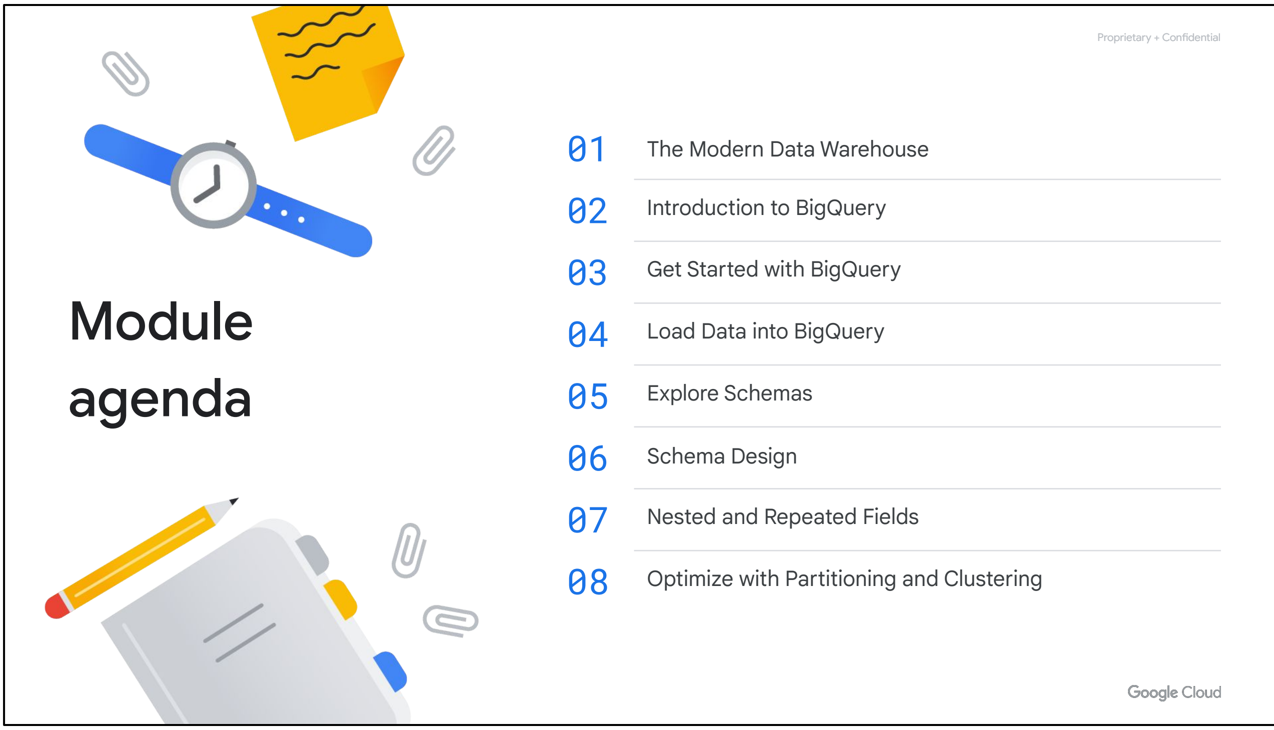
Proprietary + Confidential

Google Cloud

Building a Data Warehouse

Google Cloud

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.

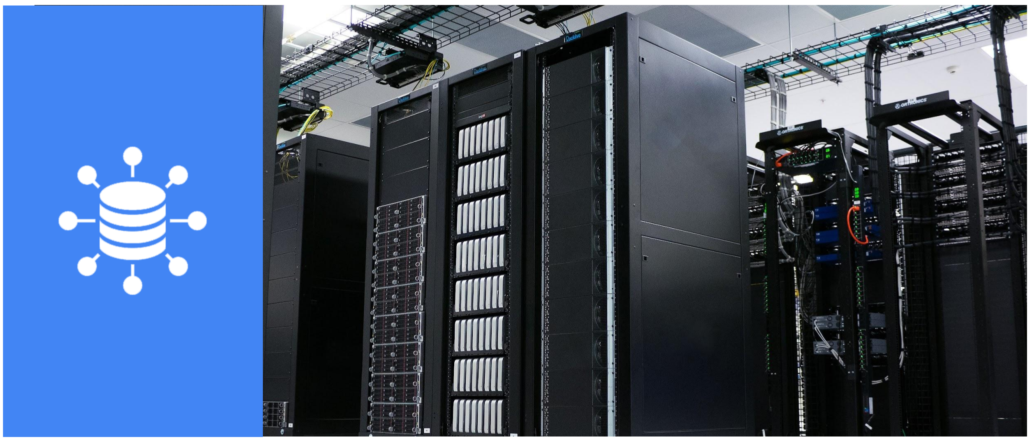
Finally, we'll dive into the world of data warehouse schemas.

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.

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

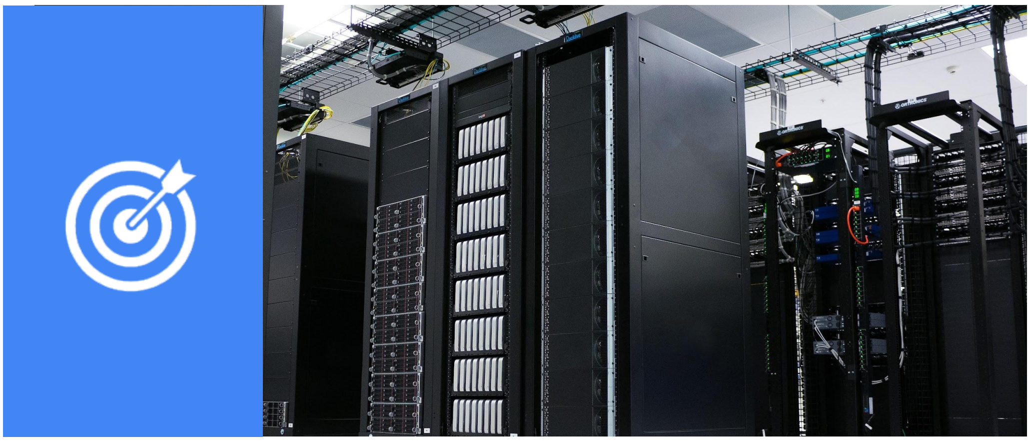
A data warehouse should consolidate data from many sources



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.

The data in a warehouse should have quality, consistency, and accuracy

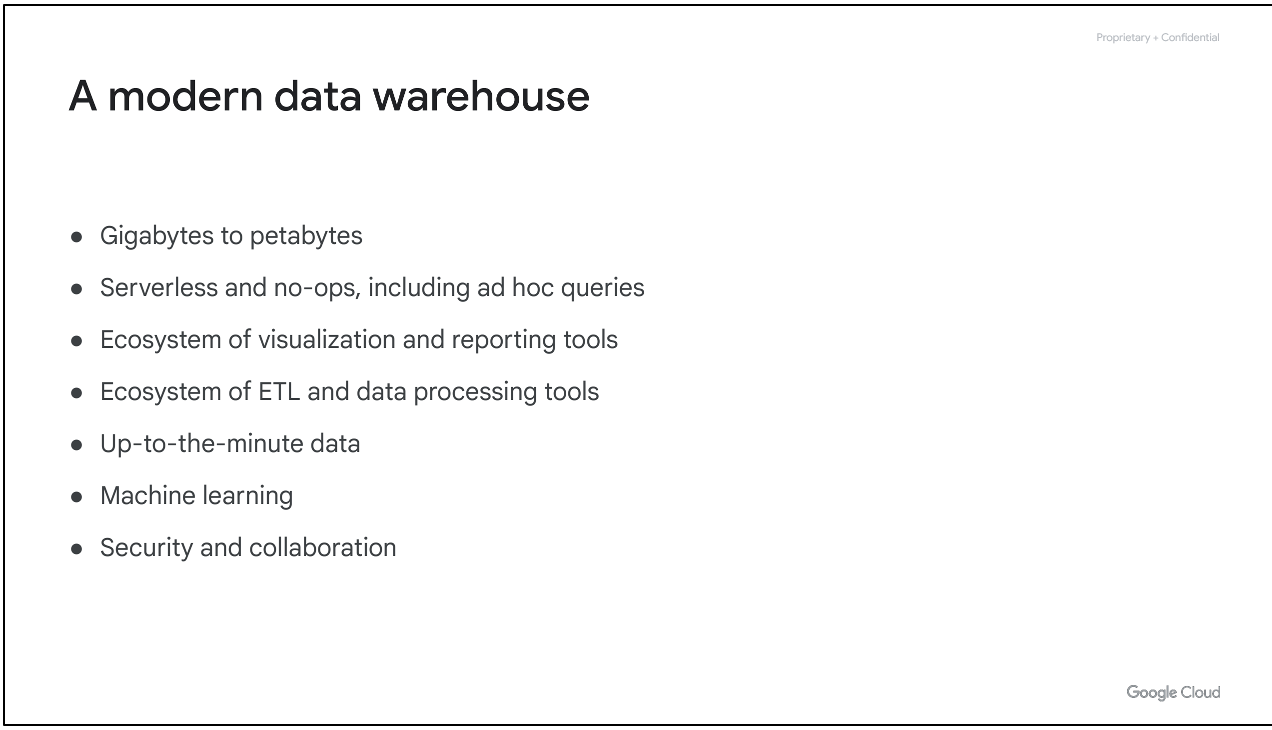


Another reason to consolidate all your data, besides standardizing the format and making it available for querying, is making sure the query results are meaningful. You want to make sure the data is clean, accurate, and consistent.

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.

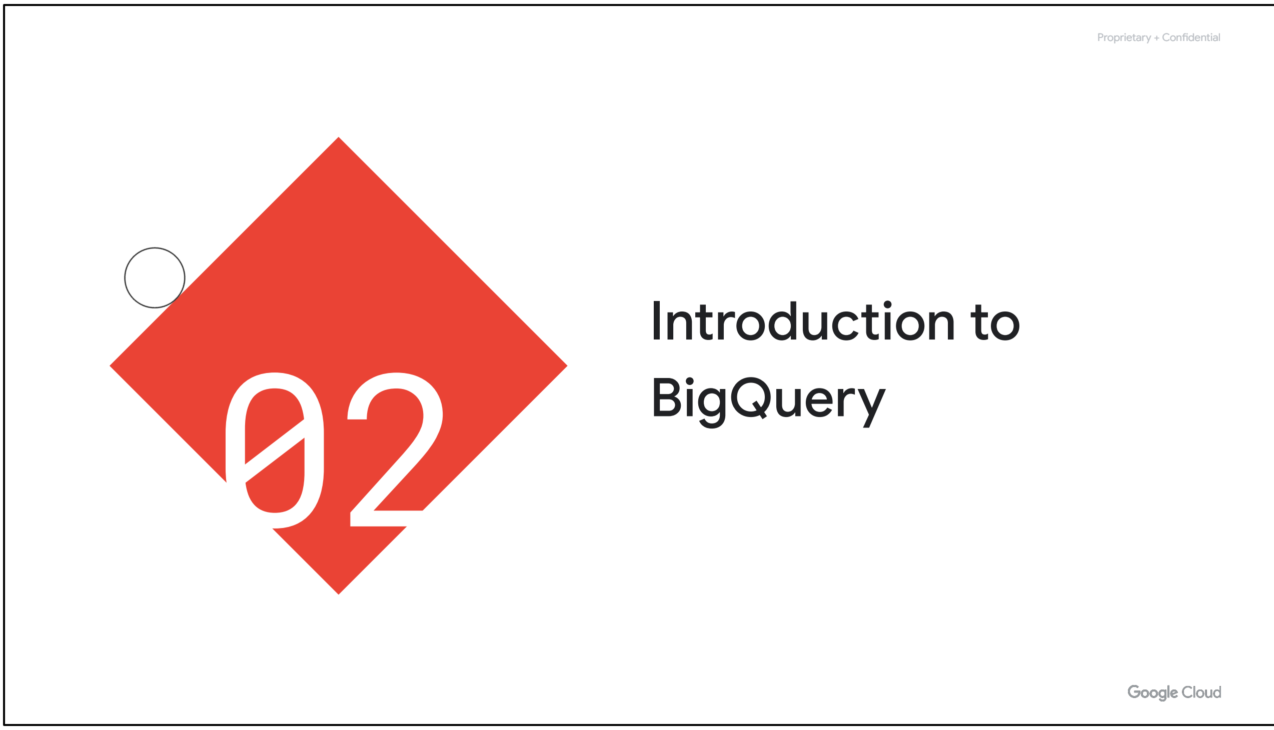
All data in a data warehouse should be available for querying. It's important to ensure that those queries are quick: you don't want people waiting hours or days for results.



