**Building a Data Warehouse.**

**Module introduction.**

Hello and welcome to the Building a Data Warehouse module.

This is the third module in the course, Modernizing Data Lakes and Data Warehouses with Google Cloud.

We'll start by describing what makes a modern data warehouse.

We'll also talk about what distinguishes a data lake from an enterprise data warehouse.

Then we're going to introduce BigQuery, a data warehouse solution on Google Cloud.

Once you're familiar with the basics of BigQuery, we'll talk about how BigQuery organizes your data and then how to load new data into BigQuery.

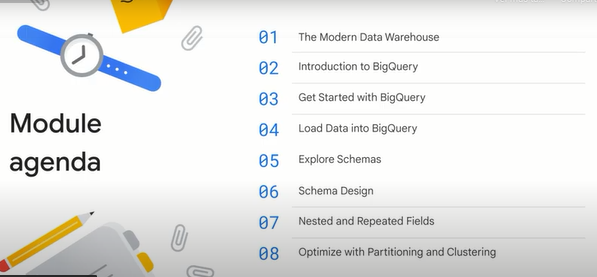
You'll also have the opportunity to load data into BigQuery through a hands-on lab.

Finally, we'll dive into the world of data warehouse schemas.

We'll talk about efficient data warehouse schema design and take a closer look at BigQuery support for nested and repeated fields and why this is such a popular schema design for enterprises.

You'll get some experience working with JSON and array data in BigQuery through a hands-on lab.

We'll end by discussing how you can optimize the tables in your data warehouse with partitioning and clustering.



**The modern data warehouse.**

Let's start by describing what makes a modern data warehouse.

An enterprise data warehouse should consolidate data from many sources.

If you recall from the previous module, a data lake does something very similar.

The key difference between the two is the word consolidate.

A data warehouse imposes a schema.

A data lake is just raw data, but an enterprise data warehouse brings the data together and makes it available for querying and data processing.

To use a data warehouse, an analyst needs to know the schema of the data.

However, unlike for a data lake, the analyst doesn't have to write code to read and parse the data.

Another reason to consolidate all your data besides standardizing the format and making it available for querying is making sure the query results are meaningful.

You want to make sure the data is clean, accurate and consistent.

The purpose of a data warehouse is not to store data.

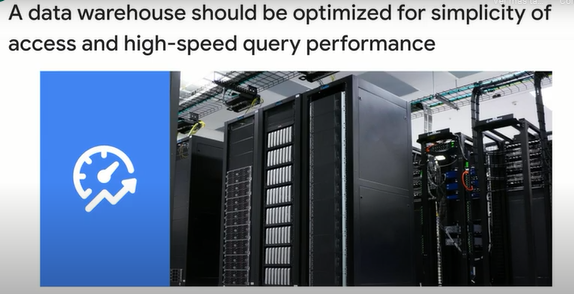
That's the purpose of a data lake.

If you have raw data that you want to keep around but not necessarily query, don't bother with cleaning and streamlining it, leave it in a data lake.

All data in a data warehouse should be available for querying.

It's important to ensure that those queries are quick.

You don't want people waiting hours or days for results.



We described an enterprise data warehouse and how it's different from a data lake.

hat makes a data warehouse modern?

Businesses' data requirements continue to grow, you want to make sure the data warehouse can deal with datasets that don't fit into memory.

Typically, this is gigabytes to terabytes of data, but occasionally can be petabytes.

You don't want separate warehouses for different datasets.

Instead, you want a single data warehouse that can scale from gigabytes to petabytes of data.

Second, you want the data warehouse to be serverless and fully no-ops.

You don't want to be limited to clusters that you need to maintain, or indexes that you need to fine-tune.

Removing these responsibilities will allow data analysts to carry out ad hoc queries faster, which is important because you want the data warehouse to increase the speed at which your business makes decisions.

Next, your data warehouse is not productive if it allows you to do queries but doesn't support rich visualization and reporting.

Ideally, your data warehouse can seamlessly plug into whichever visualization or reporting tool your business is most familiar with.

Similarly, because the data warehouse requires clean and consistent data, you will often have to build data pipelines to bring data into the warehouse.

The modern data warehouse should be able to integrate with an ecosystem of processing tools for building ETL pipelines.

Your data pipeline should be capable of constantly refreshing data in the warehouse in order to keep it up to date.

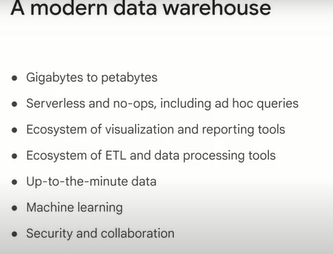
You need to be able to stream data into the warehouse and not rely on batch updates.

Also, predictive analytics is becoming increasingly important for data analysts.

As a result, a modern data warehouse has to support machine learning without moving the data out of the warehouse.

Last but not least, in a modern data warehouse, it should be possible to impose enterprise grade security like data exfiltration constraints.

It should also be possible to share data and queries with collaborators.



**Introduction to BigQuery.**

In this lesson, we're going to introduce BigQuery, a data warehouse solution on Google Cloud.

BigQuery has many capabilities that make it an ideal data warehouse.

When we talked about a modern data warehouse, we talked about having the warehouse be able to scale from gigabytes to petabytes seamlessly (perfectamente).

We talked about being able to do ad hoc queries and no-ops.

BigQuery cost-effectively handles large petabyte scale datasets for storage and querying.

In fact, it's similar to the cost of Cloud storage.

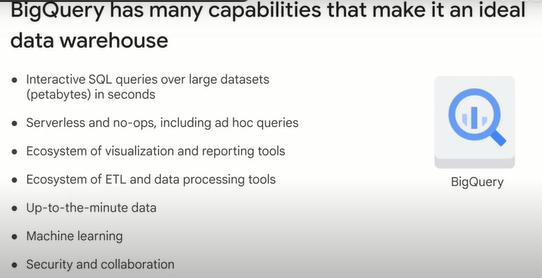
This enables you to store your data without having to worry about archiving off older data to save on storage.

Unlike traditional data warehouses, BigQuery has features like GIS and machine learning built in.

It also provides capabilities to stream data in so you can analyze your data in near real time.

Because it's part of Google Cloud, you get all of the security benefits the Cloud provides while also being able to share datasets and queries.

BigQuery supports standard SQL queries and is compatible with ANSI SQL 2011.



BigQuery is a serverless fully managed service, which means that the BigQuery engineering team takes care of updates and maintenance for you.

Upgrades don't require downtime or hinder system performance.

For example, data aging and expiration can be a cumbersome (engorrosa) operation in traditional data warehouses.

In BigQuery, you just supply a table expiration flag at the time of table creation, or update a table to add this feature.

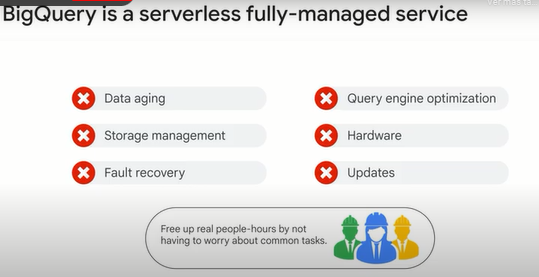
The table will automatically expire when it reaches that age or duration.

Many traditional systems require resource-intensive vacuum processes to run at various intervals to reshuffle and sort data blocks and recover space.

BigQuery has no equivalent of the vacuum process because the storage engine continuously manages and optimizes how data is stored and replicated.

Also, because BigQuery doesn't use indexes on tables, you don't need to rebuild these.

The bottom line is that you can free up real work hours by not having to worry about common database management tasks.



So what makes BigQuery fast?

BigQuery tables are column orientated as compared to traditional RDBM tables which are row orientated.

Row orientated tables are efficient for making updates to data contained in fields.

For OLTP systems, row orientated tables are necessary because OLTP systems have frequent updates.

Analytics is slow on row orientated tables because they have to read all the fields in a row.

And depending on the kind of indexing or key, they may have to read extra rows and fields to find the information that is requested in the query.

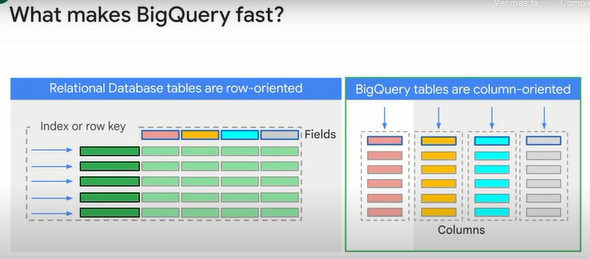
BigQuery however, is an OLAP system, is meant for analytics.

BigQuery tables are immutable and are optimized for reading and appending data.

BigQuery tables are not optimized for updating.

BigQuery leverages the fact that most queries involve few columns, and so it only reads the columns required for the query.

BigQuery is very efficient in the sense and is the reason tables are column orientated.



BigQuery is implemented in two parts, a storage engine and an analytic engine as illustrated.

The separation of compute and storage is a common theme in Google Cloud and works effectively because of Google's petabit network called Jupiter.

Jupiter allows blazing fast communication between compute and storage.

BigQuery data is physically stored on Google's distributed file system called Colossus, which ensures your ability by using a ratio encoding to store redundant chunks of the data on multiple physical disks.

Moreover, the data is replicated to multiple data centers.

Here are a couple of the optimizations that Capacitor does.

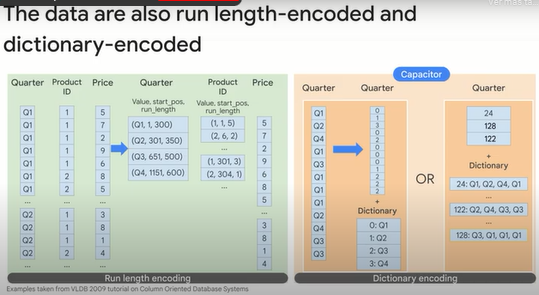
Capacitor runs length encodes on the data so that it can reduce the amount of data needed to be read.

It also reorders the data to make it more conducive for run length encoding.

Reordering the data is also called dictionary encoding.

Now all this beneath the covers happens in BigQuery native storage, it doesn't affect you in any way.

That's the whole point of serverless and fully managed.



You don't need to provision resources before using BigQuery unlike many RDBMS systems.

BigQuery allocates storage and query resources dynamically based on your usage patterns.

Storage resources are allocated as you consume them and deallocated as you remove data or drop tables.

Query resources are allocated according to query type and complexity.

Each query uses some number of slots, which are units of computation that comprise a certain amount of CPU and RAM.

You don't have to make a minimum usage commitment to use BigQuery, the service allocates and charges for resources based on your actual usage.

By default, all BigQuery customers have access to 2000 slots for query operations.

You can also reserve a fixed number of slots for your project.

BigQuery is implemented using a micro service architecture, so there are no virtual machines to configure and maintain.

Under the hood, analytics throughput is measured in BigQuery slots.

A BigQuery slot is a unit of computational capacity required to execute SQL queries.

BigQuery automatically calculates how many slots are required at each stage in a query depending on size and complexity.

A BigQuery slot is a combination of CPU, memory and networking resources.

It also includes a number of supporting technologies and sub services.

Note that each slot doesn't necessarily have the same specification during query execution.

Some slots may have more memory than others, or more CPU, or more IO.

By default, each account has a quota limit of 2000 to BigQuery slots for on-demand querying.

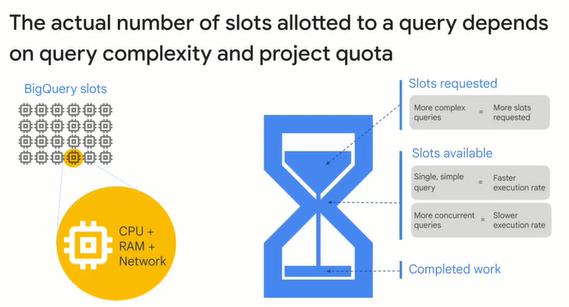
A flat rate pricing model is available that provides reserved slots for customers who want more predictable pricing.

If a single simple query is submitted that needs fewer slots than are available, the query will generally execute faster.

If you've reserved 10,000 slots but you have 30 concurrent queries that together ask for 15,000 slots, the queries will not get all the slots they require.

Instead, the slots are divided fairly among all the projects in the reservation and all the queries in the project.

This will generally result in each query executing more slowly.



**Get started with BigQuery.**

Now that you're familiar with the basics of BigQuery, it's time to talk about how BigQuery organizes your data.

BigQuery organizes data tables into units called datasets.

These datasets are scoped to your Google Cloud project.

When you reference a table from the command line in SQL queries or in code, you refer to it by using the construct Project dot dataset dot table.

What are some reasons to structure your information into datasets, projects and tables?

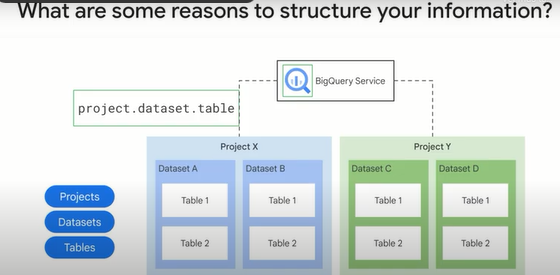
These multiple scopes, project, dataset and table, can help you structure your information logically.

You can use multiple datasets to separate tables pertaining to different analytical domains.

And you can use project level scoping to isolate datasets from each other according to your business needs.

Also, as we will discuss later, you can align projects to billing and use datasets for access control.

You store data in separate tables based on logical schema considerations.



The project is what the billing is associated with.

For example, if you query a table that belongs to the BigQuery public data project, the storage costs are built in that data project.

To run a query, you need to be logged into the Cloud console, you will run a query in your own Google Cloud project and the query charges are built to your project, not to the public data project.

