - |
| □ | Date | DATE | NULLABLE | - |
| □ | FirstName | STRING | NULLABLE | - |
| □ | LastName | STRING | NULLABLE | - |
| □ | Region | STRING | NULLABLE | - |
| □ | State | STRING | NULLABLE | - |
| □ | ProdCategory | STRING | NULLABLE | - |
| □ | Price | FLOAT | NULLABLE | - |
| □ | Units | INTEGER | NULLABLE | - |

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



| 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.

# 4 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:

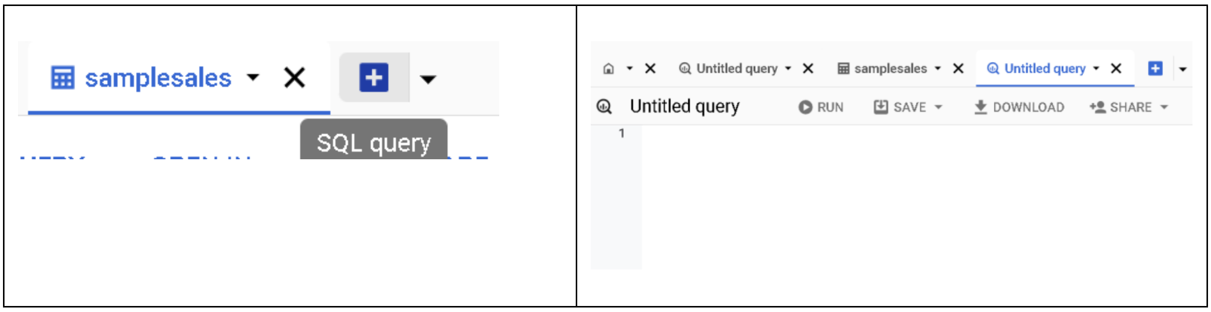
```
SELECT
   select_list
FROM
   table_name;
```

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 be 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 `SELECT` and `FROM` are SQL keywords. This 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).

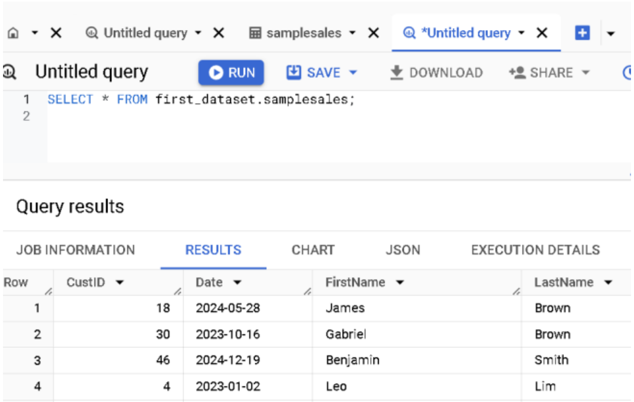
Click on SQL query icon to create a new tab to type in a new SQL query, which will initially be untitled.



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;
```

The results should appear in the Query Results section.



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

| 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.

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;
```

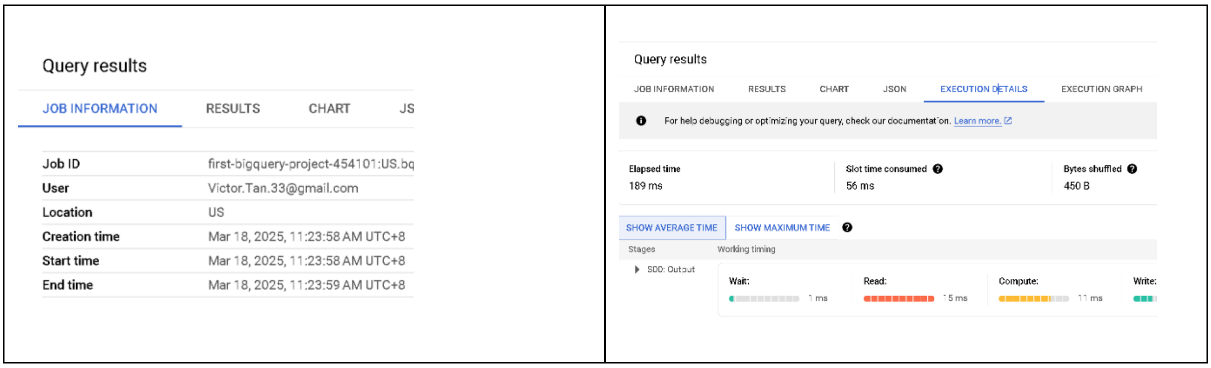
To view only the Price column values from all rows in the table, type in a new SQL query tab and run:

```
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.



| | |
|---|---|
| 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 |

For help debugging or optimizing your query, check our documentation. Learn more.

| Elapsed time | Slot time consumed | Bytes shuffled |
|---|---|---|
| 189 ms | 56 ms | 450 B |

| Stages | Working timing | | | |
|---|---|---|---|---|
| SDD: Output | Wait: 1 ms | Read: 15 ms | Compute: 11 ms | Write: |

## 4.1 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;
```