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



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

Proprietary + Confidential

BigQuery has many capabilities that make it an ideal data warehouse

- Interactive SQL queries over large datasets (petabytes) in seconds
- Serverless and no-ops, including ad hoc queries
- Ecosystem of visualization and reporting tools
- Ecosystem of ETL and data processing tools
- Up-to-the-minute data
- Machine learning
- Security and collaboration

BigQuery

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.

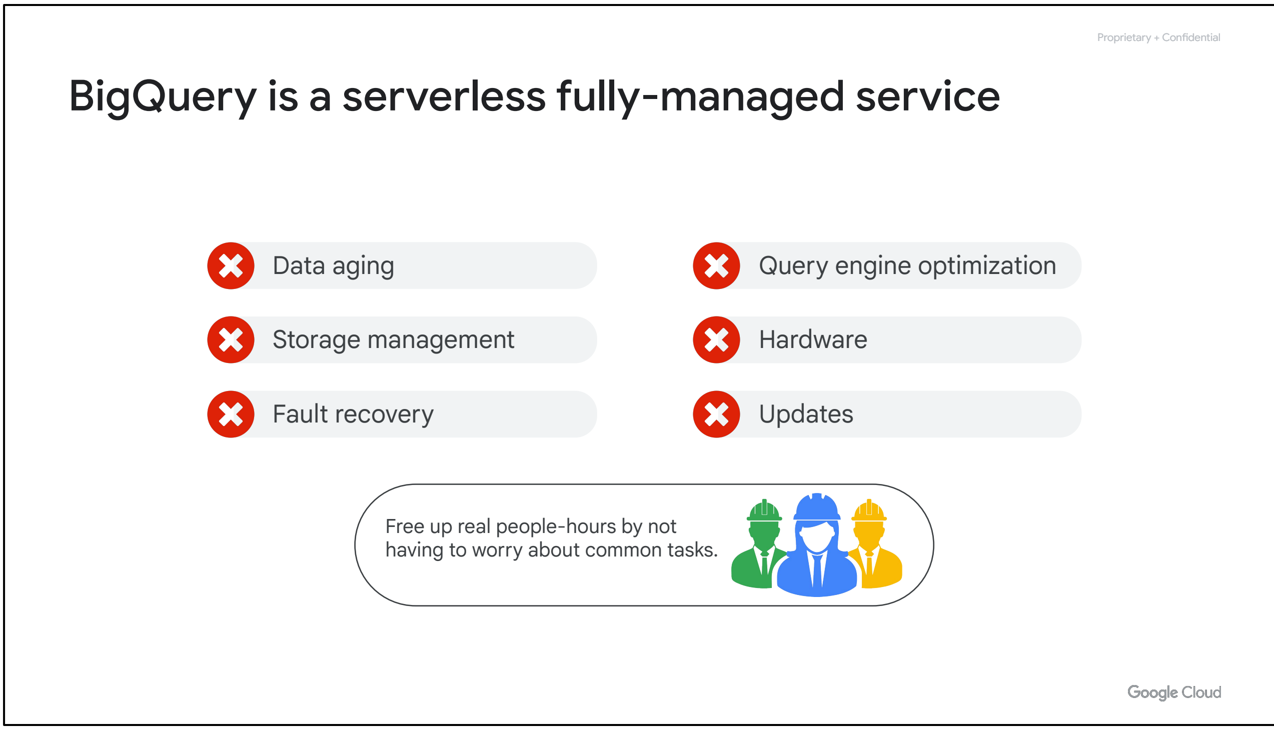
We talked about being able to do ad hoc queries and no-ops.

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.

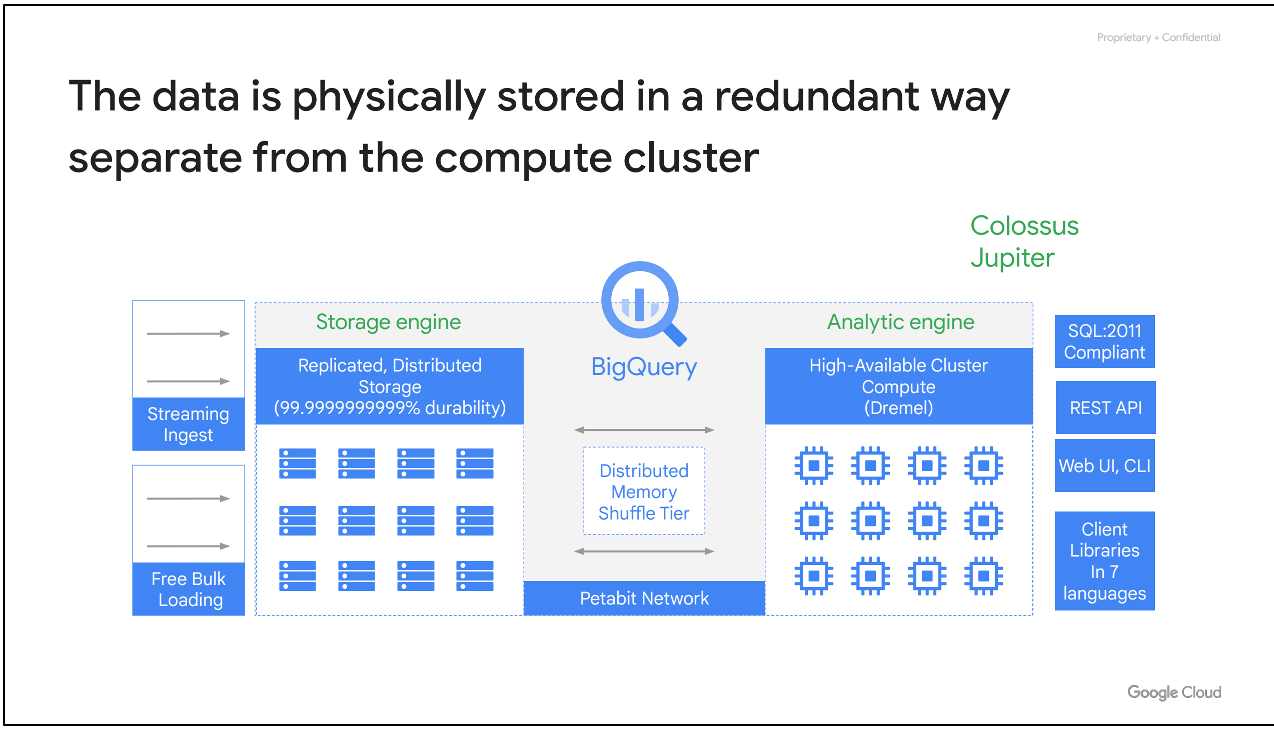


BigQuery is a serverless fully-managed service, which means that the BigQuery engineering team takes care of updates and maintenance for you. Upgrades don't require downtime or hinder system performance.

For example, data aging and expiration can be a cumbersome 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.

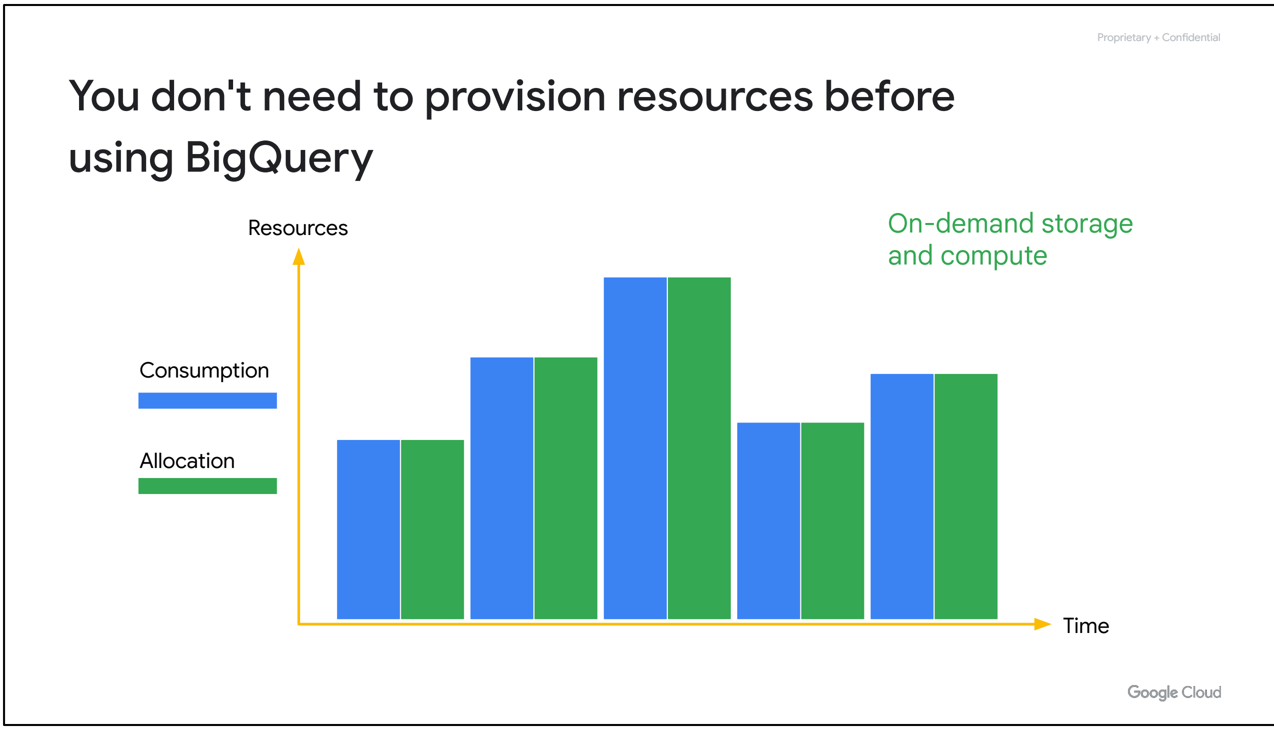
Many traditional systems require resource-intensive vacuum processes to run at various intervals to reshuffle and sort data blocks and recover space. BigQuery has no equivalent of the vacuum process, because the storage engine continuously manages and optimizes how data is stored and replicated. Also, 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.



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 durability by using erasure encoding to store redundant chunks of the data on multiple physical disks. Moreover, the data is replicated to multiple data centers.



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.