In order to run a query in a project, you need identity access management, IAM permission, to submit a job.

Remember that running a query means that you must be able to submit a query job to the service.

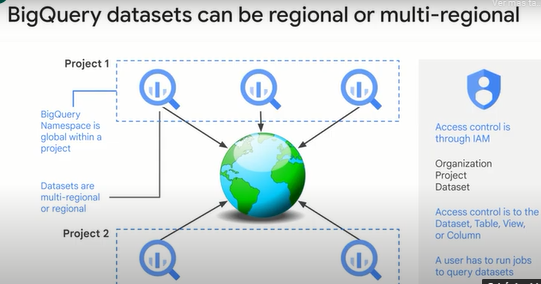
Access control is through IAM and is at the dataset table view or column level.

In order to query data in a table or view, you need at least read permissions on the table or view.

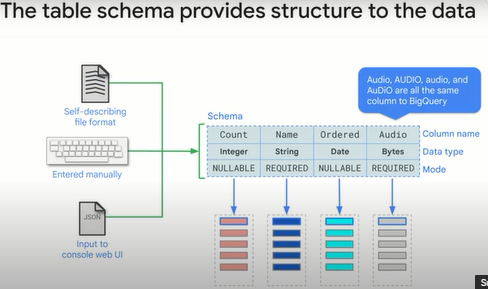
Like Cloud storage, BigQuery datasets can be regional or multi-regional.

Regional datasets are replicated across multiple zones in the region.

Multi-regional means replication among multiple regions.



Every table has a schema, you can enter the schema manually to the Cloud Console, or by supplying a JSON file.



As with Cloud storage, BigQuery storage encrypts data at REST and over the wire using Google managed encryption keys.

It's also possible to use customer managed encryption keys.

Authentication is through IAM, and so it's possible to use Gmail addresses or Google workspace accounts for this task.

Access control is through IAM roles, and involves giving permissions.

We discussed two of these in read access and the ability to submit Query jobs.

However, many other permissions are possible.

Remember that access control is at the level of datasets, tables, views or columns.

When you provide access to a dataset, either read or write, you provide access to all the tables in that dataset.

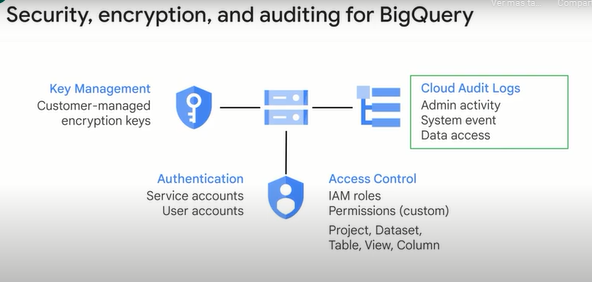
Logs in BigQuery are immutable and are available to be exported to Cloud operations.

Admin activities and system events are all logged.

An example of a system event is table expiration.

If when creating a table, you configure it to expire in 30 days, at the end of 30 days, a system event will be generated and logged.

You will also get immutable logs of every access that happens to a dataset under your project.



BigQuery provides predefined roles for controlling access to resources.

You can also create custom IAM roles consisting of your defined set of permissions, and then assign those roles to users or groups.

You can assign a role to a Google email address or to a Google workspace group.

An important aspect of operating a data warehouse is allowing shared per controlled access against the same data to different groups of users.

For example, finance, HR and marketing departments all access the same tables, but their levels of access differ.

Traditional data warehousing tools make this possible by enforcing Row Level Security.

You can achieve the same results in BigQuery with access control to datasets, tables, views or columns, or by defining authorized views and row level permissions.



In BigQuery, Row Level Security involves the creation of row level access policies on a target BigQuery table.

This policy then acts as a filter to hide or display certain rows of data depending on whether a user or a group is in an allowed list.

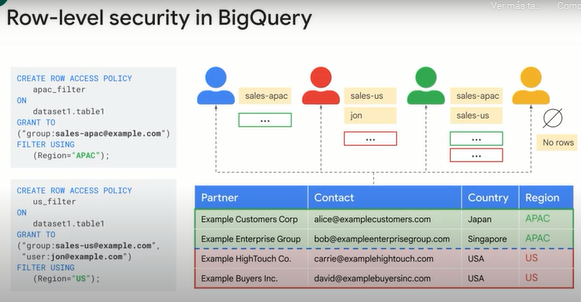
An authorized user with the IAM roles BigQuery admin or BigQuery data owner can create row level access policies on a BigQuery table.

When you create a row level access policy, you specify the table by name and which users or groups called the grantee list should have access to certain row data.

The policy includes the data on which you wish to filter, called the filter expression.

The filter expression functions like a Where clause in a typical query.

In this example, users in the group APAC can only see partners from the APAC region.



Giving view access to a dataset is also known as creating an authorized view in BigQuery.

An authorized view allows you to share query results with particular users and groups without giving them access to the underlying source data.

You can also use the view SQL query to restrict the columns, fields the users are able to query.

Sharing access to datasets is easy.

Traditionally, onboarding new data analysts involves significant lead time.

To enable analysts to run simple queries, you had to show them where data sources resided, and set up ODBC connections and tools and access rights.

Using Google Cloud, you can greatly accelerate an analyst time to productivity.

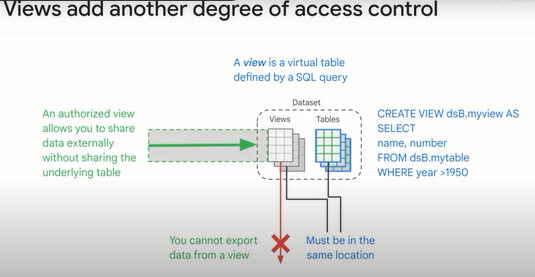
To onboard an analyst on Google Cloud, you grant access to relevant projects, introduce them to the Cloud Console and BigQuery web UI, and share some queries to help them get acquainted with the data.

The Cloud Console provides a centralized view of all assets in your Google Cloud environment.

The most relevant asset to data analysts might be Cloud Storage buckets, where they can collaborate on files.

The BigQuery web UI presents the list of datasets that the analyst has access to.

Analysts can perform tasks in the Cloud Console according to the role you grant them, such as viewing metadata, previewing data, executing and saving and sharing queries.



When you provide read access to a dataset to a user, every table in that dataset is readable by that user.

What if you want more fine grained control?

In addition to access controls at the table or column level, you can use views.

In this example, we are creating a view in dataset B, and the view is a subset of the table data in dataset A. Now, by providing users with access to dataset B, we are creating an authorized view that is only a subset of the original data.

Note that you cannot export data from a view and dataset B has to be in the same region or multiregion as dataset A. A view as a SQL query that looks like and has properties similar to a table.

You can query a view just like you query a table.

BigQuery supports materialized views as well.

These are views that are persisted so that the table does not need to be queried every time the view is used.

BigQuery will keep the materialized view refreshed and up to date with the contents of the source table.

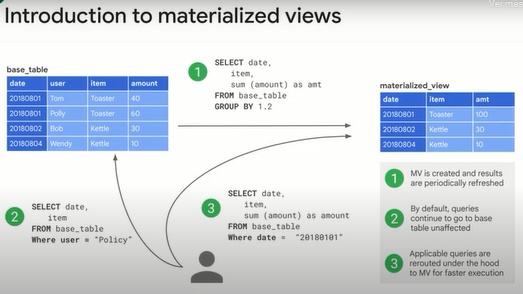
In BigQuery, materialized views periodically cache the results of a query for increased performance and efficiency.

BigQuery leverages pre-computed results from materialized views, and whenever possible, reads only delta changes from the base table to compute up to date results.

Materialized views can be queried directly, or can be used by the BigQuery optimizer to process queries to the base tables.

Queries that use materialized views are generally faster and consume fewer resources than queries that retrieve the same data only from the base table.

Materialized views can significantly improve the performance of workloads that have the characteristic of common and repeated queries.



In the queries we saw earlier, we wrote the query in SQL and selected run on the UI.

What this did was to submit a query job to the BigQuery service.

The BigQuery query service is separate from the BigQuery storage service.

However, they are designed to collaborate and be used together.

In this case, we were querying native tables in the BigQuery public data project.

Querying native tables is the most common case and is the most performant way to use BigQuery.

BigQuery is most efficient when working with data contained in its own storage service.

The storage service and the query service work together to internally organize the data to make queries efficient over huge datasets of terabytes and petabytes in size.

The query service can also run query jobs on data contained in other locations, such as tables and CSV files hosted in Cloud Storage.

So you can query data in external tables or from external sources without loading it into BigQuery.

These are called federated queries.

In either case, the query service puts the results into a temporary table, and the user interface pulls and displays the data in the temporary table.

This temporary table is stored for 24 hours.

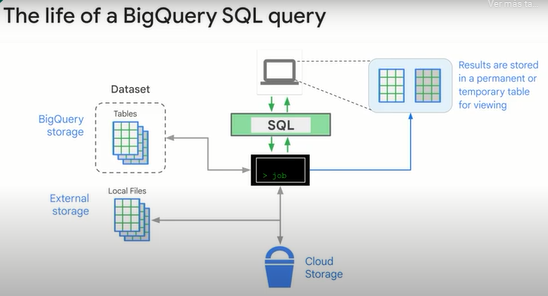
So if you run the exact same query again, and if the results would not be different, then BigQuery will simply return a pointer to the cached results.

Queries that can be served from the cache do not incur any charges.

It is also possible to request that the query job write to a destination table.

In that case, you get to control when the table is deleted.

Because the destination table is permanent and not temporary, you will get charged for the storage of the results.



To calculate pricing, you can use BigQuery's query validator in combination with the pricing calculator for estimates.

The query validator provides an estimate of the size of data that will be processed during a query.

You can plug this into the calculator to find an estimate of how much running the query will cost.

This is valid if you are using an on-demand plan where you pay for each query based on how much data is processed by that query.

Your company might have opted for a flat rate plan.

In that case, your company will be better paying a fixed price, and so the cost is really how many slots your query uses.

You can separate the cost of storage and the cost of queries.

By separating projects A and B, it's possible to share data without giving access to run jobs.

In this diagram, users one and two have access to run jobs and access the datasets in their own respective projects.

If they run a query, that job is built to their own project.

What if user one needs the ability to access dataset D in Project B?

The person who owns project B can allow user one to query project B dataset D and the charges will go to project A when executed from Project A. The public dataset project owner granted all authenticated user's access to use their data.

The special setting all authenticated users makes a dataset public.

Authenticated users must use BigQuery within their own project and have access to run BigQuery jobs so that they can query the public dataset.

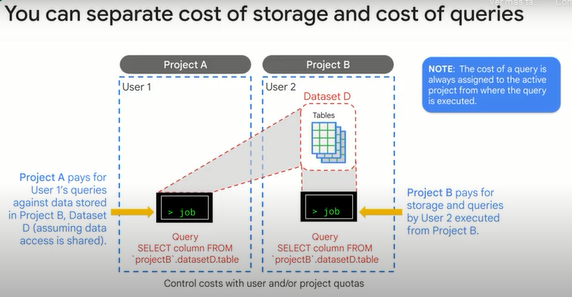
The billing for the query goes to their project, even though the query is using public or shared data.

In summary, the cost of a query is always assigned to the active project from where the query is executed.

The active project for user is displayed at the top of the Cloud console or set by an environmental variable in the Cloud Shell or client tools.

Note, BigQuery offers one TB of querying for free every month.

So public datasets are an easy way to try out BigQuery.



**Load data into BigQuery.**

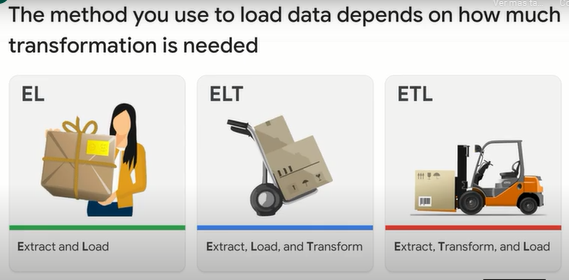
Next, we'll talk about how to load new data into BigQuery.

Recall from an earlier module that the method you use to load data depends on how much transformation is needed.

EL or Extract and Load is used when data is imported as is where the source and target have the same schema.

ELT or Extract-Load-Transform, is used when raw data will be loaded directly into the target and transformed there.

ETL or Extract-Transform-Load is used when transformation occurs in an intermediate service before it is loaded into the target.



You might say that the simplest case is EL.

If the data is usable in its original form, there's no need for transformation, just load it.

You can batch load data into BigQuery.

In addition to CSV, you can also use data files with delimiters other than commas by using the field underscore delimiter flag.

BigQuery supports loading G zip compressed files.

However, loading compressed files isn't as fast as loading uncompressed files.

For time sensitive scenarios or scenarios in which transferring uncompressed files to Cloud Storage is bandwidth or time constraint, conduct a quick loading test to see which alternative works best.

Because load jobs are asynchronous, you don't need to maintain a client connection while the job is being executed.

More importantly, load jobs don't affect your other BigQuery resources.

A load job creates a destination table if one doesn't already exist.

BigQuery determines the data schema as follows.

If your data is in Avro format, which is self describing, BigQuery can determine the schema directly.

If the data is in JSON or CSV format, BigQuery can auto detect the schema, but manual verification is recommended.

You can specify a schema explicitly by passing the schema as an argument to the load job.

Ongoing load jobs can append to the same table using the same procedure as the initial load, but do not require the schema to be passed with each job.

If your CSV files always contain a header row that should be ignored after the initial load and table creation, you can use the skip underscore leading underscore rows flag to ignore the row.

For details, see the documentation on BigQuery load flags dot Avro.

BigQuery sets daily limits on the number and size of load jobs that you can perform per project and per table.

In addition, BigQuery sets limits on the sizes of individual load files and records.