## 4.2 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;
```

## 4.3 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.

# 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;
```

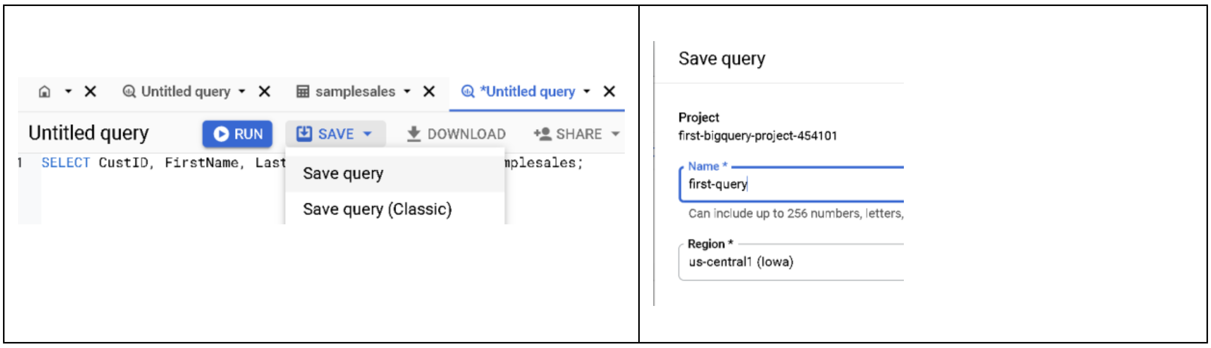
To find the 10 lowest transactions in terms of total sale price, we would use a query like this:

```
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.



```
SELECT CustID, FirstName, Last          mplesales;
```

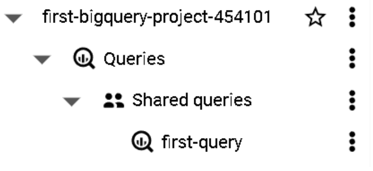
Project
first-bigquery-project-454101

Name *
first-query

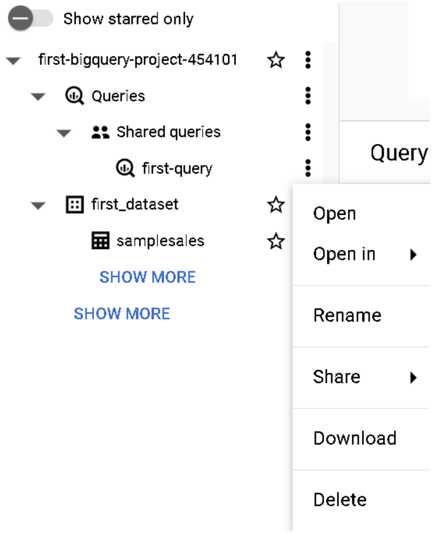
Can include up to 256 numbers, letters,

Region *
us-central1 (Iowa)

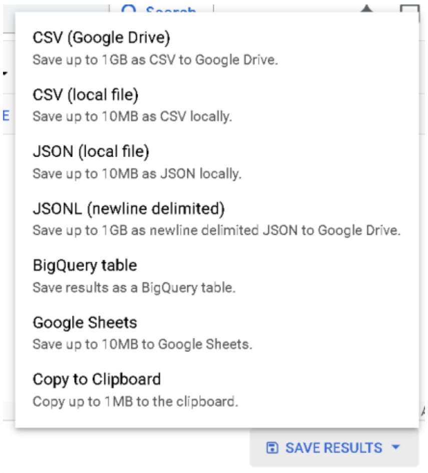
Once it is saved, you should be able to view it in the Queries section of the project in the Explorer pane. You can then close the Query tab and double click on this entry in the Explorer Pane to access it again.



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.



CSV (Google Drive)
Save up to 1GB as CSV to Google Drive.

CSV (local file)
Save up to 10MB as CSV locally.

JSON (local file)
Save up to 10MB as JSON locally.

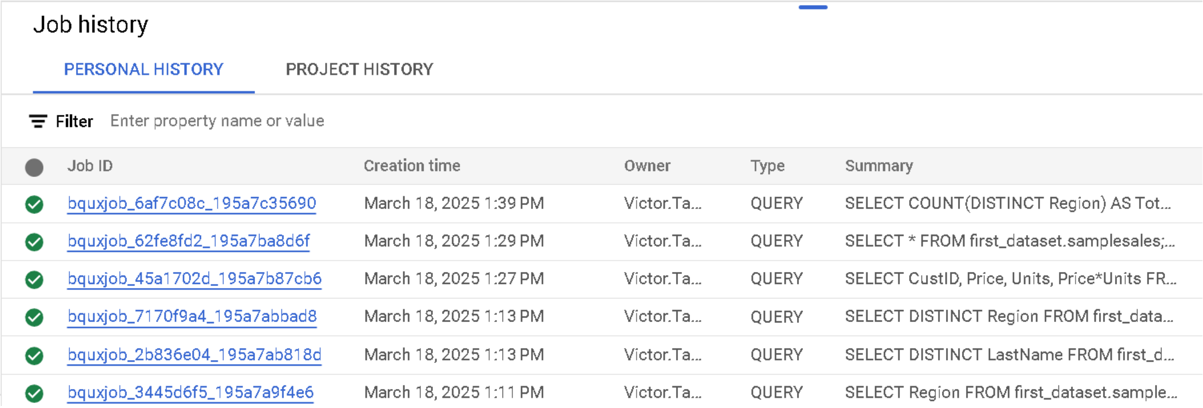
JSONL (newline delimited)
Save up to 1GB as newline delimited JSON to Google Drive.

BigQuery table
Save results as a BigQuery table.

Google Sheets
Save up to 10MB to Google Sheets.

Copy to Clipboard
Copy up to 1MB to the clipboard.

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

| ● | 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;
```