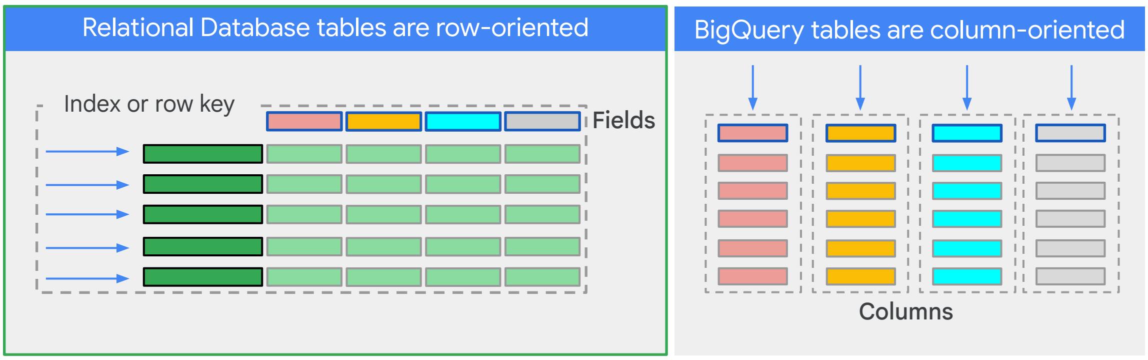
Storage resources are allocated as you consume them and deallocated as you remove data or drop tables.

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.

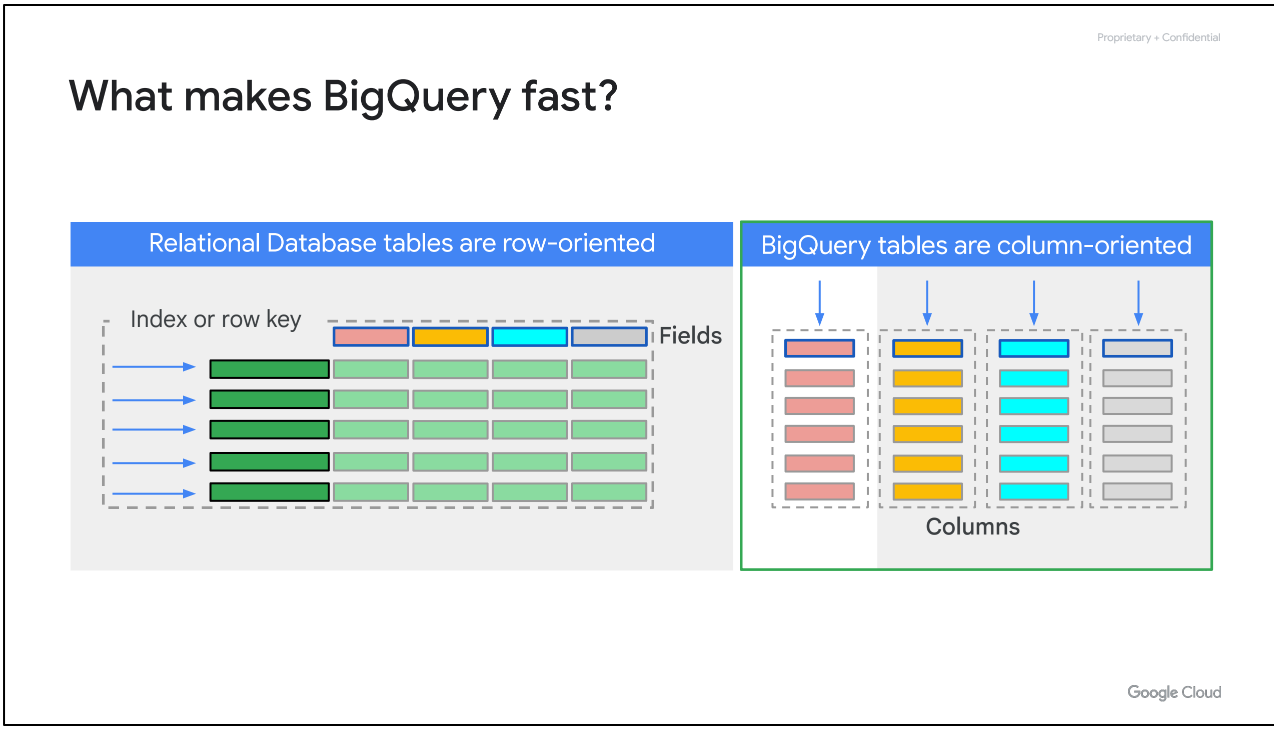
What makes BigQuery fast?

So what makes BigQuery fast? BigQuery tables are column-oriented, compared to traditional RDBMS tables which are row-oriented.

What makes BigQuery fast?

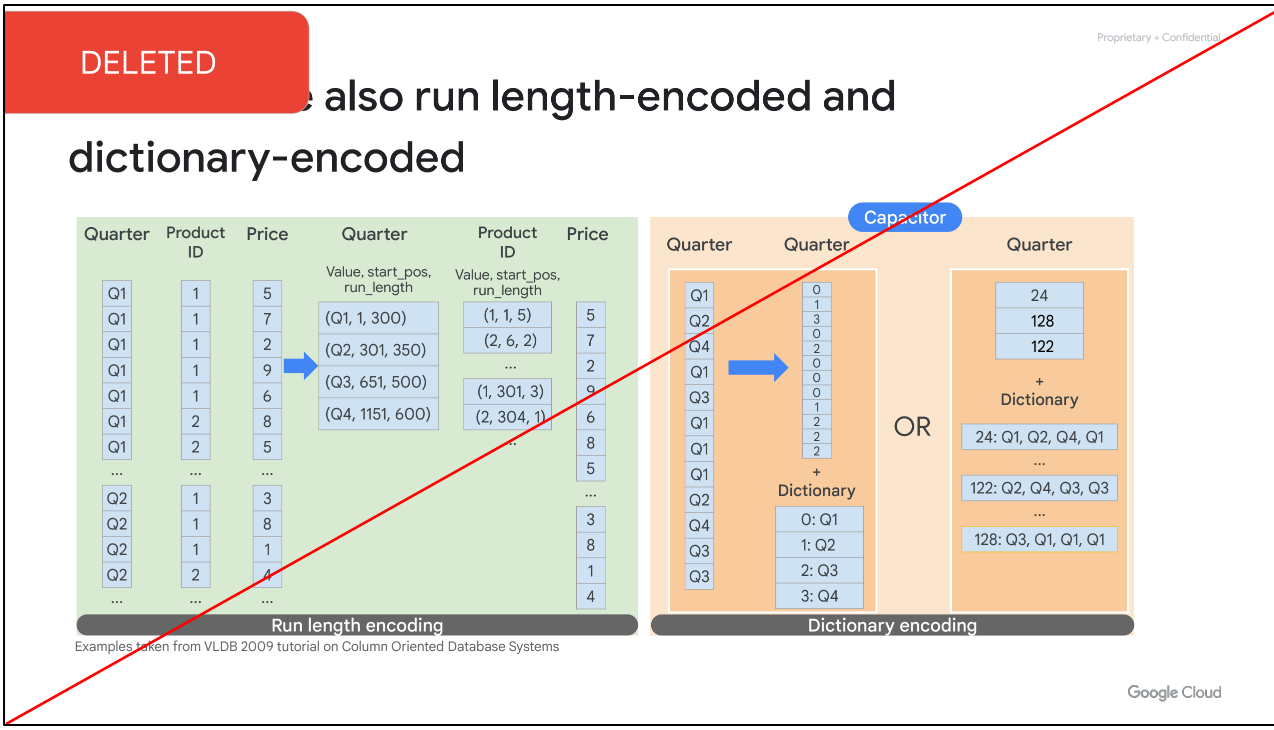


Row-oriented tables are efficient for making updates to data contained in fields. For OLTP systems, row-oriented tables are necessary because OLTP systems have frequent updates. Analytics is slow on row-oriented tables because queries have to read all the fields in a row and, depending on the kind of indexing or key, queries may have to read extra rows and fields to find the information that is requested.



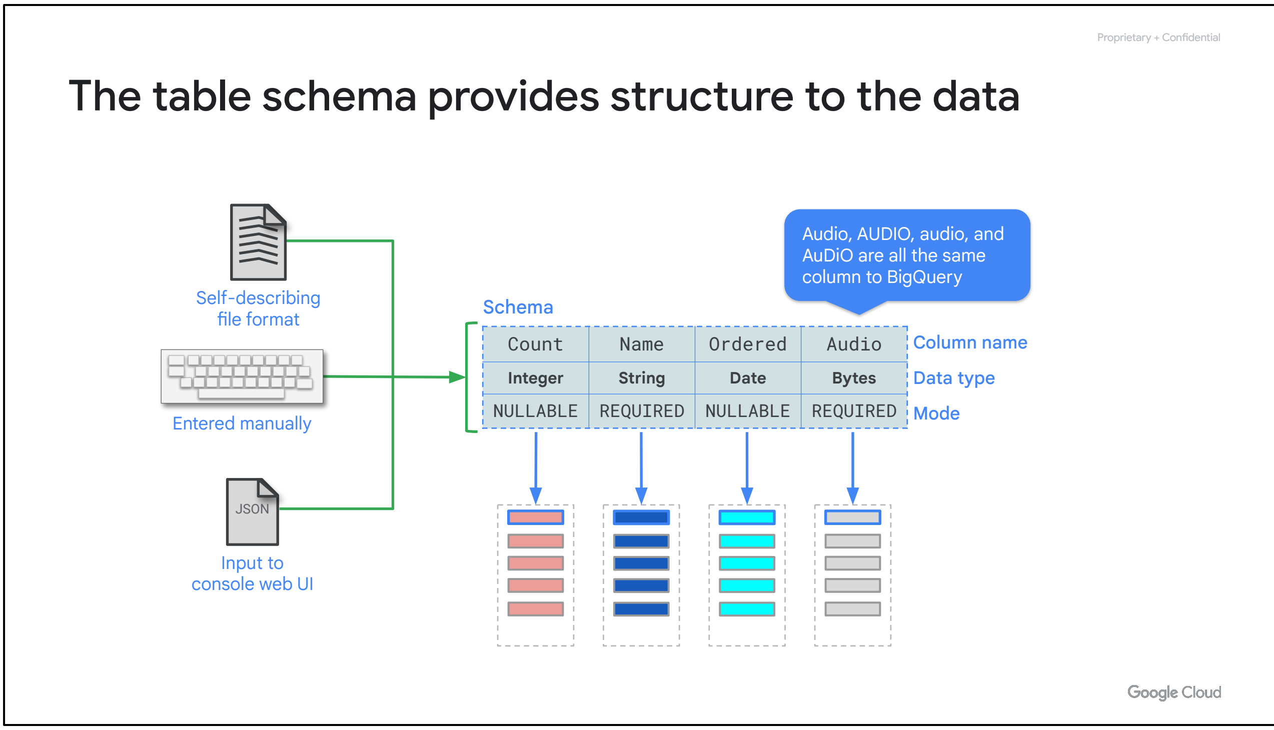
BigQuery, however, is an O-LAP system. It's meant for analytics. BigQuery tables are immutable and are optimized for reading and appending data. BigQuery tables are not optimized for updating.

BigQuery leverages the fact that most queries involve few columns, so it only reads the columns required for the query. BigQuery is very efficient in this sense and is the reason tables are column-oriented.