You can launch loads jobs through the BigQuery web UI.

To automate the process, you can set up Cloud Functions to listen to a Cloud storage event that is associated with new files arriving in a given bucket and launch a BigQuery load job.



BigQuery can import data stored in the JSON file format as long as it is new line delimited.

It can also import files in Avro, parquet, and ORC format.

The most common import is with CSV files, which are the bridge between BigQuery and spreadsheets.

BigQuery can also directly import FireStore onto DataStore export files.

Another way that BigQuery can import data is through the API.

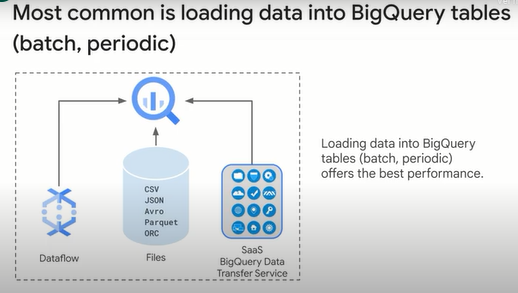
Basically, any place where you can get code to run can theoretically insert data into BigQuery tables.

You could use the API from a Compute Engine instance, a container on Kubernetes App Engine or from Cloud Functions.

However, you would have to recreate the data processing foundation in these cases.

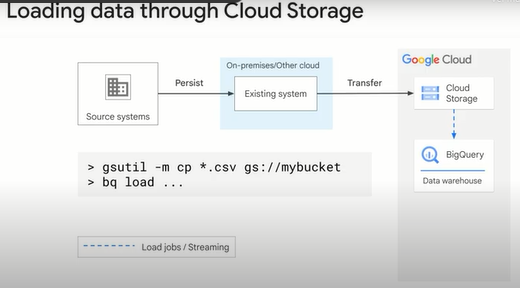
In practice, the API is mainly used from either Data Proc or Dataflow.

The BigQuery Data Transfer Service provides connectors and pre-built BigQuery load jobs that perform the transformations necessary to load report data from various services directly into BigQuery.



Cloud storage can be useful in the EL process.

You can transfer files to Cloud storage in the schema that is native to the existing on-premises data storage, and then load those files into BigQuery.

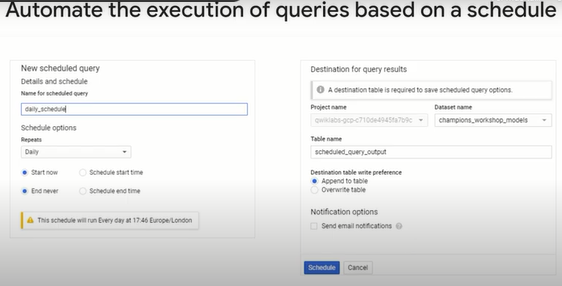


It is a common practice to automate execution of queries based on a schedule or event, and cache the results for later consumption.

You can schedule queries to run on a recurring basis.

Scheduled queries must be written in standard SQL, which can include data definition language, and data manipulation language statements.

The query string and destination table can be parameterized allowing you to organize query results by date and time.



By maintaining a complete seven-day history of changes against your tables, BigQuery allows you to query a point in time snapshot of your data.

You can easily revert changes without having to request a recovery from backups.

This slide shows how to do a select query to query the table as of 24 hours ago.

Because this is a select query, you can do more than just restore a table.

You can join against some other table or correct the value of individual columns.

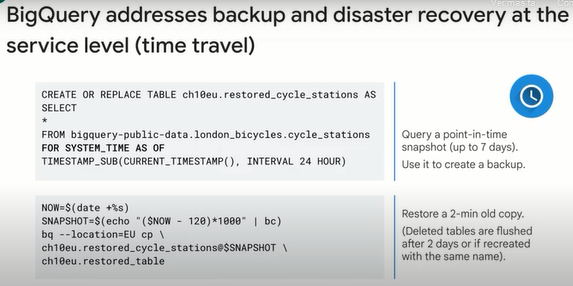
You can also do this using the BigQuery command line tool as shown in the second snippet.

Here, we're restoring data as of 120 seconds ago.

You can recover a deleted table only if another table with the same ID in the dataset has not been created.

In particular, this means you cannot recover a deleted table if it is being streamed to, chances are that the streaming pipeline would have already created an empty table and started pushing rows into it.

Also be careful using Create or Replace table because this makes the table irrecoverable.



BigQuery is a managed service, so you don't have the overhead of operating, maintaining or securing the system.

A typical data warehouse system requires a lot of code for coordination and interfacing.

You can get BigQuery data transfer service running without coding.

The core of BigQuery Data Transfer Service is scheduled and automatic transfers of data from wherever it is located, in your data center, on other Clouds, in SAS services, into BigQuery.

Transferring the data is only the first part of building a data warehouse.

If you were assembling your own system, you would need to stage the data so that it can be cleaned, data quality, and transformed ELT, Extract-Load-Transform, and processed, put into its final and stable form.

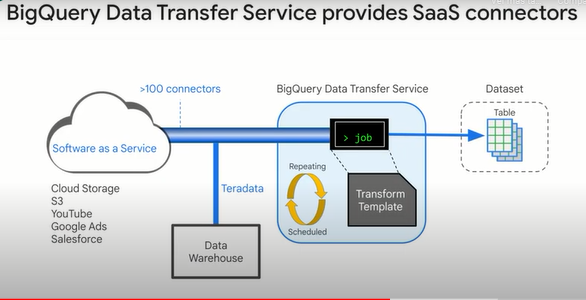
A common issue with data warehouse systems is late arriving data.

For example, a cash register closes late and does not report its daily receipts during the scheduled transfer period.

To complete the data, you would need to detect that not all of the data was received, and then request the missing data to fill in the gap.

This is called data backfill, and it is one of the automatic processes provided by BigQuery Data Transfer Service.

Backfilling data means adding missing past data to make a dataset complete with no gaps and to keep all analytic processes working as expected.



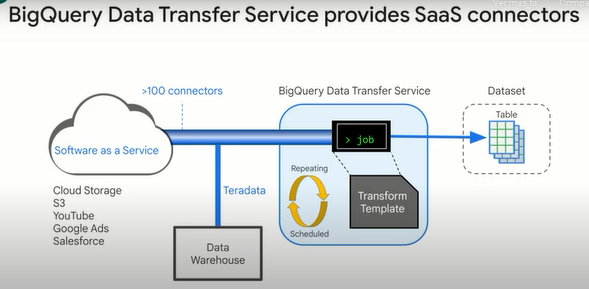
Use the Data Transfer Service for repeated, periodic, scheduled imports of data directly from software as a service systems into tables in BigQuery.

The BigQuery Data Transfer Service provides connectors, transformation templates and the scheduling.

The connectors establish secure communications with the source service and collect standard data exports and reports.

This information is transformed within BigQuery.

The transformations can be quite complicated, resulting in from 25 to 60 tables, and the transfer can be scheduled to repeat as frequently as once a day.

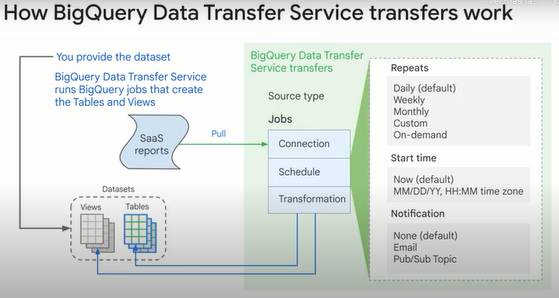


The BigQuery Data Transfer Service can also be used to efficiently move data between regions.

Notice that you don't need Cloud storage buckets.

BigQuery Data Transfer Service runs BigQuery jobs that transform reports from SAS sources into BigQuery tables and views.

Google offers several connectors including Campaign Manager, Cloud Storage, Amazon S3, Google Ad Manager, Google Ads, Google Play Transfers, YouTube channel, YouTube content owner, Teradata migration and over 100 other connectors through partners.



Keep in mind, if your data transformations are simple enough, you may be able to do them with just SQL.

If Cloud Storage is part of your workflow, you can load files from Cloud storage into staging tables in BigQuery first, and then transform the data into the ideal schema for BigQuery by using BigQuery SQL commands.

BigQuery supports standard DML statements such as insert, update, delete, and merge.

There are no limits on DML statements.

However, you should not treat BigQuery as an OLTP system.

The underlying infrastructure is not structured to perform optimally as an OLTP.

There are other more appropriate products on Google Cloud for such workloads.

BigQuery also supports DDL statements like Create or Replace table.

In the example on this slide, the Replace statement is used to transform a string of genres into an array.

We'll cover arrays in greater detail later in the course.

Lastly, what if your transformations went beyond what functions were currently available in BigQuery?

Well, you can create your own.

BigQuery supports user-defined functions or UDF.

A UDF enables you to create a function using another SQL expression or an external programming language.

Java Script is currently the only external language supported.

We strongly suggest you use standard SQL though, because BigQuery can optimize the execution of SQL much better than it can for JavaScript.

UDFs allow you to extend the built-in SQL functions.

UDFs take a list of values, which can be arrays or structs, and return a single value, which can also be an array or a struct.

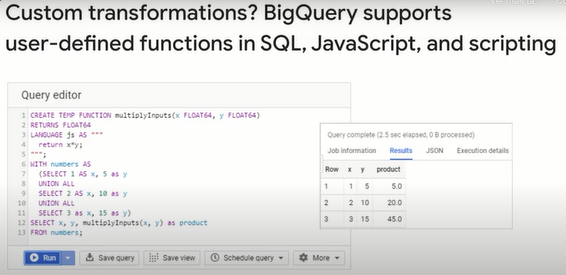
UDFs written in JavaScript can include external resources, such as encryption, or other libraries.

Previously, UDFs were temporary functions only.

This meant you could only use them for the current query or command line session.

Now we have permanent functions, scripts and procedures in beta, but they might even be generally available by the time you are seeing this.

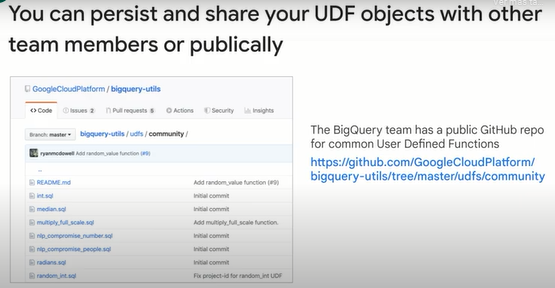
Please check the documentation.



When you create a UDF, BigQuery persists it and stores it as an object in your database.

What this means is you can share your UDFs with other team members or even publicly if you wanted to.

The BigQuery team has a public GitHub repo for common user-defined functions at the link you see here.



**Lab: Loading data into BigQuery.**

Open BigQuery Console

In the Google Cloud Console, select Navigation menu > BigQuery.

The Welcome to BigQuery in the Cloud Console message box opens. This message box provides a link to the quickstart guide and lists UI updates.

Click Done.

Create a new dataset to store tables

To create a dataset, click on the View actions icon next to your project ID and select Create dataset.

Next, name your Dataset ID nyctaxi and leave all other options at their default values, and then click Create dataset.

You'll now see the nyctaxi dataset under your project name.

Click Check my progress to verify the objective.

Creating a dataset to store new tables

Check my progress

Ingest a new Dataset from a CSV

In this section, you will load a local CSV into a BigQuery table.

Download a subset of the NYC taxi 2018 trips data locally onto your computer from here :

In the BigQuery Console, Select the nyctaxi dataset then click Create Table

Specify the below table options:

Source:

Create table from: Upload

Choose File: select the file you downloaded locally earlier

File format: CSV

Destination:

Table name: 2018trips Leave all other setting at default.

Schema:

Check Auto Detect (tip: Not seeing the checkbox? Ensure the file format is CSV and not Avro)

Advanced Options

Leave at default values

Click Create Table.

You should now see the 2018trips table below the nyctaxi dataset.

Select the 2018trips table and view details:

How many rows are in the table?

1,200

10,018

1,090

900

Select Preview and confirm all columns have been loaded (sampled below):

You have successfully loaded in a CSV file into a new BigQuery table.

Running SQL Queries

Next, practice with a basic query on the 2018trips table.

In the Query Editor, write a query to list the top 5 most expensive trips of the year:

#standardSQL

SELECT

\*

FROM

nyctaxi.2018trips

ORDER BY

fare\_amount DESC

LIMIT 5

Copied!

content\_copy

What was the highest fare amount in the year?

339

250

300

Click Check my progress to verify the objective.

Ingest a new Dataset from a CSV

Check my progress

Ingest a new Dataset from Google Cloud Storage

Now, lets try load another subset of the same 2018 trip data that is available on Cloud Storage. And this time, let's use the CLI tool to do it.

In your Cloud Shell, run the following command :

bq load \

--source\_format=CSV \

--autodetect \

--noreplace \

nyctaxi.2018trips \

gs://cloud-training/OCBL013/nyc\_tlc\_yellow\_trips\_2018\_subset\_2.csv

Note: With the above load job, you are specifying that this subset is to be appended to the existing 2018trips table that you created above.

When the load job is complete, you will get a confirmation on the screen.

Back on your BigQuery console, select the 2018trips table and view details. Confirm that the row count has now almost doubled.

You may want to run the same query like earlier to see if the top 5 most expensive trips have changed.

Click Check my progress to verify the objective.

Ingest a dataset from google cloud storage

Check my progress

Create tables from other tables with DDL

The 2018trips table now has trips from throughout the year. What if you were only interested in January trips? For the purpose of this lab, we will keep it simple and focus only on pickup date and time. Let's use DDL to extract this data and store it in another table

In the Query Editor, run the following CREATE TABLE command :

#standardSQL

CREATE TABLE

nyctaxi.january\_trips AS

SELECT

\*

FROM

nyctaxi.2018trips

WHERE

EXTRACT(Month

FROM

pickup\_datetime)=1;

Now run the below query in your Query Editor find the longest distance traveled in the month of January:

#standardSQL

SELECT

\*

FROM

nyctaxi.january\_trips

ORDER BY

trip\_distance DESC

LIMIT

1

Click Check my progress to verify the objective.