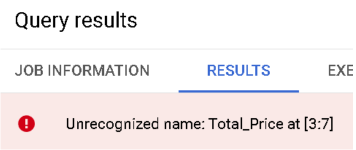
To identify all sales where the total sale price is less than 100.

```
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:



Unrecognized name: Total_Price at [3:7]

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;
```

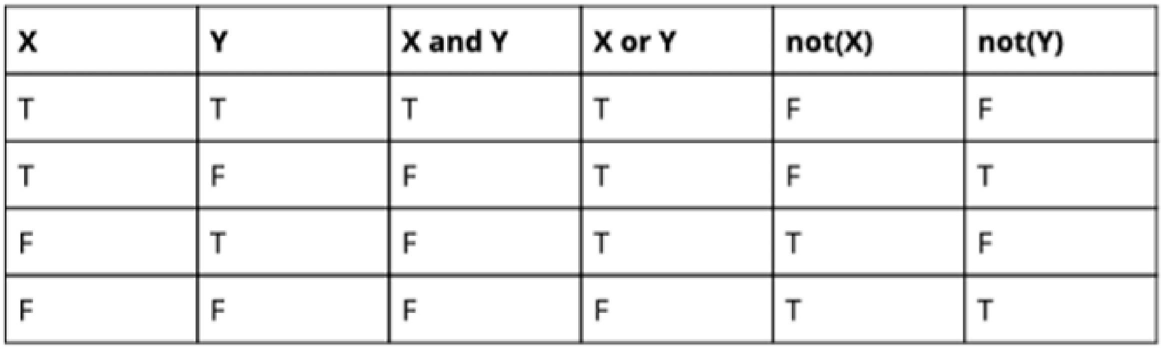
In a complex query like this, the order of evaluation is: **FROM -> WHERE -> SELECT -> ORDER BY -> LIMIT**

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';
```

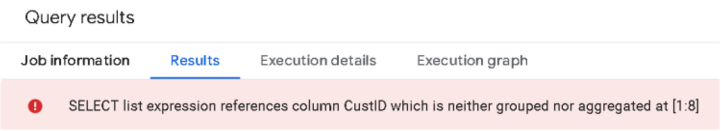
## 9.1 Refining queries with aggregate functions for column details

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.



SELECT list expression references column CustID which is neither grouped nor aggregated at [1:8]

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 GROUPING SETS for multiple simultaneous groupings

GROUPING SETS subclause is an advanced feature of the GROUP BY clause that allows you to create multiple groupings within the same query. This allows you to get multiple results that would typically require several separate queries.

From the previous example, if we wanted 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;
```

Here, the grouping is on both the Region and State columns.