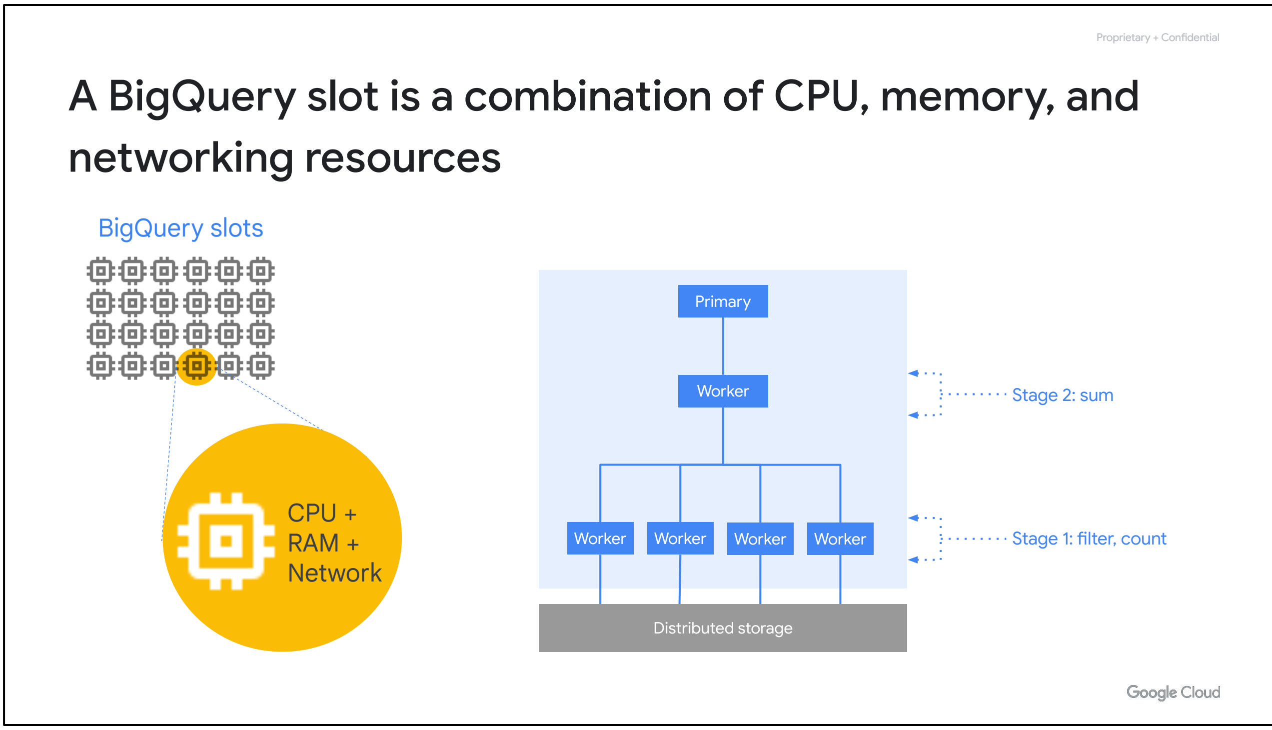


~~Here are a couple of the optimizations that Capacitor does. Capacitor runs length-encodes on the data so that it can reduce the amount of data needed to be read. It also reorders the data to make it more conducive for run-length-encoding. Reordering the data is also called dictionary encoding.~~

~~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.~~



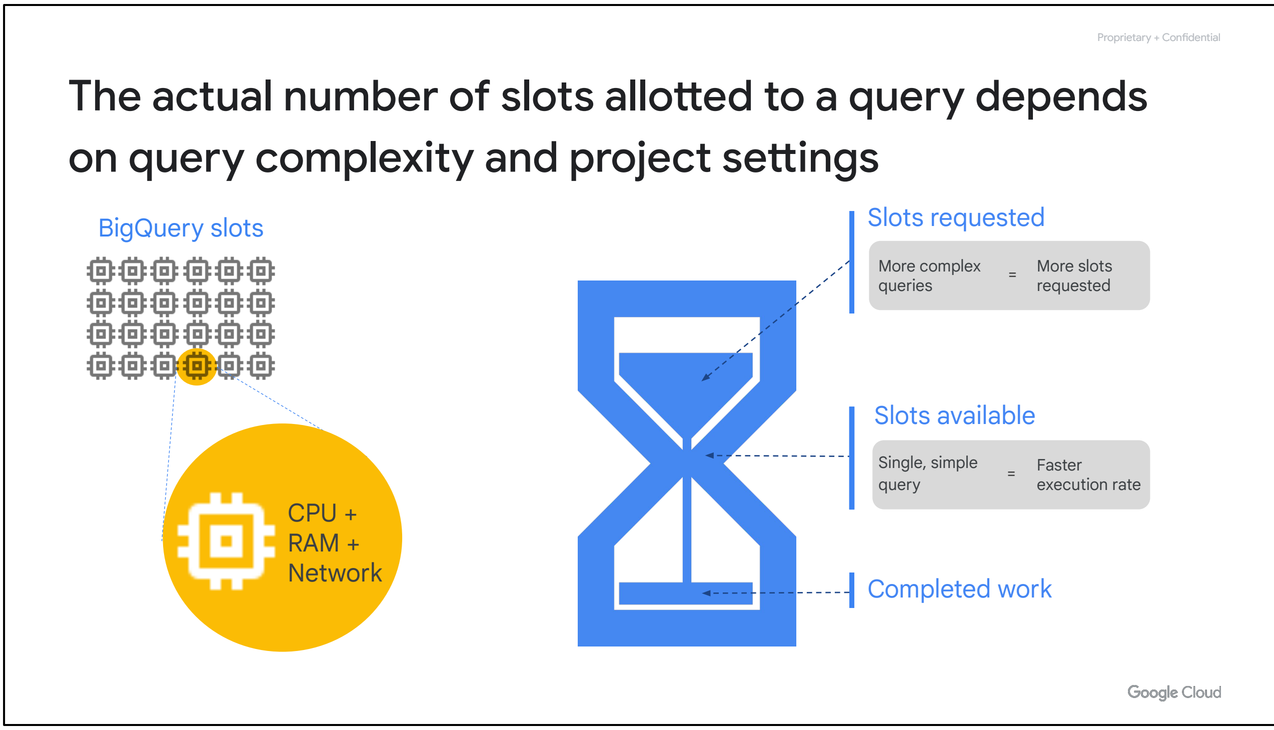
Every table has a schema. You can enter the schema manually through the Cloud Console, or by supplying a JSON file.



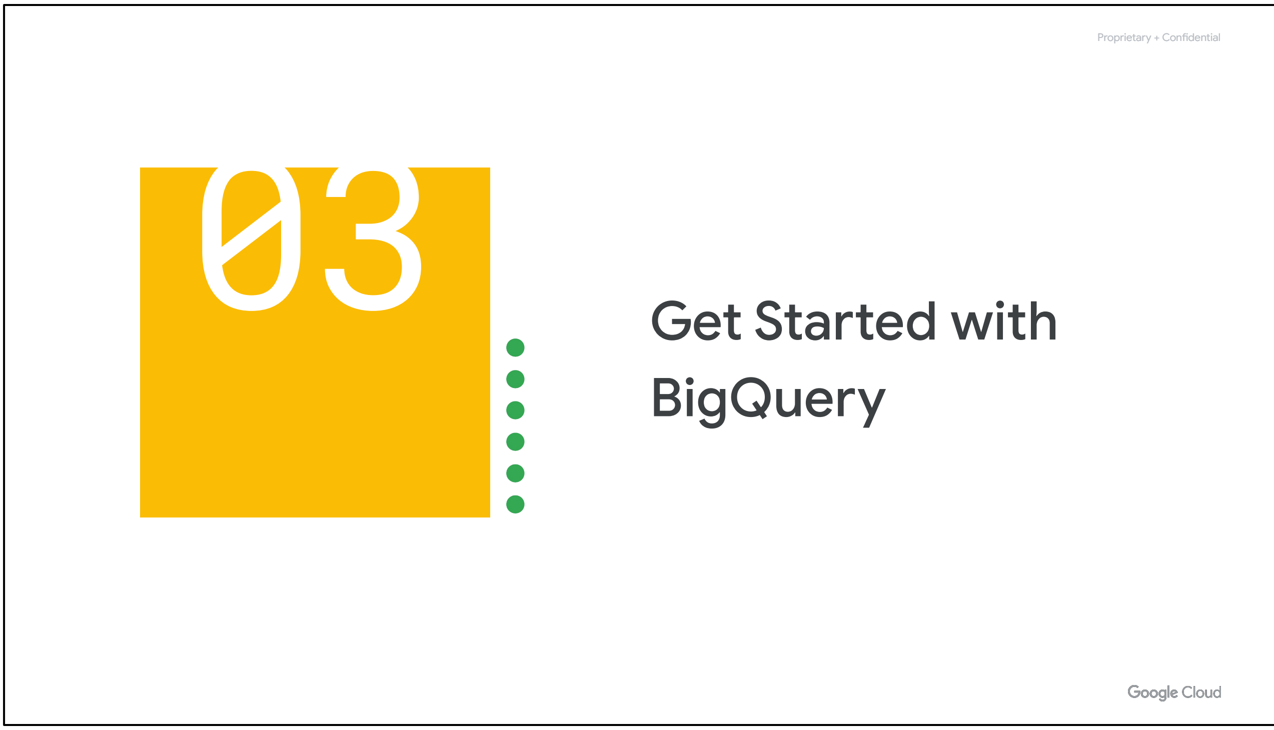
BigQuery is implemented using a microservice architecture, so there are no virtual machines to configure and maintain. Under the hood, analytics throughput is measured in BigQuery slots. A BigQuery slot is a unit of computational capacity required to execute SQL queries. BigQuery automatically calculates 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 I/O.

On the right hand side you can see how multiple slots work together under the hood to execute a query. You can imagine a slot operates here the same way as a worker node in a cluster for distributed processing of big data. When executing a query, BigQuery may split it up into one or multiple stages that contain different tasks to do. Tasks are assigned to workers to perform the work in parallel. In this example, we have 2 stages. In the first one, the workers will pick up a subset of the data they need to work on from BigQuery's storage, then they apply a specific filter to their data (coming from the WHERE clause) and do a partial count of the data that is left (which is specified as COUNT() in the SELECT clause). Each worker will send its intermediate result to stage 2, where one worker will create the final result set by summing up all the counts it received by the previous workers. This result set is then presented to us in the UI. We will look at this again in our module around BigQuery performance.

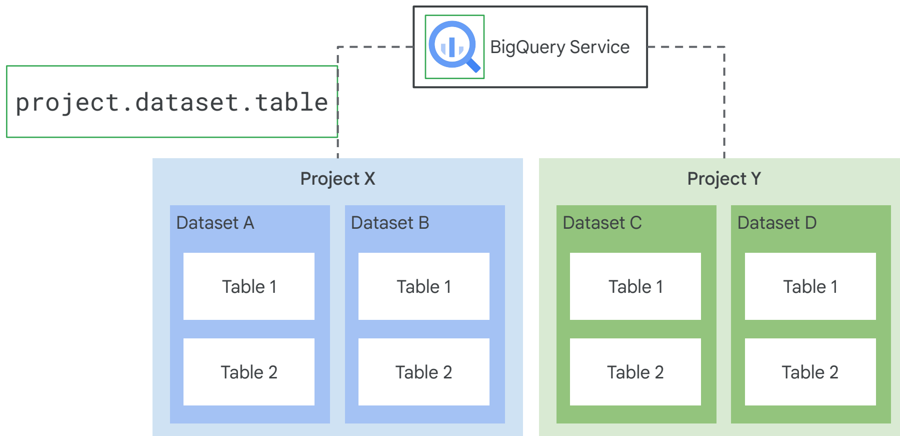


If a single, simple query is submitted that needs fewer slots than are available, the query will generally execute faster.



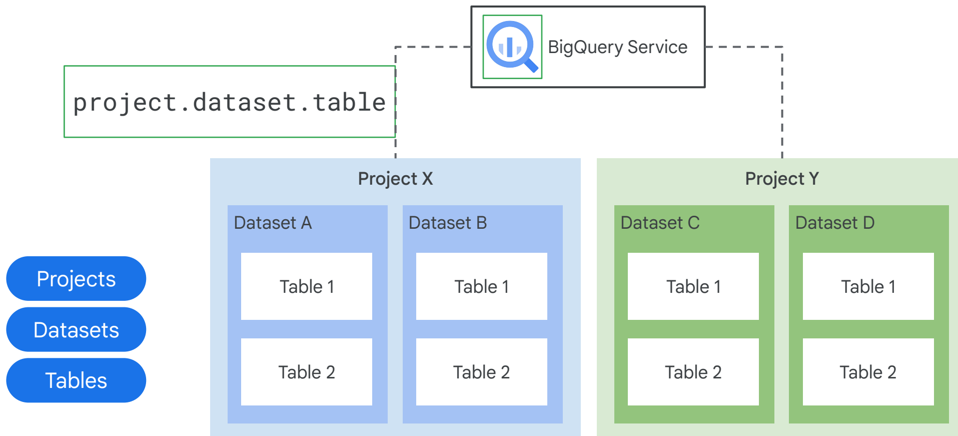
Now that you're familiar with the basics of BigQuery, it's time to talk about how BigQuery organizes your data.