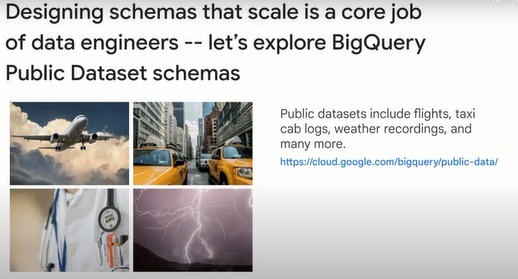
Create tables from other tables with DDL

Check my progress

Congratulations!

You've successfully created a new dataset and ingested data into BigQuery from CSV, Google Cloud Storage, and other BigQuery tables

**Explore schemas.**



**Demo: Exploring Schemas.**

Welcome back to another BigQuery demo. In this one, it's a little bit of a meta demo, meaning we're going to be using a lot of BigQuery metadata that's stored inside of BigQuery native storage, and some unique functions like information schema, and tables to explore the metadata of a dataset that's given to you.

Why is this important? Well, as a data engineer, you're often given a dataset or tables within a dataset or multiple datasets as part of a GCP project, and you'd have a very short amount of time to figure out a lot of key information about those tables. How many tables are there? How many columns are there?

Are any of those columns partitioned or clustered columns? What's the data size? When were the tables last updated? And yes, you can go in and click on the UI and individual places inside of BigQuery to find that information, but it wouldn't be amazing to use SQL to query information about the metadata

that already exists inside of BigQuery. So we're going to do that with a couple quick queries here, but first, we need a dataset. As usual, we're going to be going to the BigQuery public datasets and getting some information there. So I'm going to copy the first query inside of there.

And inside of BigQuery, we're going to paste the query. Now, if you notice, I have my project BigQuery public data, and then I have a table. I'm just using- there's many datasets inside of BigQuery public data. I'm using the baseball dataset, which has multiple tables. If you notice, I'm using this suffix underscore

underscore tables, which is interesting. So if you hold down the Command key on your Macs, if you have a Mac, or use a shortcut there for the Windows, you'll see that for tables, if I'm just bringing this up, it has some interesting metadata. So it has your project, your dataset, your table ID

and it has some short but useful information about when the table was created, when it was last modified or updated, how many rows it has, you don't need to do this select count star, what's the size of the table in bytes, and whether or not it's a table or a view.

I think table is one, view is two. Now, I already have the- the whole point of this query is because this is unreadable to me, it's just milliseconds from the dawn of time or Unix epoch or something like that. And the size in bytes, I can't do the conversion in my head.

So that's literally all the rest of my query does here for you, is it just as a little bit of rounding from bytes to gigabytes, from the millisecond timestamp to the actual readable timestamp. So actually running that query will just translate those results into something that's a little bit more readable.

So we have three tables inside of our baseball dataset, we have games wide, games post wide, and some baseball schedules. You can see the largest table here is about two gigabytes, and it has 761,000 rows. I can infer without much analysis that there are over almost 1,700,000 baseball games

that we can have analysis on, immediately right off the bat, no select star queries or anything like that. All right, so that's just information about it, and again, you can just replace baseball with say, New York. And then boom, you can get all the information about the New York data tables that are in there, a lot more tables.

And you can see that there is a Yellow Taxi Cab trips, 311 service requests for New Yorkers, a lot of motor vehicle collisions. So New York has a lot of tables in that dataset for BigQuery public data. And you can see the row count in there as well, large, very large datasets.

But literally, you're just plugging in your project, your dataset, and you get a lot of metadata about the tables. So it's comparing information across tables very easily. So now if we're going to drill in to the columns of data, how many columns of data are present in one of those tables?

Again, I'm just going to be using baseball, for example, but you can replace that with one that's interesting to you. Here, we're using information schema dot columns. Executing that, let's get all that data. All right, here we go. So we have the table, which is baseball, that's the dataset it belongs to, the table name is games wide.

Don't forget, we also have a couple more tables in this dataset, we're not doing a Where clause filter. So we have 306 columns in total. So no kidding, that's a very wide table. But hopefully, you'll be able to see games wide, there should be two other tables, games post wide,

and there was another one there as well. So if you wanted to do a filter for just a single table, you could see, let's just do a quick Where clause filter, where the actual table name is just this one singular table, games wide. And then we'll see just simply by row count,

this table has 145 columns in it. Wow, that is a wide table. No kidding. And then the ordering position of the columns, whether or not it's allowed to be No, the data type, is it a generative column? Is it hidden? Here's really interesting, as a data engineer, is this a-

is there a partitioning on that table itself? And if there is partitioning, is there clustering as well, which is super interesting. So that's literally what this next query is going to do for us, is just a very simple Where clause basically to say, "Hey, for performance reasons, one of the easiest things that you can do is just have a partitioned

and clustered columns in very large datasets if your use case warrants it." Let's see if there are any in this particular dataset. So I'm going to run this. And it's going to look again filtering on the partition column. And then no. So it's very easy, look through all of the different

tables within that baseball dataset, and then you got nothing. Let's see if the New York one has any in it as well. No, no partitioned, no clustered columns there. So again, some immediate insight that you can give back is, "Hey, let's explore the benefit of partitioned or clustered columns."

All right, the last query that I'm going to show you here is, you can basically get metadata across the different datasets just by doing a simple union all. I've chosen about 10 or 15 interesting ones from the BigQuery public datasets. And I want to see, we looked at the first table,

three tables inside of the baseball dataset, I want to basically list all the tables from all these different datasets. And I want to order it by the dataset that has the table with the most rows. So let's see which table out of all of these has the most row,

just pasting in that query. I think it'll probably be those taxi transactions. So let's see. Getting all that metadata together- Oh, I was wrong. Wow. So we have- and again, you can change this with the timestamp seconds query that I had a little bit earlier. It's actually the GitHub repository,

their files inside of GitHub, and the row count- Wow, can I buy a comma?- is 2.3- is that billion? Yeah, 2.3 billion. And you can see the size that are on disk there as well. So it's GitHub, GitHub, GitHub. Wikipedia is up there. It's only number two, though.

And then here's my New York taxi cabs there as well. And again, this is just unioning together, unioning again inside of SQLs mashing together rows vertically, whereas joins is joining together columns horizontally, and it just mashes them together here as well. Lastly, I'll show you a bonus query, I'm not going to run it for you, though,

is if you wanted to, you want to view all the datasets within your given GCP project, actually, a lot, I will run this for you because it looks pretty cool. I'm going to copy this. And I want to see, let's see if- data to insights, this is a public-

at least the tables within here are public. Let's see if we can- So all I'm doing is basically saying, if somebody just gives you the project name, just the project name, you want to query all the datasets they're within. It looks like yeah, we don't have- it's permissions denied.

So we can use our own project. I have a couple of tables inside of here, so I'm copying the project name. Hopefully, I'm admin for my own project. And essentially, what you're going to see is I've just a bunch of example datasets from previous demos. And you'll see from, for example,

you'll have all of the different datasets returned, and then within there, all the tables and views inside of them. And then let's see which ones have been modified most recently, last modified time descending. So let's go ahead and run that. Again, you would just replace your project name inside of there as well.

So the use case is somebody gives you a GCP account, and you're like, "Well, I really want to just look at what came up inside of here." All right, cool. So we have, it looks like I was doing some machine learning stuff on movie recommendations data most recently. Yep.

And I actually get the dataset description, and it's been around for 39 days last modified in there as well. So you can get a general sense for all of the- and this is- I think this is only ones that actually have a description for it- all of the different dataset values and table values within a given project,

assuming you have access to query it. And that's what information schema dot schemata does, and you can get schemata options. A little bit more advanced but that's also there for you as well. A really interesting use case is all the way at the bottom, let me get this on screen.

All the way at the bottom, this link right here will show you, hey, if I wanted to recreate- check this out- if I weren't to recreate all the tables in my dataset, so for example, if I wanted to just have something that says, hey, I have a staging environment or a production environment,

I need to recreate all these data tables, or at least the schema for them to populate them with data. And then doing the Create or replace statements, if I have 100 different datasets here, it's awful. Is there a way to programmatically create that? And there is. So you can actually say,

the pre-canned query that's given to you actually uses a Create function, and it respects partitioning and it actually does all of this stuff for you. And it can catch together those Create or Replace table names and the list of the columns. And if it has partitioning and clustering and options

like a description, it will all do that for you. So for example, the output looks like this, you want to get a single dot SQL file that will recreate all of your projects and all of your datasets. Boom, here you go. There's the table one, my dataset population by zip.

Boom, there's the table two, this is the GitHub commits or something like that as well. Why is this useful? Well, it's really a good idea to track your schema changes over time, like if your column definitions change. So generally what I would do on a production project is generate this, check this into version control somewhere,

and anytime there's changes to my schema, I would regenerate this and then check it into version control so I can see how my schema has evolved over time. A little bit more advanced use case but that is out there for you as well.

**Schema design.**

Next we will talk about efficient data warehouse schema design.

Take a look at the original data table here and the normalized data tables which contain the same data.

The data in the original table is organized visually as you might have used merge cells or columns in a spreadsheet.

But if you had to write an algorithm to process the data, how might you approach it?

Access could be by rows, by columns, by rows then columns, and the different approaches would perform differently based on the query.

Also, your method might not be parallelizable.

The original data can be interpreted and stored in many ways in a database.

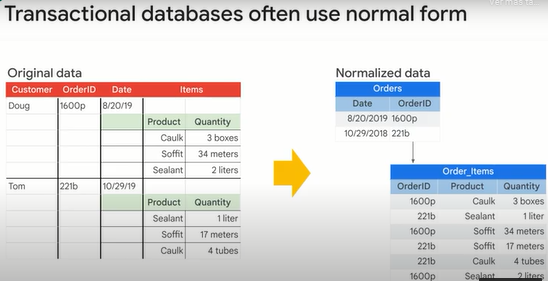
normalizing the data means turning it into a relational system.

This stores the data efficiently and makes query processing a clear and direct task.

Normalizing increases the orderliness of the data, it is useful for saving space.

Many people with database experience will recognize this procedure.

Normalizing data usually happens when a schema is designed for a database.



Denormalizing is the strategy of allowing duplicate field values for a column in a table in the data to gain processing performance.

Data is repeated rather than being relational.

Flattened data takes more storage, but the flattened non-relational organization makes queries more efficient because they can be processed in parallel using columnar processing.

Specifically, denormalizing data enables BigQuery to more efficiently distribute processing among slots resulting in more parallel processing and better query performance.

You would usually denormalize data before loading it into BigQuery.



However, there are cases where denormalizing data is bad for performance.

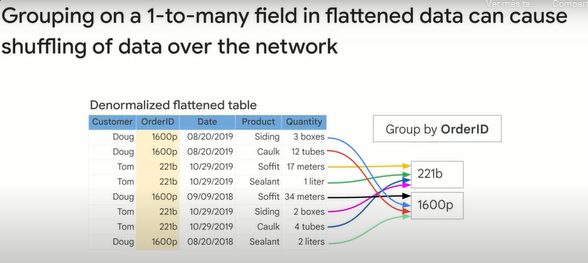
Specifically, if you have to group by a column with a one to many relationship.

In the example shown, order ID is such a column.

In this example, to group the data, it must be shuffled.

That often happens by transferring the data over a network between servers or systems.

Shuffling (“arrastrar”) is slow.



Fortunately, BigQuery supports a method to improve the situation.

BigQuery supports columns with nested and repeated data.

In this example, a denormalized flattened table is compared with one that has been denormalized and the schema takes advantage of nested and repeated fields.

Order ID is a repeated field.

Because this is declared in advance, BigQuery can store and process the data respecting some of the original organization in the data.

Specifically, all order details for each order are co-located, which makes retrieval of the whole order more efficient.

For this reason, nested and repeated fields are useful for working with data that originates in relational databases.

Nested columns can be understood as a form of repeated field.

It preserves the relational qualities of the original data and schema while enabling columnar and parallel processing of the repeated nested fields.

It is the best alternative for data that already has a relational pattern to it.

Turning the relation into a nested or repeated field improves BigQuery Performance.

Nested and repeated fields help BigQuery work with data source in relational databases.

Look for nested and repeated fields whenever BigQuery is used in a hybrid solution in conjunction with traditional databases.



**Nested and repeated fields.**

Let's take a closer look at BigQuery's support for nested and repeated fields and why this is such a popular schema design for enterprises.

I'll illustrate by using an example from a real business running on Google Cloud.

GoJek is a company in Indonesia that is well known for its ride booking service.

And they process over 13 petabytes of data on BigQuery per month from queries to support business decisions.

What kind of decisions?

For GoJek, they track whenever a new customer places an order, like hail a ride with their mobile app, that order is stored in an orders table.

Each order has a single pickup location and drop off destination.

For a single order, you could have one or many events like ride ordered, ride confirmed, drive on route, drop off complete, etcetera.

As a data engineer, how would you efficiently store these different pieces of data in your data warehouse?

Keep in mind, you need to support a large user base querying petabytes per month.

Well, as you saw earlier, we could store one fact in one place with the normalization route, which is typical for relational systems.

Or we could go the fully denormalized route and just store all levels of granularity in a single big table, where you would have one order ID like 123 repeated in a row for each event that happens on that order.

Faster for querying, sure, but what are the drawbacks? (inconvenientes)

For relational schemas, normalized schemas, often the most intensive computational workloads are joins across very large tables.