Lets assume we wanted to find the highest price of products in each of the ProdCategories, we would use the statement:

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

Here, the grouping is on only the ProdCategory column

Lets assume we simply wanted to find the product with the highest price over all in the table, in which case the statement would simply be:

```
SELECT Max(Price) as HighestPrice
FROM first_dataset.samplesales;
```

Here, the aggregate function is applied on the entire table content (or effectively there is no grouping).

Now assume we want to combine the previous 3 queries into a single query which essentially seeks to find the product with the highest price in the following 3 groupings:

- Region and State
- ProdCategory
- Entire table (i.e. no grouping applied)

We can do this using the GROUPING SETS subclause to specify these 3 separate groupings as shown below:

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

Each row in the result returned represents results from the 3 individual queries that we issued separately.

a) When you have `null` in all 3 columns, this essentially refers to the aggregate function being applied on the entire table (no grouping)
b) When you have `null` in the ProdCategory column, this essentially refers to the aggregate function being applied on both the Region and State column
c) When you have `null` in both the Region and State column, this essentially refers to the aggregate function being applied only on the ProdCategory column

Note: We will talk more about handling `null` values in an upcoming lab topic

AS you can see, GROUPING SETS does not provide any new grouping or aggregation functionality, but simply allows you to apply aggregate functions simultaneously on multiple groupings within a single query.

## 10.3 Using CUBE for comprehensive combination of groupings

CUBE subclause is another advanced feature of the GROUP BY clause that extends on the concept of GROUPING SETS by creating subgroupings that contain all possible combination of individual columns specified in a typical GROUPING SETS. The CUBE subclause essentially provides a short cut way to define multiple grouping sets in a comprehensive manner.

For e.g. the following two specifications are equivalent

```
CUBE(c1,c2,c3)
```

```
GROUPING SETS (
     (c1,c2,c3),
     (c1,c2),
     (c1,c3),
     (c2,c3),
     (c1),
     (c2),
     (c3),
     ()
 )
```

In general, if the number of columns specified in the CUBE is n, then you will have $2^n$ combinations.

Consider the result returned from the statement below:

```
SELECT Region, State, SUM(Units) as TotalUnits
FROM first_dataset.samplesales
GROUP BY CUBE(Region, State)
ORDER BY Region, State;
```

Each row in the result returned represents results from the 3 individual queries that we issued separately.

a) When you have `null` in all 3 columns, this essentially refers to the aggregate function being applied on the entire table (no grouping)
b) When you have `null` in the State column, this essentially applies to the aggregate function being applied to the Region column
c) When you have `null` in the Region column, this essentially applies to the aggregate function being applied to the State column
d) There is a combination of all possible grouping of values for both the Region and State columns

As you can see, there is total of 34 possible combinations of unique groupings involving all possible values for these 2 columns.

## 10.4 Using ROLLUP for hierarchical combination of groupings

ROLLUP subclause is another advanced feature of the GROUP BY clause that extends on the concept of GROUPING SETS by creating subgroupings that contain all hierarchical combination of individual columns specified in a typical GROUPING SETS. The ROLLUP subclause is conceptually similar to the CUBE subclause, except that it assumes a hierarchical relationship between the columns in the grouping and creates combination of groupings based on this hierarchy. This is as opposed to the CUBE subclause which generates all possible unique combination of groupings without any assumption with regards to hierarchical relationships.

For e.g. the following two specifications are equivalent (contrast this with the case of CUBE).

```
ROLLUP(c1,c2,c3)

GROUPING SETS (
    (c1,c2,c3),
    (c1,c2),
    (c1),
    ()
 )
```

The classic use case for ROLLUP is to generate subtotals and grand totals for reports where there is hierarchy between the categories for multiple columns.

In our table, we have a nested hierarchy in Region and State: there are multiple states in each Region. With the statement below:

```
SELECT Region, State, SUM(Units) as TotalUnits
FROM first_dataset.samplesales
GROUP BY ROLLUP(Region, State)
ORDER BY Region, State;
```

We can see the grand total of units sold for each Region (Central, East, East Malaysia, Northern and Southern) and the subtotals for units sold in each state in that Region.