BigQuery organizes data tables into units called datasets



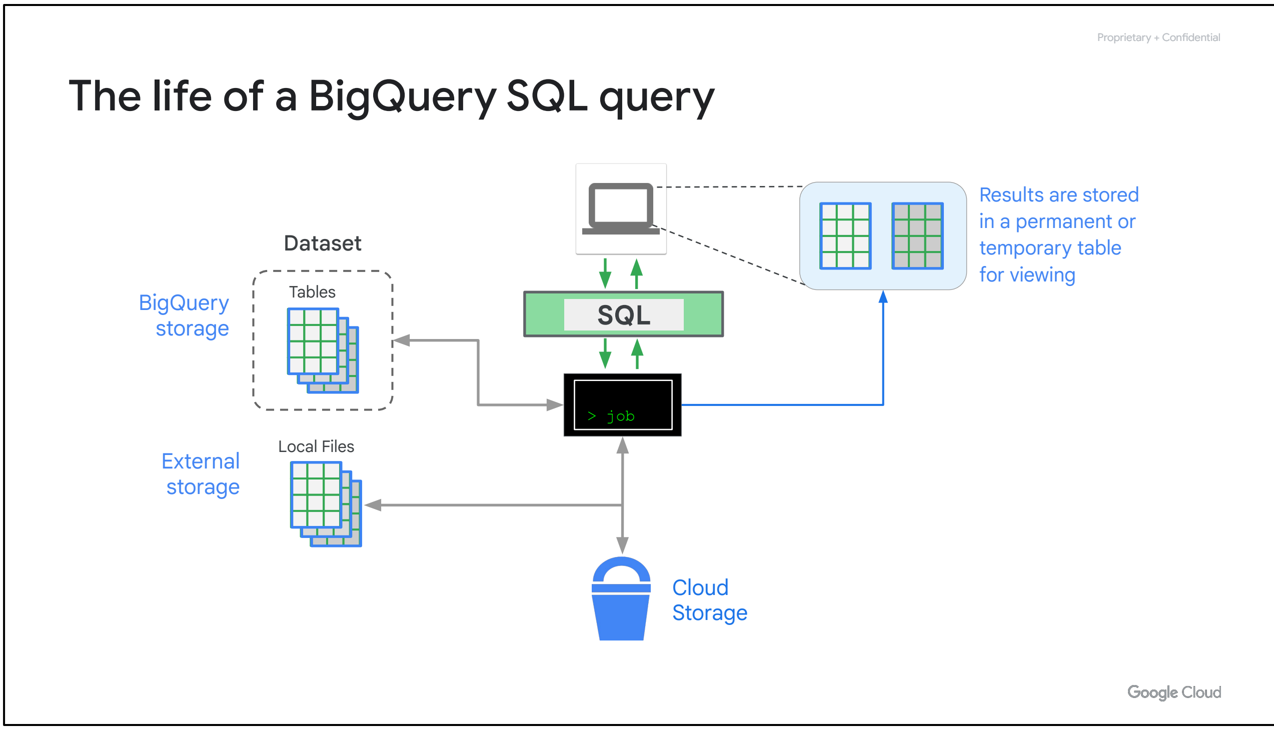
BigQuery organizes data tables into units called datasets. These datasets are scoped to your Google Cloud project. When you reference a table from the command line in SQL queries or in code, you refer to it by using the construct: project.dataset.table.

What are some reasons to structure your information?



What are some reasons to structure your information into projects, datasets, 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.



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 QueryJob 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.

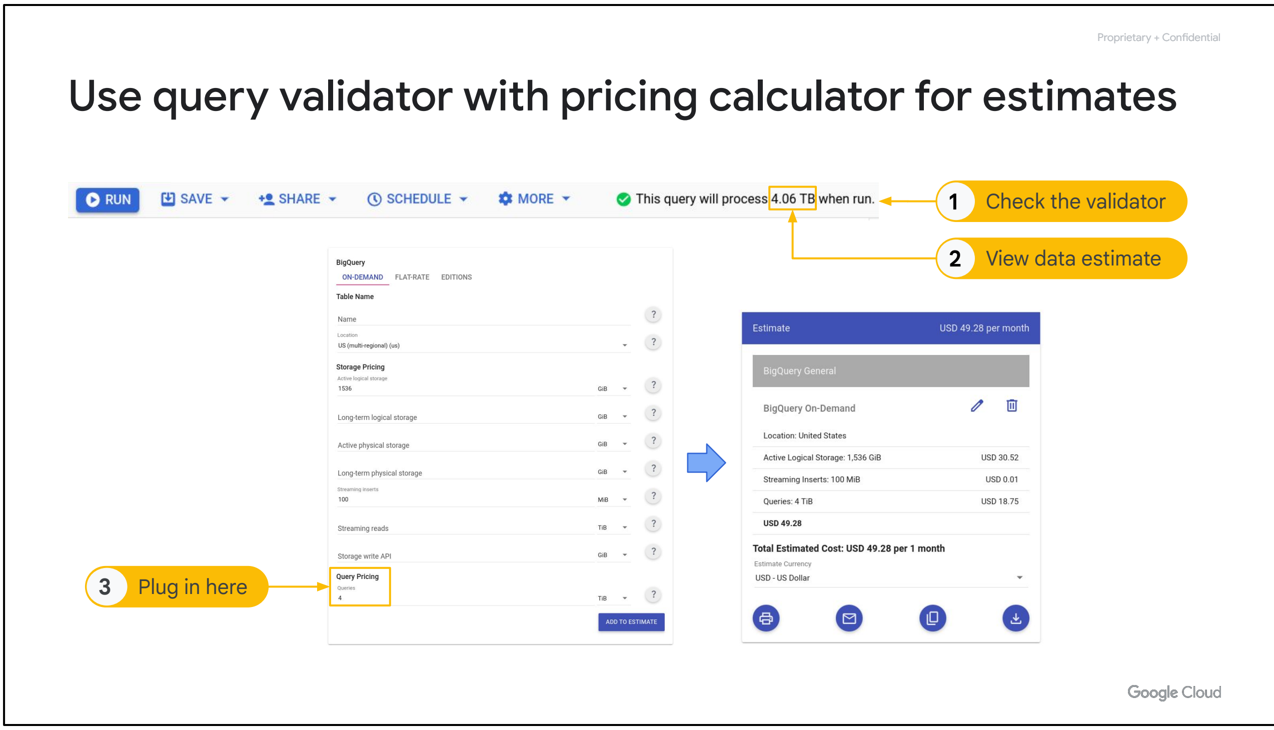
BigQuery is most efficient when working with data contained in its own storage service. The storage service and the query service work together to internally organize the data to make queries efficient over huge datasets of Terabytes and Petabytes in size.

The query service can also run query jobs on data contained in other locations, such as tables in 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.

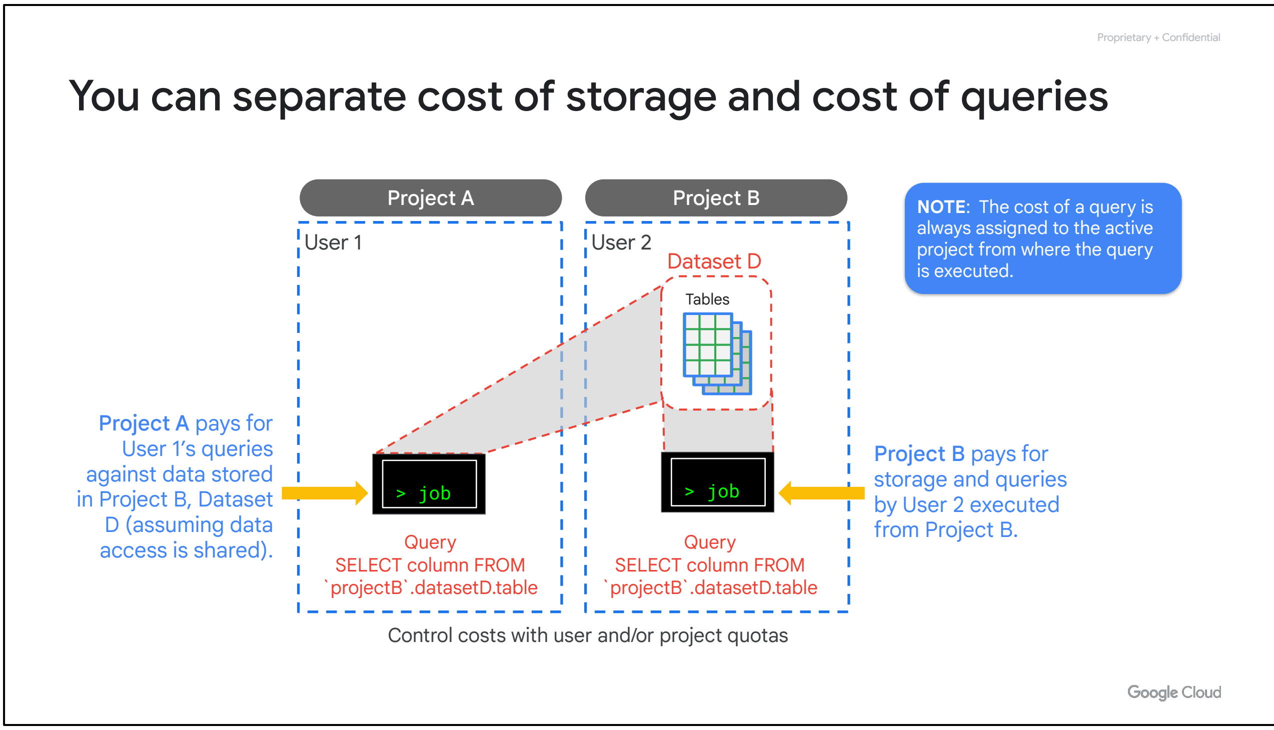
In either case, the query service puts the results into a temporary table and the user interface pulls and displays the data in the temporary table. This temporary table is stored for 24 hours, so if you run the exact same query again, and if the results would not be different, then BigQuery will simply 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.