Remember, RDBMSs are record based, so they have to open each record entirely, and pull out the join key from each table where a match exists.

And that's assuming you know all the tables that need to be joined together.

Imagine for each new piece of information about an order like promotion codes, or user information, and you could be talking about 10 plus table join.

The alternative has different drawbacks.

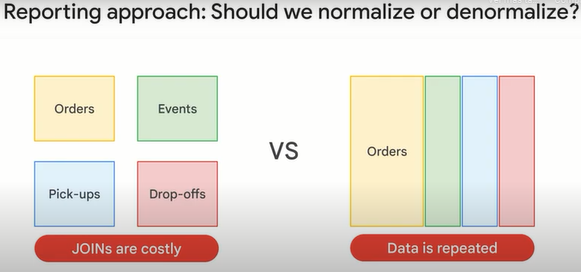
Pre-joining all your tables into one massive table makes reading data faster, but you now have to be really careful if you have data at different levels of granularity.

In our example, each row would be at the level of granularity of a specific event like driver confirmed for a given order.

What does that mean for an order ID like 123?

It is duplicated for each event in that order.

Imagine if you're looking to join higher level information like the revenue per order, and you now have to be exceedingly careful with aggregations to not double or triple count your duplicate order IDs.

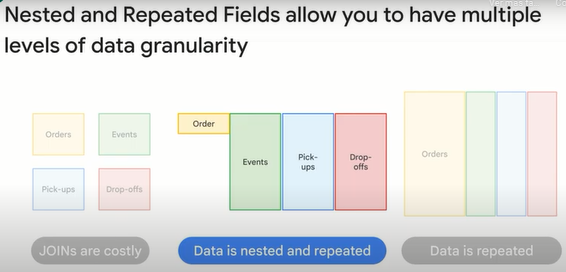


See the problem?

One common solution in enterprise data warehouse schemas is to take advantage of nested and repeated data fields.

You can have one row for each order, and repeated values within that one row for data that is at a more granular level.

For example, you could simply have an array of timestamps as your events.



Let's see an example to illustrate this point.

Here you see it clearly, shown here on screen are just four rows for four unique order IDs.

Notice all that gray space in between the rows.

That's because the event status and event time is at a deeper level of granularity.

That means there are multiple repeated values for these events per each order.

An array is a perfect data type to handle this repeated value and keep all the benefits of storing that data in a single row.

I mentioned the fields event dot status and event dot time.

If this is one giant table, what is a dot doing in those column names?

There are no other table aliases we've joined on.

What's up with those fields.

Event, pickup, and destination are what are called struct or structured data type fields in SQL.

This isn't BigQuery specific.

Structs are standard SQL data types and BigQuery just supports them really well.

Structs you can think of as pre-joined tables within a table.

So instead of having a separate table for event and pickup and destination, you simply nest them within your main table.

So let's recap.

You can go deep into a single field and have it be more granular than the rest by using an array data type like you see here for status and time.

And you can have really wide schemas by using structs which allow you to have multiple fields of the same or different data types within them, much like a separate table would.

The major benefit of structs is that the data is conceptually pre-joined already.

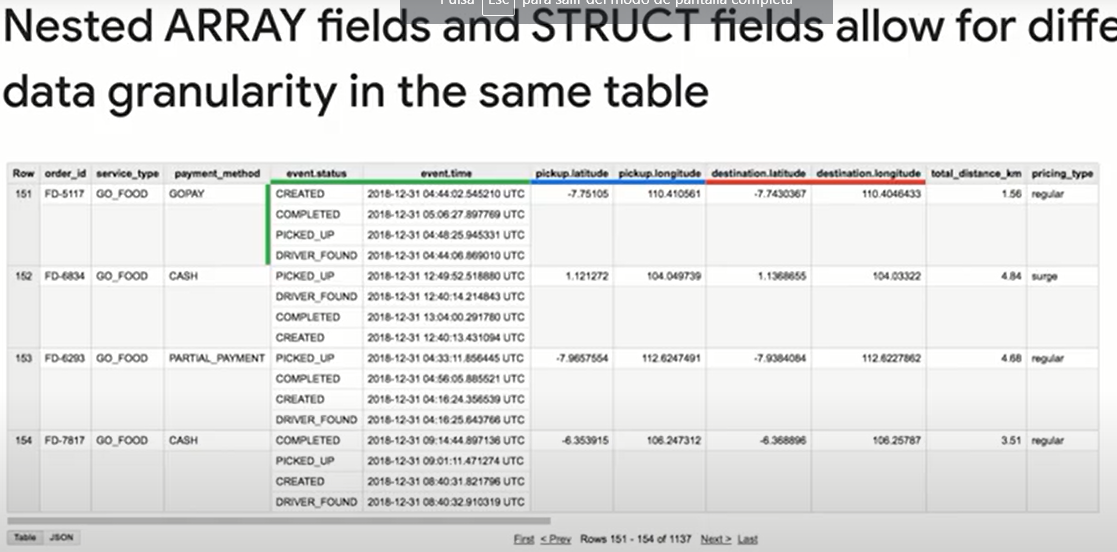
So it's much faster to query.

People often ask, with really wide schemas like 100 columns, how is it still fast to query?

Remember that BigQuery is column based storage, not record based when storing data out on disk.

If you did just account order underscore ID here to get your total orders, BigQuery wouldn't even care that you have 99 other columns, some of which are more granular with array data types, it wouldn't even look at them.

That gives you the best of both worlds if you're an analyst, lots of data all in one place, and no issues with multiple granularity pitfalls when doing aggregations.



Now it's your turn to practice reading one of these schemas that has nested and repeated fields, take a moment and spot those structs.

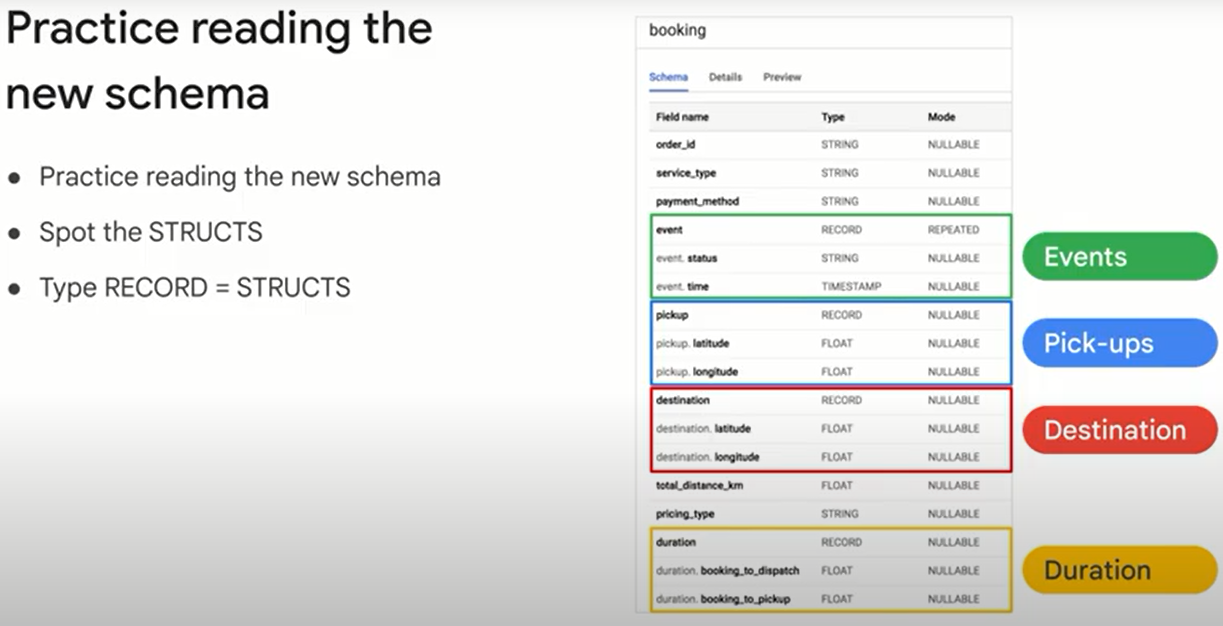
As a hint, you can look at the field name to see any field with a dot in the name or you can look at the data type for any field values of the type record, which means struct.

Did you get them all?

Here are the four strokes in this dataset you saw earlier: events, pickups, destination, and duration.

Duration is a new one, but we can simply keep adding more dimensions to our dataset by adding more structs.

Remember, structs let you build really wide and informative schemas.



Now it's time to go deep.

Find the array data types in this schema.

As a hint, look at the mode and find the repeated values.

Got them?

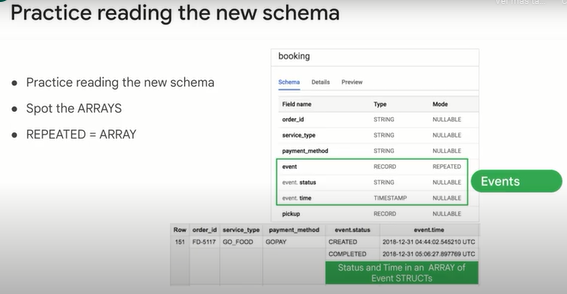
In this schema, the repeated value is the event struct, which means here we have an array of event structs with each having a status and time possibly.

A critical point I like to make here is that struct and array data types in SQL can be absolutely independent of each other.

You can have a regular column in SQL be an array column that has nothing to do with any struct.

Likewise, you can have a struct that has zero array field types in its columns.

The benefit of using them together is that arrays allow a given field to go deep into granularity, and structs allow you to organize all those useful fields into logical containers instead of separate tables.



So here's the cheat sheet.

Structs are a type of record when looking at a schema, and arrays are of mode repeated.

Arrays can be of any single type, like an array of floats, or an array of strings, et cetera.

Arrays can be part of a regular field or be part of a nested field nestled inside of a struct, a single table can have zero to many structs.

And lastly, the real mind bending point is that a struct can have other structs nested inside of it as you will soon see in your upcoming Lab, which uses the real Google Analytics schema.

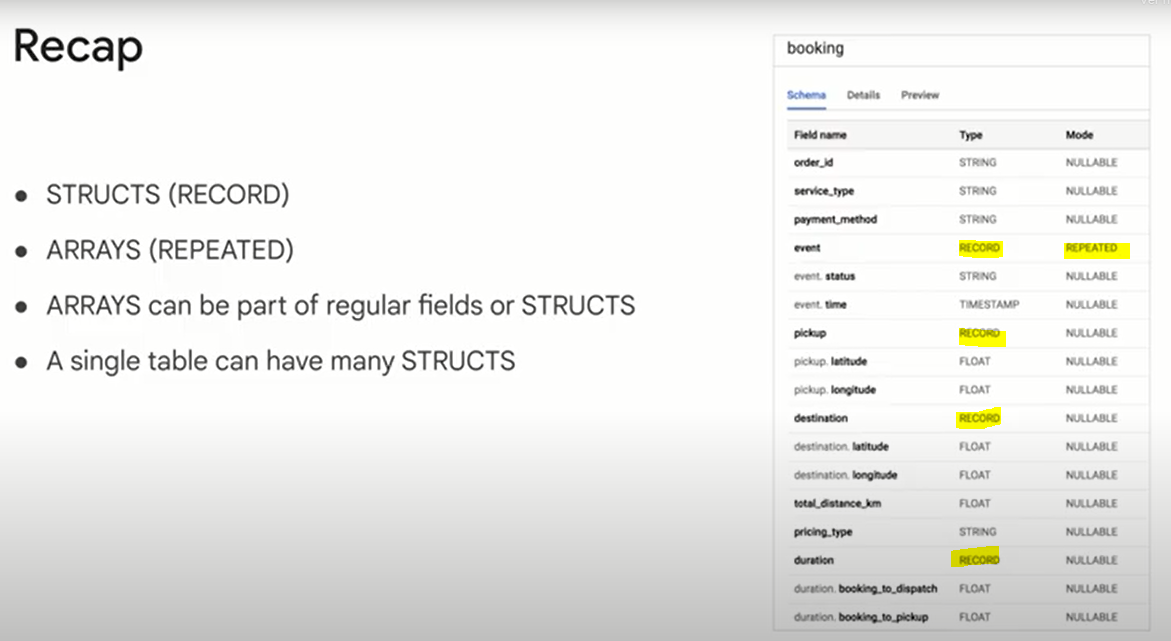
We've been talking a lot about nested and repeated fields.

So you're probably wondering what to do with your existing star schema, snowflake and third normal form data.

The great news is that BigQuery also works well with those schema types.

Use arrays and structs when your data naturally arrives in that format, and you'll benefit immediately from optimal performance.

For the other schema types, bring them directly to BigQuery and you'll likely be pleased with the performance.



**Design the optimal schema for BigQuery.**

Let's recap some of the ways to design the schema of tables to improve query performance and lower query costs.

It's much more efficient to define your schema to use nested repeated fields instead of joins.

Suppose you have orders and purchase items for each order.

In a traditional relational database system, you'd have two tables, one table for purchase items, and another for orders with a foreign key to connect the two tables.

In BigQuery, it's much more efficient if you store each order in a row and have a nested repeated column called purchase\_item.

Arrays are a native type in BigQuery.

Learn to think in terms of arrays.

**When you have dimension tables that are smaller than 10 gigabytes, keep them normalized.**

The exception to this is if the table rarely goes through update and delete operations.

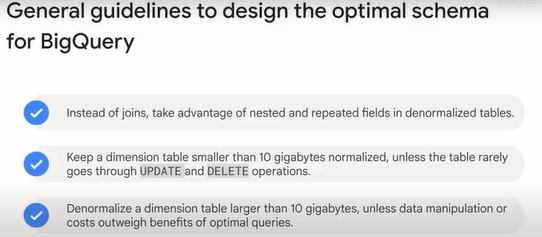
If you cannot define your schema in terms of nested repeated fields, you have to make a decision on whether to keep the data in two tables or denormalize the tables into one big flattened table.