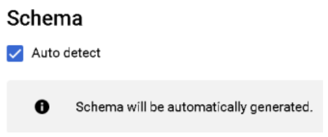
As you can see, there is total of 20 possible combinations of hierarchical groupings involving possible values for these 2 columns (as compared to the 34 for a CUBE subclause).
Another common use case is to compute running totals and grant totals for unit sales / revenue across hierarchical periods of time (Year, Quarter, Month, etc) for historical transactions.

To demonstrate this, we will use the `sales-time.csv` file in the `data` subfolder of the downloaded workshop resources to populate the contents of a new table in the existing dataset. You can open `sales-time.csv` in Excel to quickly preview it first if you wish.

Next, we will create another table in the existing dataset `first_dataset` with the following values in the dialog box that appears.

Create table from: `Upload`
Select file*: `salestime.csv`
File Format: `CSV`
Destination: `first-bigquery-project-xxxxx`
Dataset: `first_dataset`
Table: `salestime`
Table type: `Native table`

Tick Auto detect for Schema.

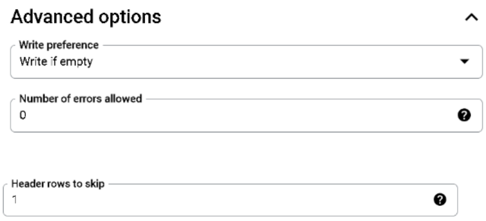


Schema

☑ Auto detect

Schema will be automatically generated.

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 `salestime` table.

NOTE: This table data is not realistic since there will be certain months of the year which will have more than 4 weeks, and certain weeks will be split across consecutive months. The data is provided simply to demonstrate clearly the concept of hierarchical grouping.

With this statement below:

```
SELECT Year, Quarter, Month, SUM(Units) as TotalUnits
FROM first_dataset.salestime
GROUP BY ROLLUP(Year, Quarter, Month);
```

We now are able to get grand totals for each year and also running totals for the hierarchical groups such as Quarter and Month.

## 10.5 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 Working with missing / NULL values in a dataset table

Occasionally, the source CSV file that we use to populate new tables in BigQuery datasets may have missing values in them. The most common reasons for this are:

a) Omitted fields during manual entry – Users may leave out fields unintentionally or because they lack the required information.
b) Input errors or typos – Mistakes that prevent values from being stored properly

c) Not applicable (N/A) scenarios – Some fields may not be relevant for certain records (e.g., spouse name for single individuals, or monthly rent for house owners)
d) Privacy or security concerns – Sensitive values (e.g., income, health data) may be excluded or anonymized.
e) Survey non-responses – In questionnaires or feedback forms, participants might choose not to answer certain questions.
f) Sensor or equipment failure – In IoT or industrial contexts, devices may malfunction and fail to report values.
g) File corruption or truncation – Part of the data file may be damaged or improperly saved.
h) Software bugs in data pipelines – Issues in ETL (Extract, Transform, Load) processes can skip or nullify data

When a CSV file with missing values is imported into BigQuery to populate a new table, the cells with the missing values with have a special value called NULL. Handling NULL values effectively in BigQuery SQL syntax is crucial when writing queries to avoid incorrect results or logic errors.

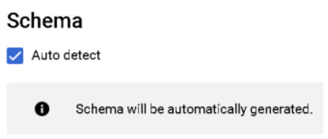
We will use the `samplesales-withnull.csv` file in the `data` subfolder of the downloaded workshop resources to populate the contents of a new table in our existing dataset. You can open `samplesales-withnull.csv` in Excel to quickly preview it first if you wish.

This file is identical in structure to the original `samplesales-withnull.csv` data, with the main exception that it contains missing values in various cells of the table.

Create another table in the existing dataset `first_dataset` with the following values in the dialog box that appears.

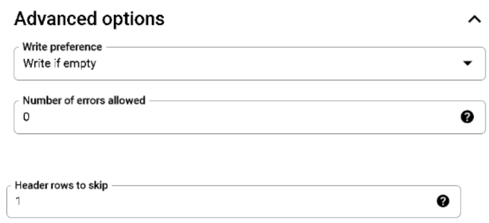
Create table from: `Upload`
Select file*: `samplesales-withnull.csv`
File Format: `CSV`
Destination: `first-bigquery-project-xxxxx`
Dataset: `first_dataset`
Table: `saleswithnull`
Table type: `Native table`

Tick Auto detect for Schema.



Schema
☑ Auto detect
Schema will be automatically generated.