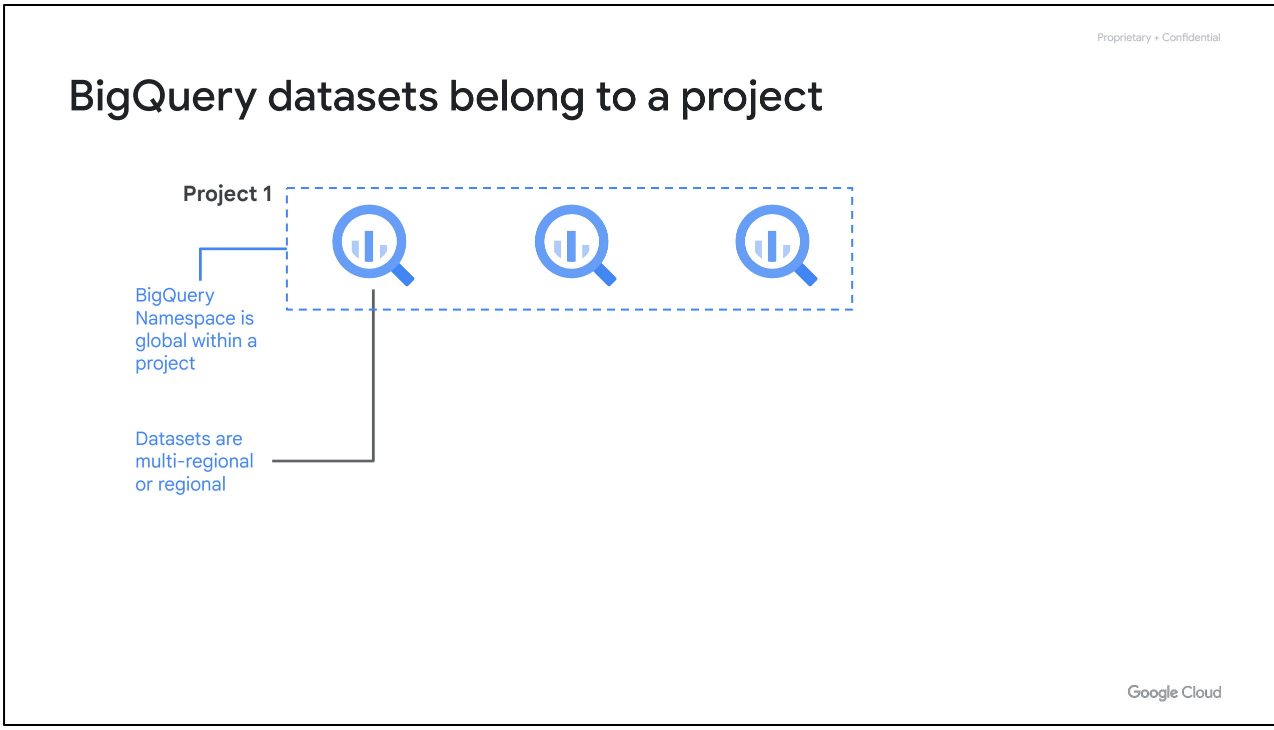
You can separate cost of storage and cost of queries.

By separating projects A and B, it's possible to share data without giving access to run jobs. In this diagram, Users 1 and 2 have access to run jobs and access the datasets in their own respective Projects. If they run a query, that job is billed to their own project.

What if User 1 needs the ability to access Dataset D in Project B? The person who owns Project B can allow User 1 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 users access to use their data. The special setting allAuthenticatedUsers 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.

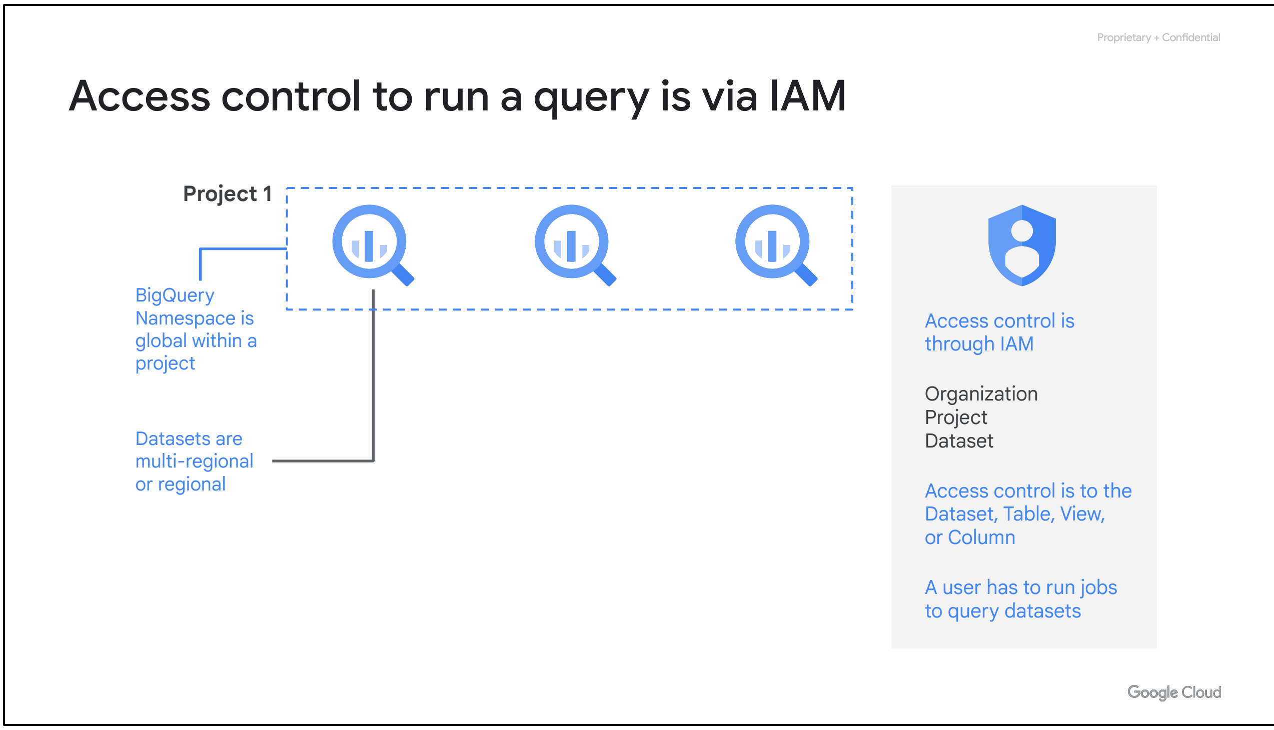
In summary, the cost of a query is always assigned to the active project from where the query is executed. The active project for a user is displayed at the top of the Cloud Console or set by an environmental variable in the Cloud Shell or client tools.



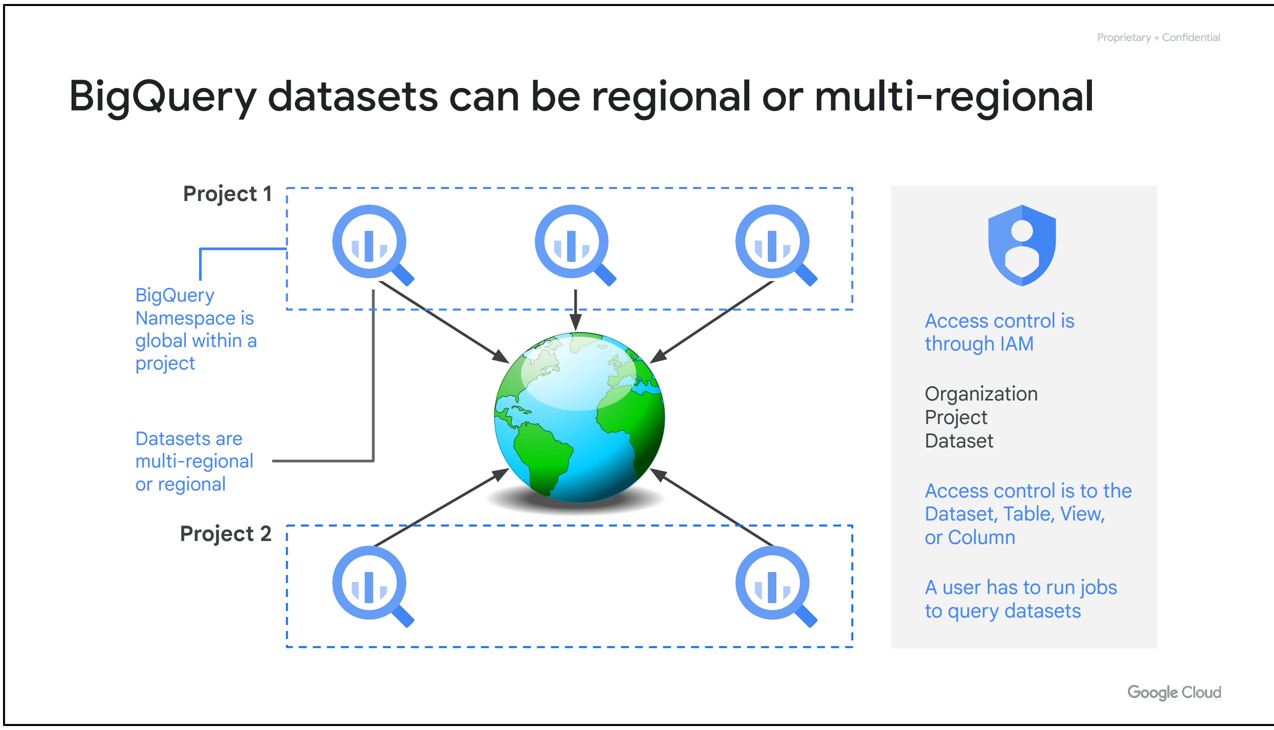
The project is what the billing is associated with. For example, if you queried a table that belongs to the `bigquery-public-data` project, the storage costs are billed to that data project.

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

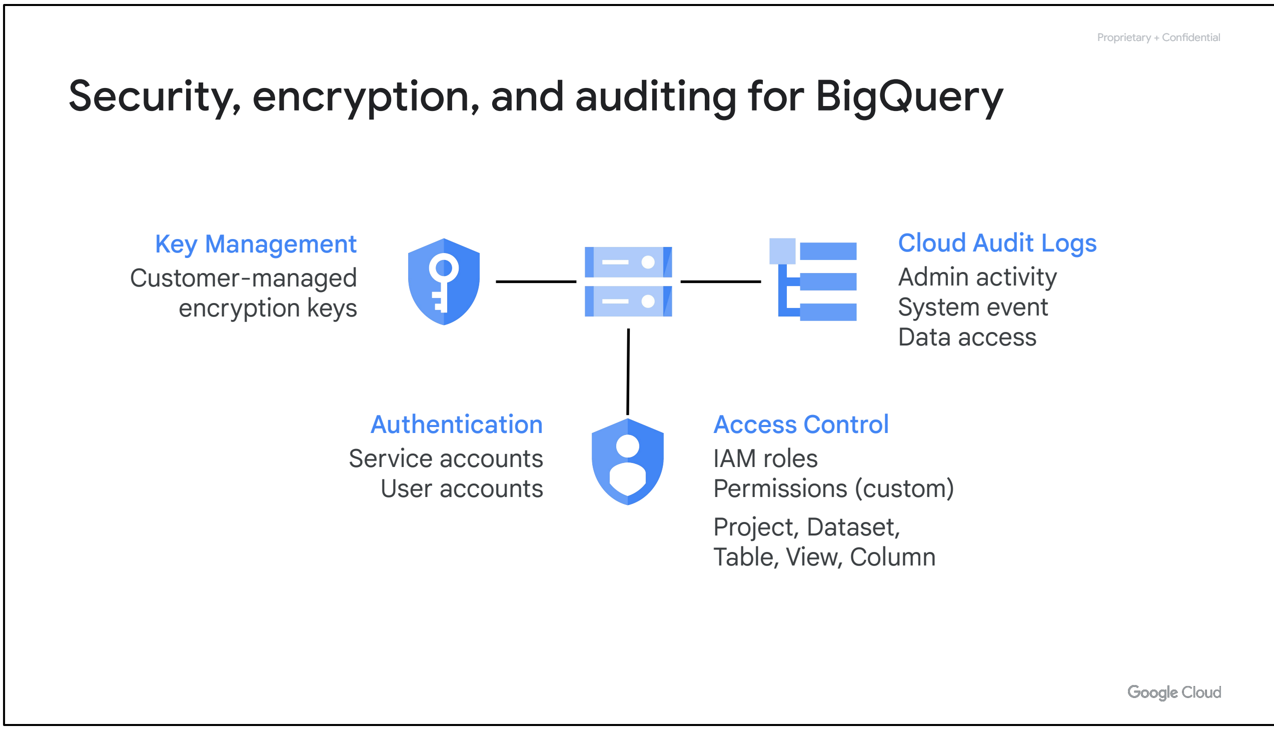
In order to run a query in a project, you need Identity Access Management (IAM) permission to submit a job. Remember that running a query means that you must be able to submit a query job to the service.