As a datasets tables increase in size, the performance impact of a join increases.

At some point, it can be better to denormalize your data.

The crossover point is around 10 gigabytes.

**If your tables are less than 10 gigabytes, keep the tables separate and do a join.**



**Lab: Working with JSON and Array data in BigQuery 2.5.**

Open BigQuery Console

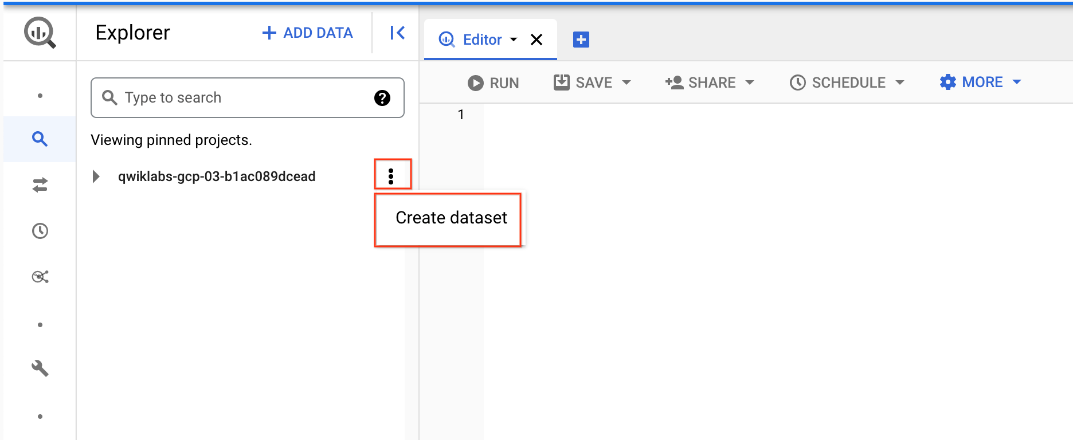
1. In the Google Cloud Console, select **Navigation menu** > **BigQuery**.

The **Welcome to BigQuery in the Cloud Console** message box opens. This message box provides a link to the quickstart guide and lists UI updates.

1. Click **Done**.

**Create a new dataset to store our tables**

1. To create a dataset, click on the **View actions** icon next to your Project ID and then select **Create dataset**.



1. Name the new dataset **fruit\_store**. Leave the other options at their default values (Data Location, Default Table Expiration). Click **Create dataset**.

**Practice working with Arrays in SQL**

Normally in SQL you will have a single value for each row like this list of fruits below:

|  |  |
| --- | --- |
| **Row** | **Fruit** |
| 1 | raspberry |
| 2 | blackberry |
| 3 | strawberry |
| 4 | cherry |

What if you wanted a list of fruit items for each person at the store? It could look something like this:

|  |  |  |
| --- | --- | --- |
| **Row** | **Fruit** | **Person** |
| 1 | raspberry | sally |
| 2 | blackberry | sally |
| 3 | strawberry | sally |
| 4 | cherry | sally |
| 5 | orange | frederick |
| 6 | apple | frederick |

In traditional relational database SQL, you would look at the repetition of names and immediately think to split the above table into two separate tables: Fruit Items and People. That process is called [normalization](https://en.wikipedia.org/wiki/Database_normalization) (going from one table to many). This is a common approach for transactional databases like mySQL.

For data warehousing, data analysts often go the reverse direction (denormalization) and bring many separate tables into one large reporting table.

What are some potential issues if you stored all your data in one giant table?

**The table row size could be too large for traditional reporting databases**

Any changes to a value (like customer email) could impact many other rows (like all their orders)

Data at differing levels of granularity could lead to reporting issues because less granular fields would be repeated.

All of the above

Submit

Now, you're going to learn a different approach that stores data at different levels of granularity all in one table using repeated fields:

|  |  |  |
| --- | --- | --- |
| **Row** | **Fruit (array)** | **Person** |
| 1 | raspberry | sally |
| blackberry |  |
| strawberry |  |
| cherry |  |
| 2 | orange | frederick |
| apple |  |

What looks strange about the previous table?

* It's only two rows.
* There are multiple field values for Fruit in a single row.
* The people are associated with all of the field values.

What the key insight? The array data type!

An easier way to interpret the Fruit array:

|  |  |  |
| --- | --- | --- |
| **Row** | **Fruit (array)** | **Person** |
| 1 | [raspberry, blackberry, strawberry, cherry] | sally |
| 2 | [orange, apple] | frederick |

Both of these tables are exactly the same. There are two key learnings here:

* An array is simply a list of items in brackets [ ]
* BigQuery visually displays arrays as *flattened*. It simply lists the value in the array vertically (note that all of those values still belong to a single row)

1. Try it yourself. Enter the following in the BigQuery Query Editor:

#standardSQL

SELECT

['raspberry', 'blackberry', 'strawberry', 'cherry'] AS fruit\_array

1. Click **Run**.
2. Now try executing this one:

#standardSQL

SELECT

['raspberry', 'blackberry', 'strawberry', 'cherry', 1234567] AS fruit\_array

You should get an error that looks like the following:

Error: Array elements of types {INT64, STRING} do not have a common supertype at [3:1]

Why did we get this error?

**Data in an array [ ] must all be the same type**

Data in an array must only be strings

Data in an array cannot exceed 4 elements

Submit

Arrays can only share one data type (all strings, all numbers).

1. Here's the final table to query against:

#standardSQL

SELECT person, fruit\_array, total\_cost FROM `data-to-insights.advanced.fruit\_store`;

Copied!

content\_copy

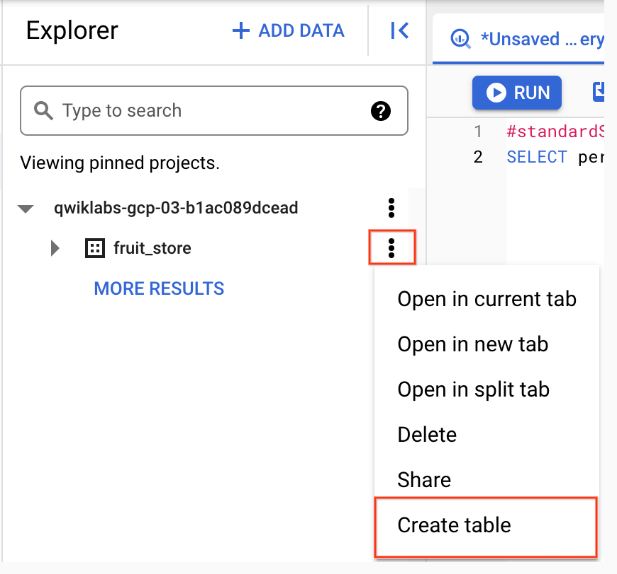
1. Click **Run**.
2. After viewing the results, click the **JSON** tab to view the nested structure of the results.



Loading semi-structured JSON into BigQuery

What if you had a JSON file that you needed to ingest into BigQuery?

1. Create a new table in the fruit\_store dataset.
2. To create a table, click on the **View actions** icon next to the **fruit\_store** dataset and select **Create Table**.



1. Add the following details for the table:

* **Source**: Choose **Google Cloud Storage** in the **Create table from** dropdown.
* **Select file from GCS bucket** (type or paste the following): data-insights-course/labs/optimizing-for-performance/shopping\_cart.json
* **File format**: JSONL (Newline delimited JSON) {This will be auto-populated}
* **Schema**: Check **Auto detect** (Schema and input parameters).

1. Call the new table "fruit\_details".
2. Click **Create table**.

In the schema, note that fruit\_array is marked as REPEATED which means it's an array.

**Recap**

* BigQuery natively supports arrays
* Array values must share a data type
* Arrays are called REPEATED fields in BigQuery

Click *Check my progress* to verify the objective.

Create a new dataset and load JSON data into the table

Check my progress

**Creating your own arrays with ARRAY\_AGG()**

Don't have arrays in your tables already? You can create them!

1. **Copy and Paste** the below query to explore this public dataset

SELECT

fullVisitorId,

date,

v2ProductName,

pageTitle

FROM `data-to-insights.ecommerce.all\_sessions`

WHERE visitId = 1501570398

ORDER BY date

Copied!

content\_copy

1. Click **Run** and view the results

How many rows are returned?

70

2

100

111

Submit

1. Now, we will use the ARRAY\_AGG() function to aggregate our string values into an array. Copy and paste the below query to explore this public dataset:

SELECT

fullVisitorId,

date,

ARRAY\_AGG(v2ProductName) AS products\_viewed,

ARRAY\_AGG(pageTitle) AS pages\_viewed

FROM `data-to-insights.ecommerce.all\_sessions`

WHERE visitId = 1501570398

GROUP BY fullVisitorId, date

ORDER BY date

Copied!

content\_copy

1. Click **Run** and view the results

How many rows are returned?

2 - one for each day

63 - one for each day

70 - one for each day

100 - one for each daySubmit

1. Next, we will use the ARRAY\_LENGTH() function to count the number of pages and products that were viewed.

SELECT

fullVisitorId,

date,

ARRAY\_AGG(v2ProductName) AS products\_viewed,

ARRAY\_LENGTH(ARRAY\_AGG(v2ProductName)) AS num\_products\_viewed,

ARRAY\_AGG(pageTitle) AS pages\_viewed,

ARRAY\_LENGTH(ARRAY\_AGG(pageTitle)) AS num\_pages\_viewed

FROM `data-to-insights.ecommerce.all\_sessions`

WHERE visitId = 1501570398

GROUP BY fullVisitorId, date

ORDER BY date

Copied!

content\_copy

How many pages were visited by this user on 20170801?

8

101

70

109

Submit

1. Next, lets deduplicate the pages and products so we can see how many unique products were viewed. We'll simply add DISTINCT to our ARRAY\_AGG()

SELECT

fullVisitorId,

date,

ARRAY\_AGG(DISTINCT v2ProductName) AS products\_viewed,

ARRAY\_LENGTH(ARRAY\_AGG(DISTINCT v2ProductName)) AS distinct\_products\_viewed,

ARRAY\_AGG(DISTINCT pageTitle) AS pages\_viewed,

ARRAY\_LENGTH(ARRAY\_AGG(DISTINCT pageTitle)) AS distinct\_pages\_viewed

FROM `data-to-insights.ecommerce.all\_sessions`

WHERE visitId = 1501570398

GROUP BY fullVisitorId, date

ORDER BY date

Copied!

content\_copy

How many DISTINCT pages were visited by this user on 20170801?

101

8

109

70

Submit

**Recap**

You can do some pretty useful things with arrays like:

* finding the number of elements with ARRAY\_LENGTH(<array>)
* deduplicating elements with ARRAY\_AGG(DISTINCT <field>)
* ordering elements with ARRAY\_AGG(<field> ORDER BY <field>)
* limiting ARRAY\_AGG(<field> LIMIT 5)

Click *Check my progress* to verify the objective.

Creating arrays with ARRAY\_AGG()

Check my progress

**Querying datasets that already have ARRAYs**

The BigQuery Public Dataset for Google Analytics bigquery-public-data.google\_analytics\_sample has many more fields and rows than our course dataset data-to-insights.ecommerce.all\_sessions. More importantly, it already stores field values like products, pages, and transactions natively as ARRAYs.

1. **Copy and Paste** the below query to explore the available data and see if you can find fields with repeated values (arrays)

SELECT

\*

FROM `bigquery-public-data.google\_analytics\_sample.ga\_sessions\_20170801`

WHERE visitId = 1501570398

Copied!

content\_copy

1. **Run** the query.
2. **Scroll right** in the results until you see the hits.product.v2ProductName field (we will discuss the multiple field aliases shortly).

You will notice a lot of seemingly 'empty' cells in the results as you scroll. These cells are grayed out and not marked as null. Why do you think that is?

The entire dataset has no data values for the grayed out cells

BigQuery is still in the process of loading values for the grayed out cells

The grayed out cells are visual placeholders to make it possible to show each item in an array type column on its own row within the context of a row in the result set

Submit

1. The amount of fields available in the Google Analytics schema can be overwhelming for our analysis. Let's try to query just the visit and page name fields like we did before.

SELECT

visitId,

hits.page.pageTitle

FROM `bigquery-public-data.google\_analytics\_sample.ga\_sessions\_20170801`

WHERE visitId = 1501570398

Copied!

content\_copy

You will get an error: Cannot access field product on a value with type ARRAY> at [5:8]

Before we can query REPEATED fields (arrays) normally, you must first break the arrays back into rows.

For example, the array for hits.page.pageTitle is stored currently as a single row like:

['homepage','product page','checkout']

Copied!

content\_copy

and we need it to be

['homepage',

'product page',

'checkout']

Copied!

content\_copy

1. How do we do that with SQL? Answer: Use the UNNEST() function on your array field:

SELECT DISTINCT

visitId,

h.page.pageTitle

FROM `bigquery-public-data.google\_analytics\_sample.ga\_sessions\_20170801`,

UNNEST(hits) AS h

WHERE visitId = 1501570398

LIMIT 10

Copied!

content\_copy

We'll cover UNNEST() more in detail later but for now just know that:

* You need to UNNEST() arrays to bring the array elements back into rows
* UNNEST() always follows the table name in your FROM clause (think of it conceptually like a pre-joined table)

Click *Check my progress* to verify the objective.

Querying datasets that already have ARRAYs

Check my progress

**Introduction to STRUCTs**

You may have wondered why the field alias hit.page.pageTitle looks like three fields in one separated by periods. Just as ARRAY values give you the flexibility to *go deep* into the granularity of your fields, another data type allows you to *go wide* in your schema by grouping related fields together. That SQL data type is the [STRUCT](https://cloud.google.com/bigquery/docs/reference/standard-sql/data-types#struct-type) data type.