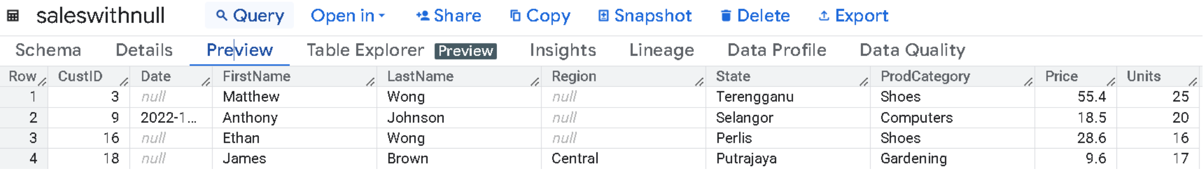
In Advanced Options, type 1 for Header Rows to skip as the first row in our CSV file is essentially a header row containing the names of the columns/fields for the table we are creating. You can leave the other options as they are.



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 `saleswithnull` table.

Click on Preview tab in the Details pane to all the rows in this table. You will notice that all the missing values in the original CSV file are now replaced with the value `null` in the table.



| Row | CustID | Date | FirstName | LastName | Region | State | ProdCategory | Price | Units |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | null | Matthew | Wong | null | Terengganu | Shoes | 55.4 | 25 |
| 2 | 9 | 2022-1... | Anthony | Johnson | null | Selangor | Computers | 18.5 | 20 |
| 3 | 16 | null | Ethan | Wong | null | Perlis | Shoes | 28.6 | 16 |
| 4 | 18 | null | James | Brown | Central | Putrajaya | Gardening | 9.6 | 17 |

## 11.1 Using IS NULL / IS NOT NULL operators

We can detect `null` values or vice versa, valid values (non-null values) through the WHERE clause and the NULL operator

For e.g. to show all rows with the value of `null` in the Region column, we write:

```
SELECT * FROM first_dataset.saleswithnull
WHERE Region IS NULL;
```

Similarly, to show all rows with the value of `null` in the Price column, we write:

```
SELECT * FROM first_dataset.saleswithnull
WHERE Price IS NULL;
```

If there are many columns in the table where `null` values occur, we use a more complex condition to filter these rows by combining conditions using the AND, OR or NOT operators.

For e.g. to view rows where the Date and Region column are both `null`:

```
SELECT * FROM first_dataset.saleswithnull
WHERE Date IS NULL AND Region is NULL;
```

To view rows, where either the Price column or Units column have the `null` value:

```
SELECT * FROM first_dataset.saleswithnull
WHERE Price IS NULL OR Units is NULL;
```

Vice versa, we may sometimes be interested to view rows that have valid values (i.e. non-null values) for a specific column. For e.g. to show all rows with valid values in the Date column, we write:

```
SELECT * FROM first_dataset.saleswithnull
WHERE Date IS NOT NULL;
```

Quite often, we just want to determine how many rows / records in the table have `null` for a particular column / field, and we can use the COUNT function for that.

```
SELECT COUNT(*) AS NullRows FROM first_dataset.saleswithnull
WHERE Region IS NULL;
```

Vice versa, to determine the number of rows / records in the table that have a valid (non-null) value for a particular column / field, and we can use the COUNT function without the WHERE, as by default COUNT will only tabulate non-null values in that column.

```
SELECT COUNT(Region) AS ValidRows
FROM first_dataset.saleswithnull;
```

## 11.2 Behavior of NULL with mathematical expressions

If you involve columns with NULL values in mathematical expressions, the result is always NULL. For e.g. when evaluating the calculating the total price of an order where either the Price column or Unit column can potentially have NULL values, the final result is NULL.
You can verify this with the statement:

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

In a situation like this, you may only wish to perform a Total Price calculation on the rows which have valid values for BOTH the Price column and Unit column. This can be accomplished using the WHERE clause together with the AND operator. As an example:

```
SELECT CustID, Price, Units, Price*Units AS Total_Price
FROM first_dataset.saleswithnull
WHERE Price IS NOT NULL AND Units IS NOT NULL;
```

You can also replace null values with default values so that the expression evaluates correctly for all rows using the COALESCE function as we will see later.

## 11.3 Behavior of NULL with aggregate functions (COUNT, SUM, AVG, etc)

By default, aggregate functions such as COUNT, SUM, AVERAGE, MIN, MAX will ignore `null` values in their computation.