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.



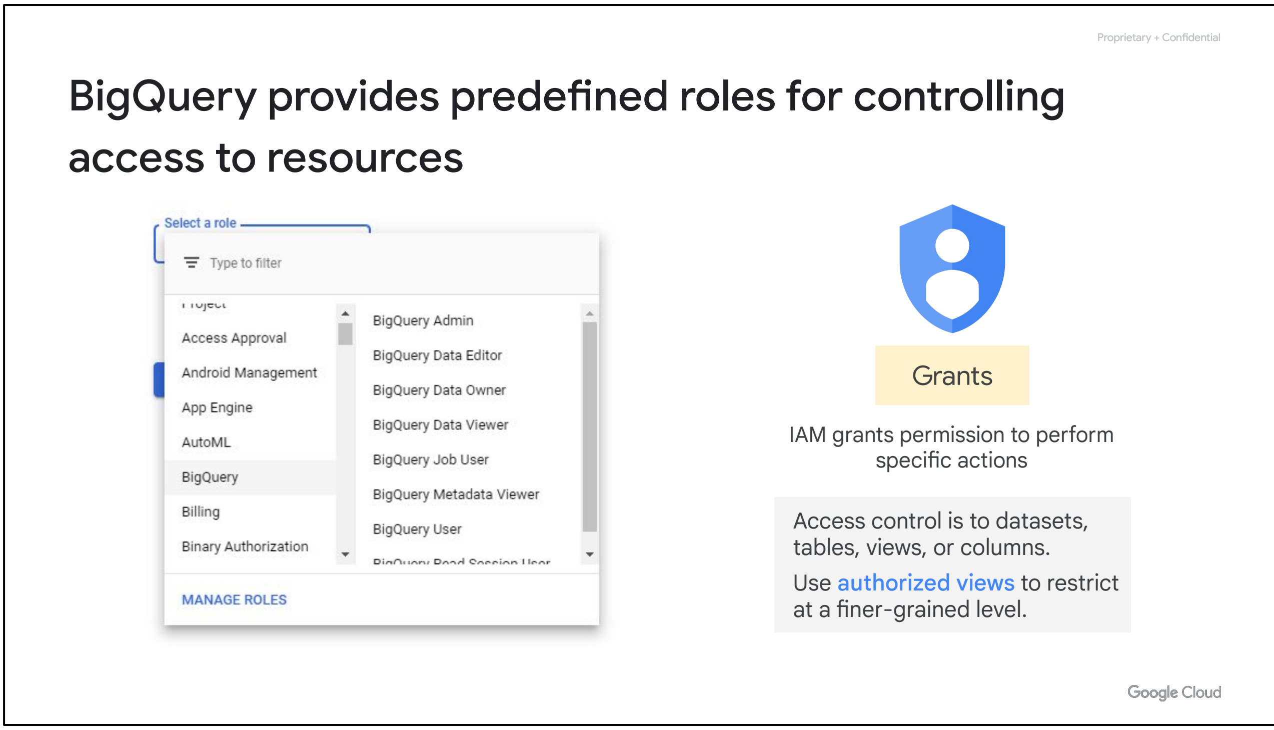
Key Management
As with Cloud Storage, BigQuery storage encrypts data at rest and over the wire using Google-managed encryption keys. It's also possible to use customer-managed encryption keys.

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

Access control
Access control is through IAM roles and involves giving permissions. We discussed two of those 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.

Cloud Audit Logs
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.

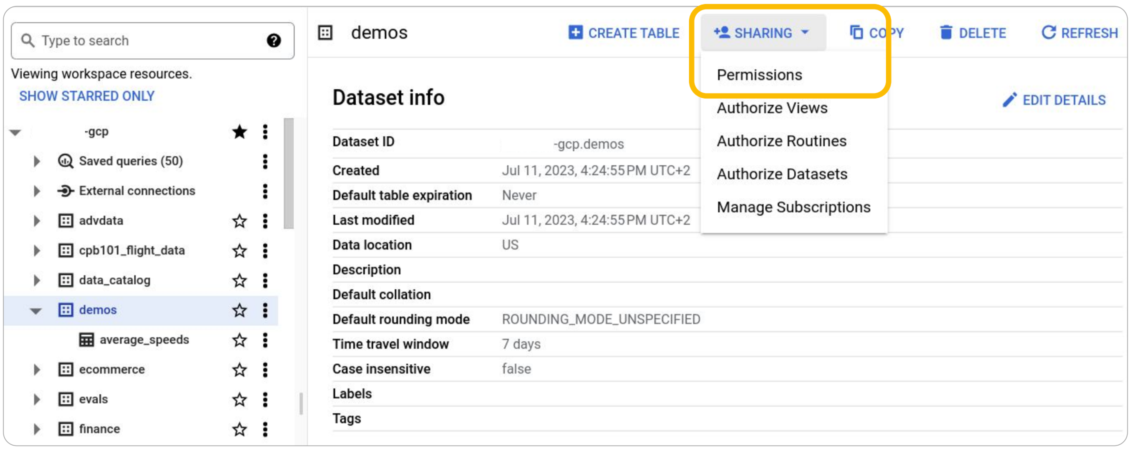
If, when creating a table, you configure it to expire in 30 days, at the end of 30 days a system event will be generated and logged. You will also get immutable logs of every access that happens to a dataset under your project.



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 but 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.

It’s easy to share access to datasets with other analysts



Google Cloud

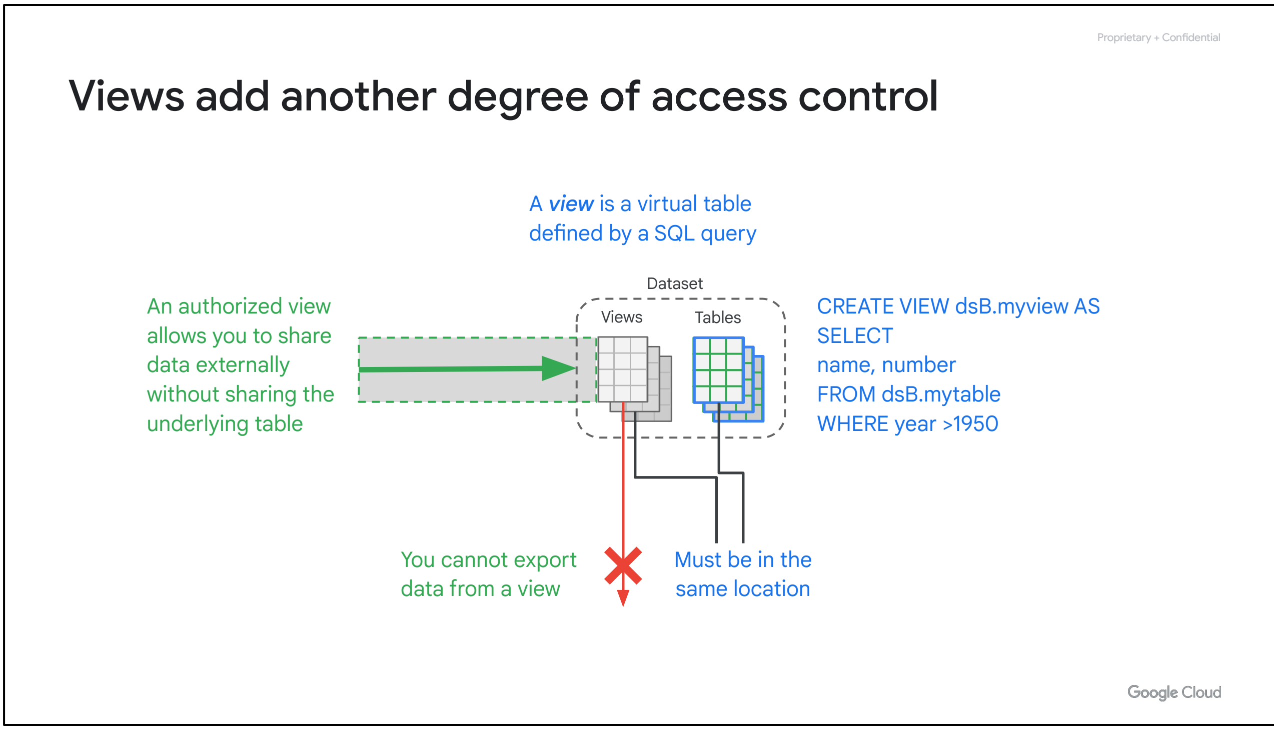
Sharing access to datasets is easy.

Traditionally, onboarding new data analysts involved 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's time to productivity.

To onboard an analyst on Google Cloud, you grant access to relevant project(s), 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.

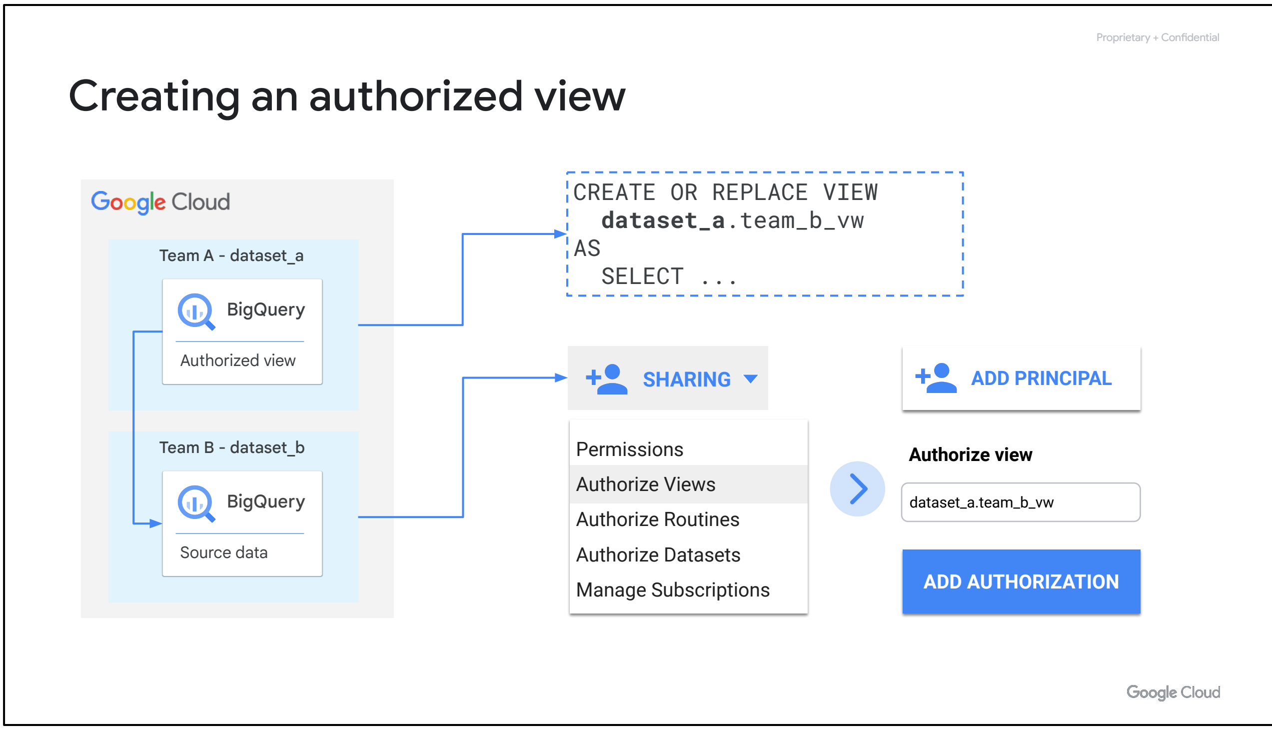
The BigQuery web UI presents the list of datasets that the analyst has access to. Analysts can perform tasks in the Cloud Console according to the role you grant them, such as viewing metadata, previewing data, executing, and saving and sharing queries.