The easiest way to think about a STRUCT is to consider it conceptually like a separate table that is already pre-joined into your main table.

A STRUCT can have:

* one or many fields in it
* the same or different data types for each field
* it's own alias

Sounds just like a table right?

Let's explore a dataset with STRUCTs

1. Under **Explorer** find the **bigquery-public-data** dataset.
2. If it's not present already, click **ADD DATA > Pin a project**.
3. Click **Enter project name**.
4. Enter bigquery-public-data and click **Pin**.
5. Click bigquery-public-data in the pinned project list to expand it.
6. Find and open **google\_analytics\_sample**.
7. Click the **ga\_sessions** table.
8. Start scrolling through the schema and answer the following question by using the find feature of your browser (i.e. CTRL + F). **Tip: Expand all columns before you begin counting.**

In a BigQuery schema, a STRUCT field is noted as a RECORD Type. Search for RECORD in the Google Analytics schema. How many STRUCTs (the RECORD Type columns) are present in this dataset?

1

5

11

**32**

Submit

What are the names of some of the STRUCT (RECORD Type) fields?

Totals

TrafficSource

trafficSource.adwordsClickInfo

device

**All of the above**

Submit

How can both TrafficSource and trafficSource.adwordsClickInfo both be STRUCTs?

**A STRUCT can have another STRUCT as one of its fields (you can nest STRUCTs)**

They are not STRUCTs

Because they are all ARRAYs

This is an invalid data type

Submit

In a BigQuery schema, an ARRAY field is noted as a REPEATED Mode. Search for REPEATED in the Google Analytics schema. How many ARRAYs are present in this dataset?

1

5

**11**

32

Submit

1. As you can imagine, there is an incredible amount of website session data stored for a modern ecommerce website. The main advantage of having 32 STRUCTs in a single table is it allows you to run queries like this one without having to do any JOINs:

SELECT

visitId,

totals.\*,

device.\*

FROM `bigquery-public-data.google\_analytics\_sample.ga\_sessions\_20170801`

WHERE visitId = 1501570398

LIMIT 10

Copied!

content\_copy

Note: The .\* syntax tells BigQuery to return all fields for that STRUCT (much like it would if totals.\* was a separate table we joined against)

Storing your large reporting tables as STRUCTs (pre-joined "tables") and ARRAYs (deep granularity) allows you to:

* gain significant performance advantages by avoiding 32 table JOINs
* get granular data from ARRAYs when you need it but not be punished if you don't (BigQuery stores each column individually on disk)
* have all the business context in one table as opposed to worrying about JOIN keys and which tables have the data you need

Click *Check my progress* to verify the objective.

Explore a dataset with STRUCTs

Check my progress

**Practice with STRUCTs and ARRAYs**

The next dataset will be lap times of runners around the track. Each lap will be called a "split".



1. With this query, try out the STRUCT syntax and note the different field types within the struct container:

#standardSQL

SELECT STRUCT("Rudisha" as name, 23.4 as split) as runner

Copied!

content\_copy

|  |  |  |
| --- | --- | --- |
| **Row** | **runner.name** | **runner.split** |
| 1 | Rudisha | 23.4 |

What do you notice about the field aliases? Since there are fields nested within the struct (name and split are a subset of runner) you end up with a dot notation.

What if the runner has multiple split times for a single race (like time per lap)?

How could you have multiple split times within a single record? Hint: the splits all have the same numeric datatype.

Store each split time in a separate STRING field with STRING\_AGG()

Store each split time in a separate table called race\_splits

Store each split time as an element in an ARRAY of splits

Use a SQL UNION to join the race and split details

Submit

1. With an array of course! Run the below query to confirm:

#standardSQL

SELECT STRUCT("Rudisha" as name, [23.4, 26.3, 26.4, 26.1] as splits) AS runner

Copied!

content\_copy

|  |  |  |
| --- | --- | --- |
| **Row** | **runner.name** | **runner.splits** |
| 1 | Rudisha | 23.4 |
| 26.3 |
| 26.4 |
| 26.1 |

To recap:

* Structs are containers that can have multiple field names and data types nested inside.
* An arrays can be one of the field types inside of a Struct (as shown above with the splits field).

Practice ingesting JSON data

1. Create a new dataset titled **racing**.
2. Create a new table titled **race\_results**.
3. Ingest this Google Cloud Storage JSON file:

data-insights-course/labs/optimizing-for-performance/race\_results.json

* **Source**: Google Cloud Storage under **Create table from** dropdown.
* **Select file from GCS bucket**: data-insights-course/labs/optimizing-for-performance/race\_results.json
* **File format**: JSONL (Newline delimited JSON)
* In **Schema**, move the **Edit as text** slider and add the following:

[

{

"name": "race",

"type": "STRING",

"mode": "NULLABLE"

},

{

"name": "participants",

"type": "RECORD",

"mode": "REPEATED",

"fields": [

{

"name": "name",

"type": "STRING",

"mode": "NULLABLE"

},

{

"name": "splits",

"type": "FLOAT",

"mode": "REPEATED"

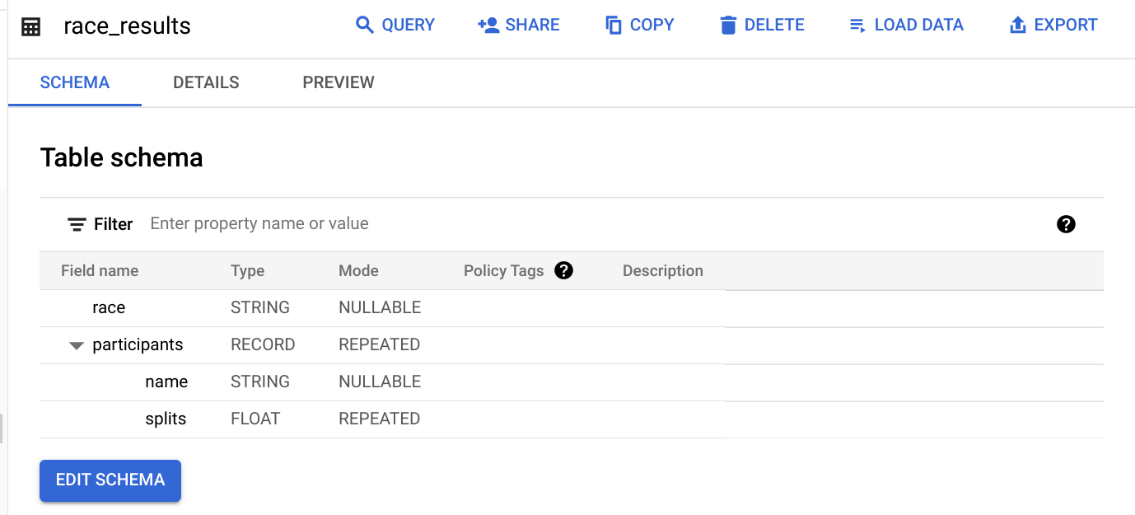
}

]

}

]

1. Click **Create table**.
2. After the load job is successful, preview the schema for the newly created table:



Which field is the STRUCT? How do you know?

The **participants** field is the STRUCT because it is of type RECORD

Which field is the ARRAY?

The participants.splits field is an array of floats inside of the parent participants struct. It has a REPEATED Mode which indicates an array. Values of that array are called nested values since they are multiple values inside of a single field.

Practice querying nested and repeated fields

1. Let's see all of our racers for the 800 Meter race.

#standardSQL

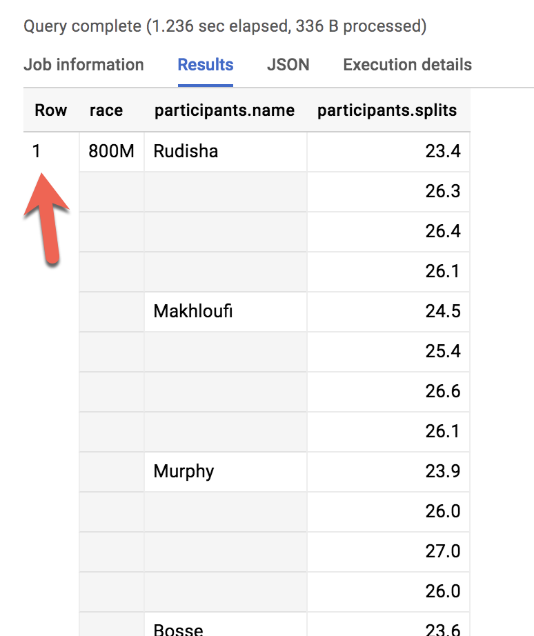
SELECT \* FROM racing.race\_results

Copied!

content\_copy

How many rows were returned?

Answer: 1



1. What if you wanted to list the name of each runner and the type of race?

Run the below schema and see what happens:

#standardSQL

SELECT race, participants.name

FROM racing.race\_results

Copied!

content\_copy

Error: Cannot access field name on a value with type ARRAY\<STRUCT\<name STRING, splits ARRAY\<FLOAT64\>>>> at [1:21]

Much like forgetting to GROUP BY when you use aggregation functions, here there are two different levels of granularity. One row for the race and three rows for the participants names. So how do you change this...

|  |  |  |
| --- | --- | --- |
| **Row** | **race** | **participants.name** |
| 1 | 800M | Rudisha |
| 2 | ??? | Makhloufi |
| 3 | ??? | Murphy |

...to this:

|  |  |  |
| --- | --- | --- |
| **Row** | **race** | **participants.name** |
| 1 | 800M | Rudisha |
| 2 | 800M | Makhloufi |
| 3 | 800M | Murphy |

In traditional relational SQL, if you had a races table and a participants table what would you do to get information from both tables? You would JOIN them together. Here the participant STRUCT (which is conceptually very similar to a table) is already part of your races table but is not yet correlated correctly with your non-STRUCT field "race".

Can you think of what two word SQL command you would use to correlate the 800M race with each of the racers in the first table?

Answer: CROSS JOIN

1. Great! Now try running this:

#standardSQL

SELECT race, participants.name

FROM racing.race\_results

CROSS JOIN

participants # this is the STRUCT (it's like a table within a table)

Copied!

content\_copy

Error: Table name "participants" cannot be resolved: dataset name is missing.

Even though the participants STRUCT is like a table, it is still technically a field in the racing.race\_results table.

1. Add the dataset name to the query:

#standardSQL

SELECT race, participants.name

FROM racing.race\_results

CROSS JOIN

race\_results.participants # full STRUCT name

Copied!

content\_copy

1. Click **Run**.

Wow! You've successfully listed all of the racers for each race!

|  |  |  |
| --- | --- | --- |
| **Row** | **race** | **name** |
| 1 | 800M | Rudisha |
| 2 | 800M | Makhloufi |
| 3 | 800M | Murphy |
| 4 | 800M | Bosse |
| 5 | 800M | Rotich |
| 6 | 800M | Lewandowski |
| 7 | 800M | Kipketer |
| 8 | 800M | Berian |

You can simplify the last query by:

* Adding an alias for the original table
* Replacing the words "CROSS JOIN" with a comma (a comma implicitly cross joins)

This will give you the same query result:

#standardSQL

SELECT race, participants.name

FROM racing.race\_results AS r, r.participants

Copied!

content\_copy

If you have more than one race type (800M, 100M, 200M), wouldn't a CROSS JOIN just associate every racer name with every possible race like a cartesian product?

**Answer**: No. This is a *correlated* cross join which only unpacks the elements associated with a single row. For a greater discussion, see [working with ARRAYs and STRUCTs](https://cloud.google.com/bigquery/docs/reference/standard-sql/arrays#flattening-arrays)

Click *Check my progress* to verify the objective.

Practice with STRUCTs and ARRAYs

Check my progress

Recap of STRUCTs:

* A SQL [STRUCT](https://cloud.google.com/bigquery/docs/reference/standard-sql/data-types#struct-type) is simply a container of other data fields which can be of different data types. The word struct means data structure. Recall the example from earlier:
* \_\_STRUCT(\_\_"Rudisha" as name, [23.4, 26.3, 26.4, 26.1] as splits\_\_)\_\_AS runner
* STRUCTs are given an alias (like runner above) and can conceptually be thought of as a table inside of your main table.
* STRUCTs (and ARRAYs) must be unpacked before you can operate over their elements. Wrap an UNNEST() around the name of the struct itself or the struct field that is an array in order to unpack and flatten it.

**Lab Question: STRUCT()**

Answer the below questions using the racing.race\_results table you created previously.

**Task:** Write a query to COUNT how many racers were there in total.

To start, use the below partially written query:

#standardSQL

SELECT COUNT(participants.name) AS racer\_count

FROM racing.race\_results

Copied!

content\_copy

**Hint:** Remember you will need to cross join in your struct name as an additional data source after the FROM.

Possible Solution:

#standardSQL

SELECT COUNT(p.name) AS racer\_count

FROM racing.race\_results AS r, UNNEST(r.participants) AS p

Copied!

content\_copy

|  |  |
| --- | --- |
| **Row** | **racer\_count** |
| 1 | 8 |

Answer: There were 8 racers who ran the race.

**Lab Question: Unpacking ARRAYs with UNNEST( )**

Write a query that will list the total race time for racers whose names begin with R. Order the results with the fastest total time first. Use the UNNEST() operator and start with the partially written query below.