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

SELECT AVG(Units) AS AVERAGE_UNITS
FROM first_dataset.saleswithnull;
```

You can verify this by performing the same computation in the original CSV file in Excel, but replacing the missing values with 0.

## 11.4 Behavior of NULL with ORDER BY

`Null` values in a column are considered to be the lowest in ranking (whether alphabetic or numeric). Therefore, these values will appear first in an ascending sort (the default sort order) and appear last in a descending sort.

Check this out with:

```
SELECT * FROM first_dataset.saleswithnull
ORDER BY Price;

SELECT * FROM first_dataset.saleswithnull
ORDER BY Units DESC;

SELECT * FROM first_dataset.saleswithnull
ORDER BY FirstName DESC;
```

As usual, if you wish to exclude rows with `null` values from the sorting operation, filter them using the WHERE clause and the NULL operator, for e.g.

```
SELECT * FROM first_dataset.saleswithnull
WHERE Price IS NOT NULL
ORDER BY Price;

SELECT * FROM first_dataset.saleswithnull
WHERE Units IS NOT NULL
ORDER BY Units DESC;
```

## 11.5 Behavior of NULL with GROUP BY

Rows with `null` values are considered to be a separate group by themselves and are treated accordingly as such by the GROUP BY clause, for e.g.

```
SELECT Region, COUNT(Region) AS NumRegions
FROM first_dataset.saleswithnull
GROUP BY Region;
```

However, notice that the count for the Region column which has null values is incorrectly shown as 0, when in fact it is more than that, and we can verify this with:

```
SELECT COUNT(*) AS NullRows FROM first_dataset.saleswithnull
WHERE Region IS NULL;
```

To resolve this subtle issue, we will need to use the COALESCE function which we will see next

## 11.6 Using the COALESCE function for default values

The COLAESCE function accepts a list of arguments and returns the first non-null argument. It evaluates arguments from left to right until it finds the first non-null argument. All the remaining arguments from the first non-null argument are not evaluated. It is useful for replacing NULL values with default values.

A simple example might be that we wish to display the values in the Price column only, but replace the rows which have `null` with 0. To accomplish this, we write:

```
SELECT CustID, COALESCE(Price, 0) FROM first_dataset.saleswithnull;
```

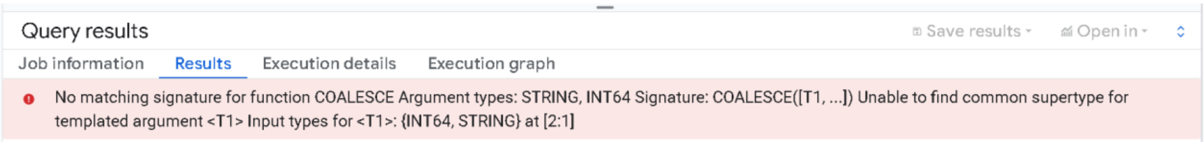
A more extended example: we wish to display the first 4 columns in the table (CustID, Date, FirstName and LastName). We know that Date and FirstName columns have `null` values in certain rows. For those rows, instead of displaying `null`, we may wish to display a default value of 2000-01-01 for the case of Date and Not provided for the case of FirstName. To accomplish this, we write:

```
SELECT CustID, COALESCE(Date, '2000-01-01') AS NewDate,
COALESCE(FirstName, 'Not Provided') AS NewFirstName, LastName
FROM first_dataset.saleswithnull;
```

Notice that the default value we specify must be of the correct type for that particular column. For e.g. if we issued the statement below:

```
SELECT CustID, COALESCE(Date, '2000-01-01') AS NewDate,
COALESCE(FirstName, 0) AS NewFirstName, LastName
FROM first_dataset.saleswithnull;
```

We would get an error as shown below since FirstName is a string type and the default value we want to replace it with is numeric, so we have a type mismatch.



No matching signature for function COALESCE Argument types: STRING, INT64 Signature: COALESCE([T1, ...]) Unable to find common supertype for templated argument <T1> Input types for <T1>: {INT64, STRING} at [2:1]

The default value provided by the COALESCE function can be used to ensure that the functionality of the other clauses (such as GROUP BY) works properly. We had seen earlier that if we had executed this statement:

```
SELECT Region, COUNT(Region) AS NumRegions
FROM first_dataset.saleswithnull
GROUP BY Region;
```

We would get a value of 0 for the `null` Region, which is incorrect in this particular situation, since there are 1 or more rows which actually have the value of `null` for the Region column.

To get the correct result, we replace `null` with some other appropriate default value (for e.g. Unknown) using the COALESCE function as shown below:

```
SELECT COALESCE(Region, 'Unknown') AS Region,
COUNT(*) AS NumRegions
FROM first_dataset.saleswithnull
GROUP BY COALESCE(Region, 'Unknown');
```

The default value provided by the COALESCE function can be used in mathematical expressions.
For e.g. earlier e.g. when evaluating the calculating the total price of an order where either the Price column or Unit column can potentially have NULL values, the final result is NULL.

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

We can use COALESCE to replace `null` with some appropriate default value for Units and Total Price (for e.g. the value 1), and then reevaluate the expression with:

```
SELECT CustID, COALESCE(Price, 1) AS Price,
COALESCE(Units, 1) AS Units,
COALESCE(Price, 1) * COALESCE(Units, 1) AS Total_Price
FROM first_dataset.saleswithnull;
```

In the same way, the default value provided by the COALESCE function can be used in aggregate functions. Earlier, we saw that aggregate functions such as COUNT, SUM, AVERAGE, MIN, MAX will ignore `null` values in their computation:

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

You can elect to replace `null` with a default value which can then subsequently be used in a particular aggregate function for e.g:

```
SELECT SUM(COALESCE(Units, 1)) AS TOTAL_UNITS
FROM first_dataset.saleswithnull;
```

## 11.7 Using the NULLIF function

So far, we have seen the various issues related to working with `null` values in any table column. There is a special function NULLIF which compares two expressions and returns the `null` value if they are equal. Otherwise, it returns the result of the first expression

As a simple example, try out the following expressions which work on literal values without accessing a table:

```
SELECT NULLIF(5, 3) result;

SELECT NULLIF('cat', 'dog') result;
```

In the first 2 examples above, the two expressions compared are not equal, so the result is simply the first expression.

```
SELECT NULLIF(5, 5) result;

SELECT NULLIF('cat', 'cat') result;
```

In the next 2 examples above, the two expressions compared are equal, so the result will be null.

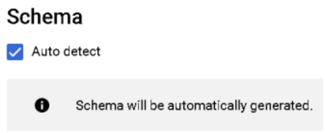
The classic use case for this function is to prevent division by zero error when executing mathematical expressions.

To demonstrate this, we will use the `salary.csv` file in the `data` subfolder of the downloaded workshop resources to populate the contents of a new table in the existing dataset. You can open `salary.csv` in Excel to quickly preview it first if you wish.

Create another table in the existing dataset `first_dataset` with the following values in the dialog box that appears.

Create table from: `Upload`
Select file*: `salary.csv`
File Format: `CSV`
Destination: `first-bigquery-project-xxxxx`
Dataset: `first_dataset`
Table: `salary`
Table type: `Native table`

Tick Auto detect for Schema.

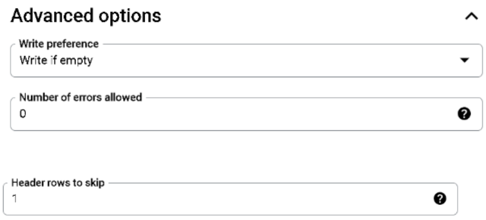


Schema

☑ Auto detect

Schema will be automatically generated.

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

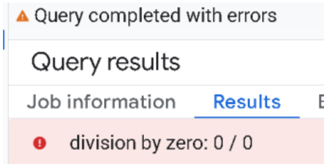
Finally, click Create Table.
A message about load job running should appear followed by notification about successful creation of the `salary` table.

Assume you wish to compute the average daily salary of each employee = monthly salary / days worked. The problem here however is that some employees did not work at all (DaysWorked = 0).

If we directly attempt a computed expression such as:

```
SELECT EmployeeID, Name,
MonthlySalary/DaysWorked AS DailySalary
FROM first_dataset.salary;
```

We will encounter a divide by 0 error.



Query completed with errors

Query results

division by zero: 0 / 0

We can obviously avoid this error by only performing the computation on rows where the DaysWorked is more than 0, for e.g.

```
SELECT EmployeeID, Name,
MonthlySalary/DaysWorked AS DailySalary
FROM first_dataset.salary WHERE DaysWorked > 0;
```

This works but has a tiny drawback: we not able to see the employees who are filtered out from the original table.
If we wish to be able to see all the employees in the table, but at the same time denote that the DailySalary computation is not valid for certain employees, we use the NULLIF:

```
SELECT EmployeeID, Name,
MonthlySalary / NULLIF(DaysWorked,0) AS DailySalary
FROM first_dataset.salary;
```

In this case, the computed expression is executed for all errors, but any mathematical expression involving `null` as a value returns `null` as a result, without causing any errors.

# 12 END