Complete the query:

#standardSQL

SELECT

p.name,

SUM(split\_times) as total\_race\_time

FROM racing.race\_results AS r

, r.participants AS p

, p.splits AS split\_times

WHERE

GROUP BY

ORDER BY

;

Copied!

content\_copy

Hint:

* You will need to unpack both the struct and the array within the struct as data sources after your FROM clause
* Be sure to use aliases where appropriate

Possible Solution:

#standardSQL

SELECT

p.name,

SUM(split\_times) as total\_race\_time

FROM racing.race\_results AS r

, UNNEST(r.participants) AS p

, UNNEST(p.splits) AS split\_times

WHERE p.name LIKE 'R%'

GROUP BY p.name

ORDER BY total\_race\_time ASC;

Copied!

content\_copy

|  |  |  |
| --- | --- | --- |
| **Row** | **name** | **total\_race\_time** |
| 1 | Rudisha | 102.19999999999999 |
| 2 | Rotich | 103.6 |

**Lab Question: Filtering within ARRAY values**

You happened to see that the fastest lap time recorded for the 800 M race was 23.2 seconds, but you did not see which runner ran that particular lap. Create a query that returns that result.

**Task:** Complete the partially written query:

#standardSQL

SELECT

p.name,

split\_time

FROM racing.race\_results AS r

, r.participants AS p

, p.splits AS split\_time

WHERE split\_time = ;

Copied!

content\_copy

Possible Solution:

#standardSQL

SELECT

p.name,

split\_time

FROM racing.race\_results AS r

, UNNEST(r.participants) AS p

, UNNEST(p.splits) AS split\_time

WHERE split\_time = 23.2;

Copied!

content\_copy

|  |  |  |
| --- | --- | --- |
| **Row** | **name** | **split\_time** |
| 1 | Kipketer | 23.2 |

**Congratulations!**

**Optimize with partitioning and clustering.**

Next up is optimizing with partitioning and clustering.

In a table partitions by a date or a timestamp column, each partition contains a single day of data.

When the data is stored, BigQuery ensures that all the data in a block belongs to a single partition.

A partition's table maintains these properties across all operations that modify it, query jobs, data manipulation language, DML statements, Data Definition Language, DDL statements, load jobs and copy jobs.

This requires BigQuery to maintain more metadata than a non-partitioned table.

As the number of partitions increases, the amount of metadata overhead increases.



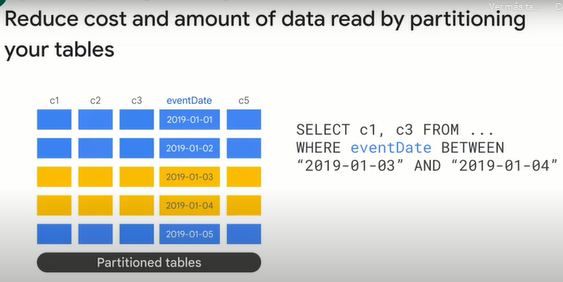
One of the ways you can optimize the tables in your data warehouse is to reduce the cost and amount of data read by partitioning your tables.

For example, assume we have partitioned this table by the event date column.

BigQuery will then change its internal storage so the dates are stored in separate shards. (fragmentos)

Now, when you run a query with a WHERE clause that looks for dates between 01-03 and 01-04, BigQuery will have to read only two fifths of the full dataset.

This can lead to dramatic cost and time savings.



You enable partitioning during the table creation process.

This slide shows how to migrate an existing table to an ingestion time partitioned table.

Using a destination table, it will cost you one table scan.

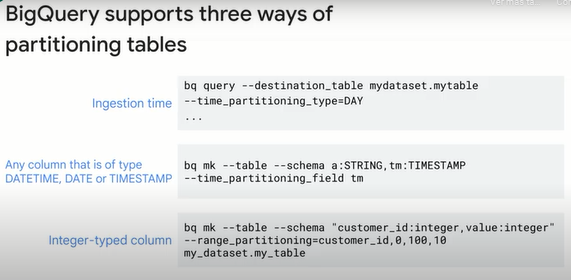
As new records are added to the table, they will be put into the right partition.

BigQuery creates new date based partitions automatically with no need for additional maintenance.

In addition, you can specify an expiration time for data in the partitions.

Partitioning can be set by ingestion time on a timestamp, date or date time column, or based on a range of an integer column.

Here, we are partitioning customer\_ID in the range zero to 100 in increments of 10.



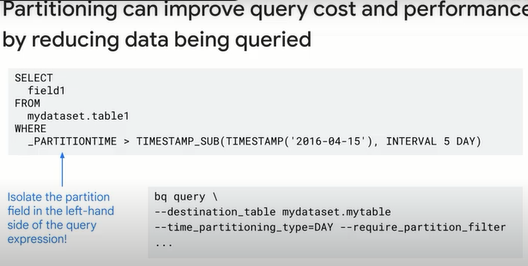
Although more metadata must be maintained, by ensuring that data is partitioned globally, BigQuery can more accurately estimate the bytes processed by a query before you run it.

This cost calculation provides an upper bound on the final cost of the query.

The good practice is to require that queries always include the partition filter, make sure that the partition field is isolated on the left side, because that's the only way BigQuery can quickly discard unnecessary partitions.

An example of this in practice can be found in the blog Optimizing BigQuery, Cluster Your Tables.

A link to the blog is available in the course resources.



Clustering can improve the performance of certain types of queries, such as queries that use Filter clauses, and those that aggregate data.

When data is written to a clustered table by a query or a load job, BigQuery sorts the data using the values in the clustering columns.

These values are used to organize the data into multiple blocks in BigQuery storage.

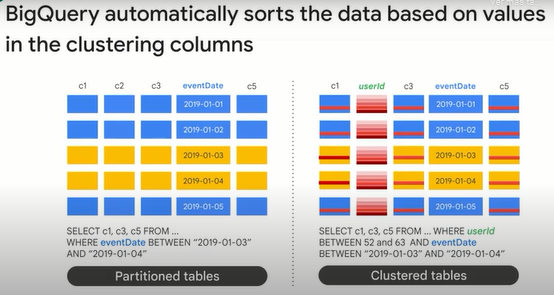
When you submit a query containing a clause that filters data based on the clustering columns, BigQuery uses the sorted blocks to eliminate scans of unnecessary data.

Similarly, when you submit a query that aggregates data based on the values and the clustering columns, performance is improved, because the sorted blocks co-locate rows with similar values.

In this example, the table is partitioned by event date, and clustered by user ID.

Now, because the query looks for partitions in a specific range, only two of the five partitions are considered.

Because the query looks for user ID in a specific range, BigQuery can jump to the row range and read only those rows for each of the columns needed.



You set up clustering at table creation time.

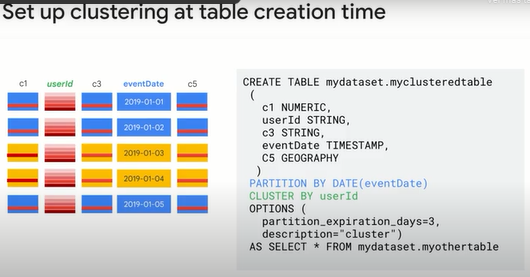
Here, we are creating the table partitioning by event date and clustering by user ID.

We are also telling BigQuery to expire partitions that are more than three days old.

The columns you specify in the cluster are used to co-locate related data.

When you cluster a table using multiple columns, the order of columns you specify is important.

The order of the specified columns determines the sort order of the data.



Over time, as more and more operations modify a table, the degree to which the data is sorted begins to weaken, and the table becomes only partially sorted.

In a partially sorted table, queries that use the clustering columns may need to scan more blocks compared to a table that is fully sorted.

You can re-cluster the data in the entire table by running a select asterisk query that selects from and overwrites the table.

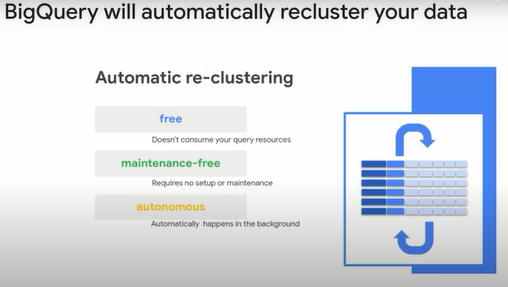
But guess what, you don't need to do that anymore.

The great news is that BigQuery now periodically does auto re-clustering for you.

So you don't need to worry about your clusters getting out of date as you get new data.

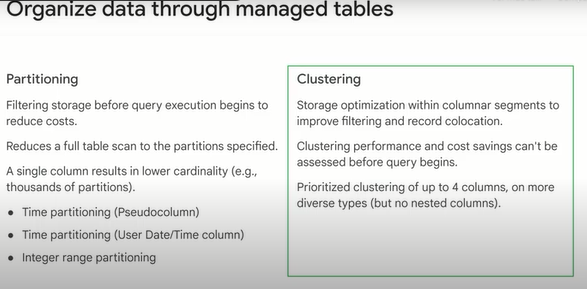
Automatic re-clustering is absolutely free and automatically happens in the background.

You don't need to do anything additional to enable this.



Partitioning provides a way to obtain accurate cost estimates for queries and guarantees improved cost and performance.

Clustering provides additional cost and performance benefits in addition to the partitioning benefits.



BigQuery supports clustering for both partitioned and non-partitioned tables.

When you use clustering and partitioning together, the data can be partitioned by a date, date time or timestamp column, and then clustered on a different set of columns.

In this case, data in each partition is clustered based on the values of the clustering columns.

Partitioning provides a way to obtain accurate cost estimates for queries.

Keep in mind, if you don't have partitioned columns, and you want the benefits of clustering, you can create a fake underscore date column of type date and have all the values be null.



**Review.**

I started by describing what makes a modern data warehouse and what distinguishes a data lake from an enterprise data warehouse.

You were then introduced to BigQuery, a scalable data warehouse solution on Google Cloud.

You don't need to provision resources before using BigQuery unlike with many relational database systems.

BigQuery allocates storage and query resources dynamically based on your usage patterns.

BigQuery enables you to structure your information into datasets, projects, and tables.

You can use multiple datasets to separate tables pertaining to different analytical domains.

And you can use project level scoping to isolate datasets from each other according to your business needs.

Also, you can align projects to billing and use datasets for access control.

BigQuery allows you to batch load source data into a BigQuery table in a single batch operation.

For example, the data source could be a CSV file, an external database, or a set of log files.

BigQuery Data Transfer Service enables you to run batch transfers on a schedule.

Streaming allows you to continually send smaller batches of data in real time, so the data is available for querying as it arrives.

You can also use SQL to generate data and store the results in BigQuery.

Also, some third party applications and services provide connectors that can ingest data into BigQuery.

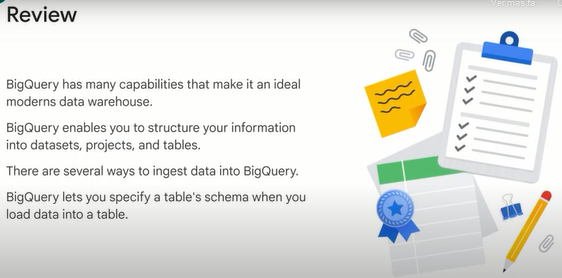
The table schema provides structure to the data.

Remember that every table has a schema which you can enter manually or provide a JSON file with the structure.

Those table schemas can also have array data types, which makes them repeated and or struct data types, which makes them nested.

This type of denormalization will often give you a performance boost because it avoids intensive joins.

You can also setup table partitioning and clustering to reduce the amount of data scanned and speed up your queries.



**Summary.**

Let's review some key concepts we covered in this course on data lakes and data warehouses.

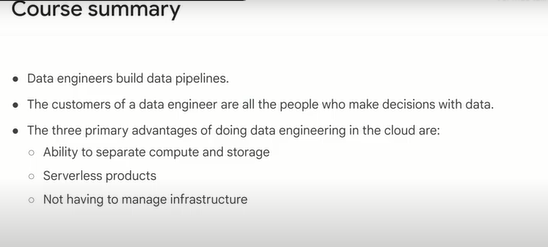
The primary role of a data engineer is to build data pipelines.

The ultimate purpose of a data pipeline is to enable stakeholders in an organization to use data to make faster and better decisions.

While the role of a data engineer is not new, being able to build data pipelines entirely in the Cloud is relatively new.

We argue (discutimos) that doing data engineering in the cloud is advantageous because you can separate compute from storage, and you don't have to worry about managing infrastructure and even software.

This allows you to spend more time on what matters, getting insights from data.



We introduced data lakes and data warehouses and discussed the key differences between the two.

At a high level, a data lake is a place to store unprocessed data, while a data warehouse is a place to store transformed data that you ultimately want to use for analytics, machine learning and dashboards.

Next, we discussed Cloud storage as the data lake solution on Google Cloud in some technical depth.

We also presented other Google Cloud solutions for low latency requirements, transactional workloads, and structured data.

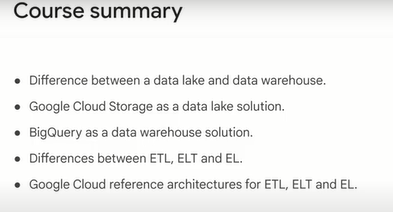
We introduced BigQuery as the data warehouse solution on Google Cloud.

We discussed partitioning and clustering in BigQuery as techniques for improving query performance.

Also, we talked about EL, ELT, and ETL, and how these relate to data lakes and warehouses.

Finally, we presented some reference architectures on Google Cloud for streaming and batch data pipelines.

The hope is that these reference architectures serve as a starting point for your data pipeline.



Congratulations on completing modernizing data lakes and data warehouses with Google Cloud.

Building batch data pipelines on Google Cloud is the second course of the data engineering on Google Cloud core series.

We hope to see you there.

**Quizz.**

1. True or False: ARRAYS can be a standalone field type or part of a STRUCTS field in BigQuery?

False

**True**

2. Which of the following statements on BigQuery is incorrect?

Data on BigQuery is physically stored in a redundant way separate from the compute cluster

A BigQuery slot is a combination of CPU, memory, and networking resources => TRUE

Data is run length-encoded and dictionary-encoded

**The number of slots allotted (asignado) to a query is independent of query complexity**