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

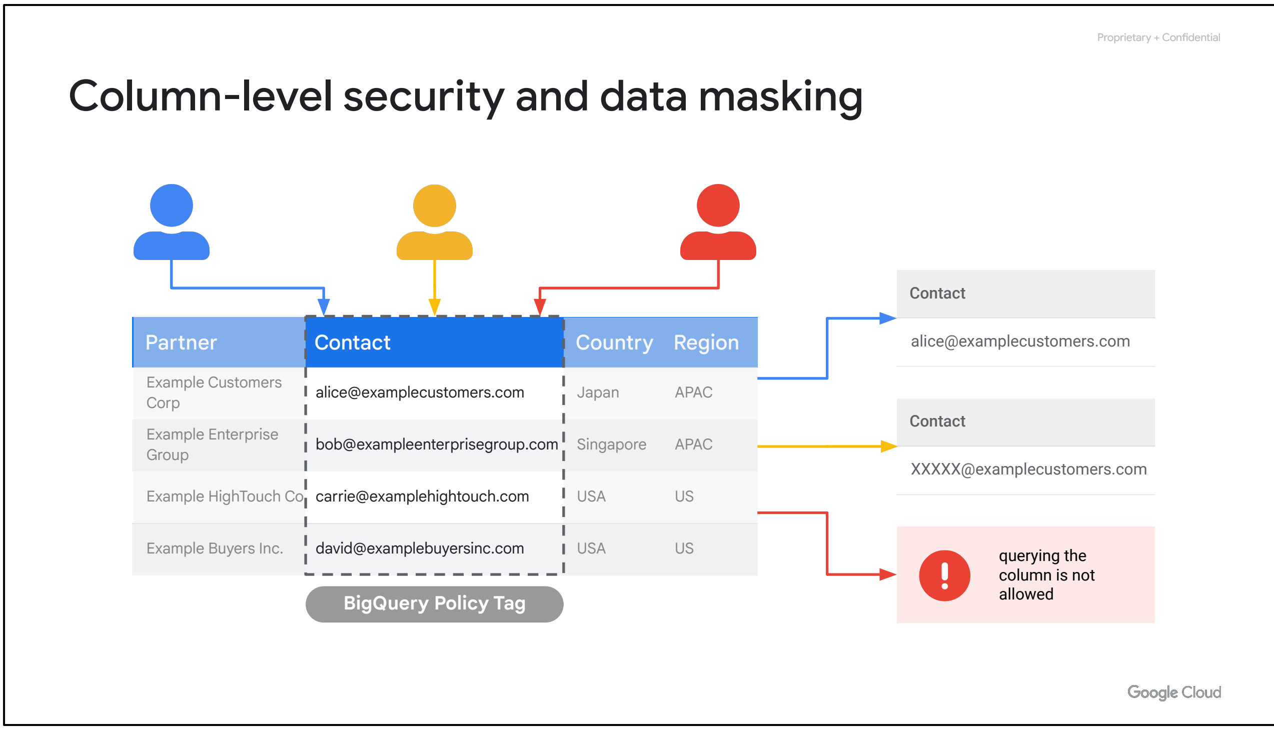
What if you want more fine-grained control? In addition to access controls at the table or column level, you can use views. In this example, we are creating a view in Dataset B, and the view is a subset of the table data in Dataset A. Now, by providing users with access to Dataset B, we are creating 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 multi-region as dataset A.

A view is 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.



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.

[Creating authorized views: https://cloud.google.com/bigquery/docs/authorized-views]



With column-level security you can achieve a similar use case, so that you can define when to show the content of the column, or when to hide or obfuscate it.

You will define a Policy Tag in BigQuery and assign the corresponding users/groups to it. These users will now be able to see the column's content.

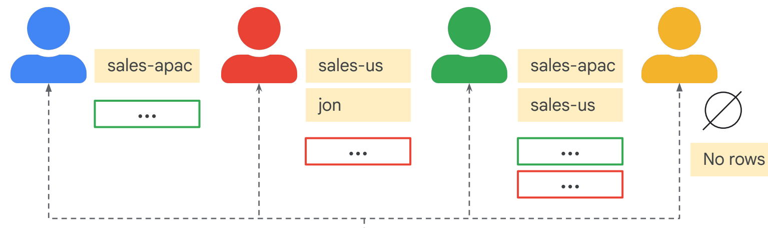
You can also define data masking rules and assign them to the Policy Tag and users/groups, so that the content is obfuscated / nullified / or transformed using your own custom logic.

If a user/group isn't included into the Policy Tag's definition, they won't be able to query the column at all.

Row-level security in BigQuery

```
CREATE ROW ACCESS POLICY
    apac_filter
ON
    dataset1.table1
GRANT TO
("group:sales-apac@example.com")
FILTER USING
    (Region="APAC");
```

```
CREATE ROW ACCESS POLICY
    us_filter
ON
    dataset1.table1
GRANT TO
("group:sales-us@example.com",
 "user:jon@example.com")
FILTER USING
    (Region="US");
```



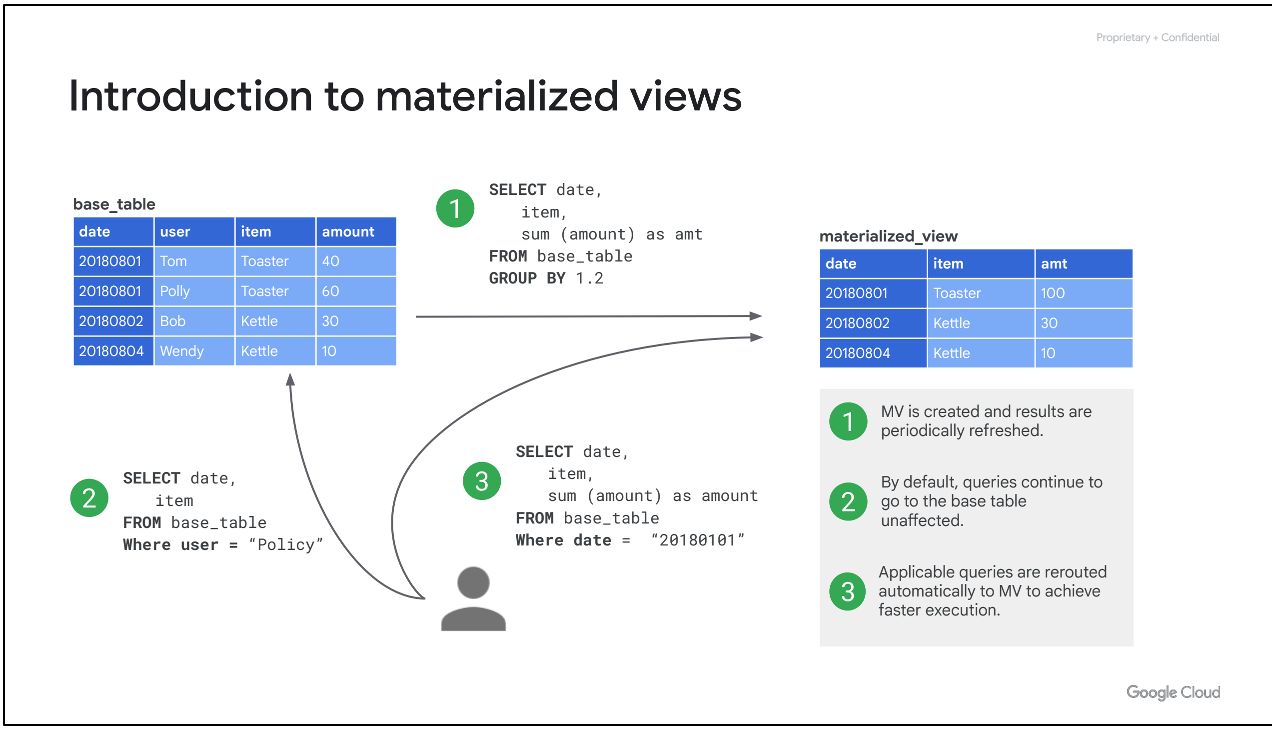
Partner	Contact	Country	Region
Example Customers Corp	alice@examplecustomers.com	Japan	APAC
Example Enterprise Group	bob@exampleenterprisegroup.com	Singapore	APAC
Example HighTouch Co.	carrie@examplehightouch.com	USA	US
Example Buyers Inc.	david@examplebuyersinc.com	USA	US

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 group is in an allowed list.

An authorized user, with the IAM roles BigQuery Admin or BigQuery DataOwner, 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.

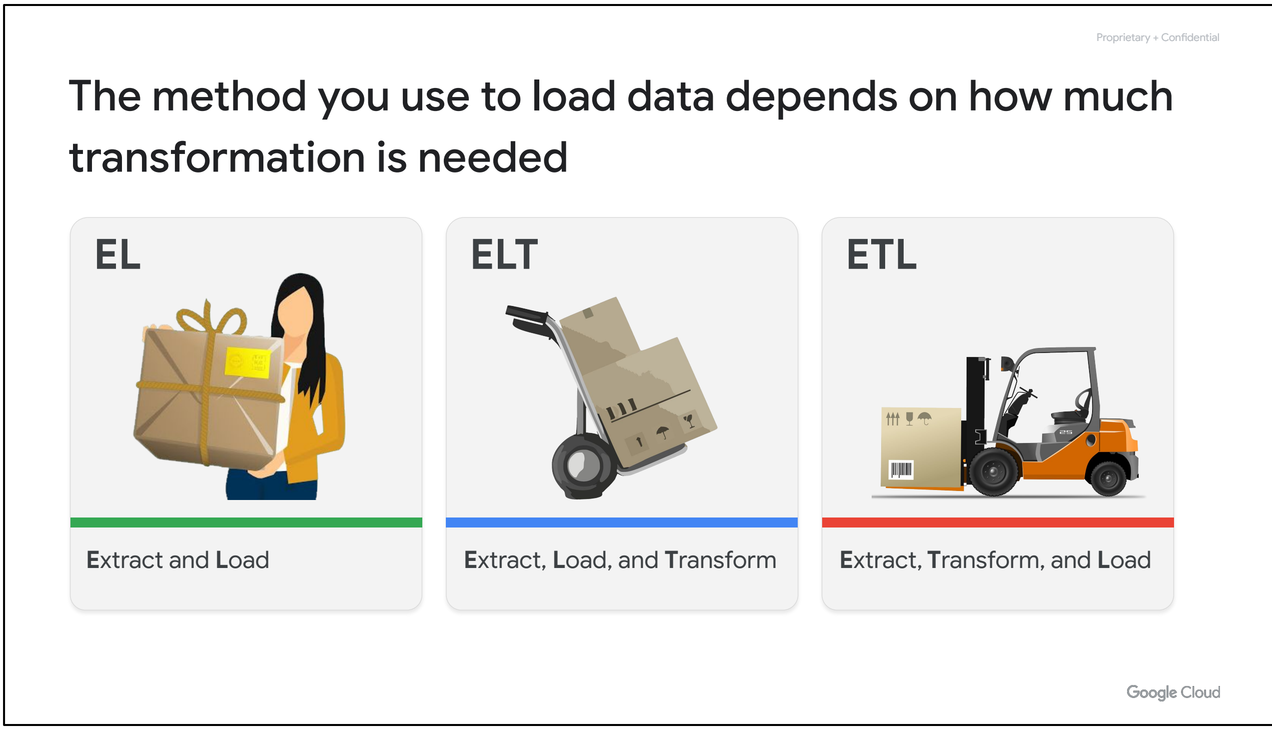


In BigQuery, materialized views periodically cache the results of a query for increased performance and efficiency. BigQuery leverages precomputed results from materialized views and whenever possible reads only delta changes from the base table to compute up-to-date results. Materialized views can be 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.

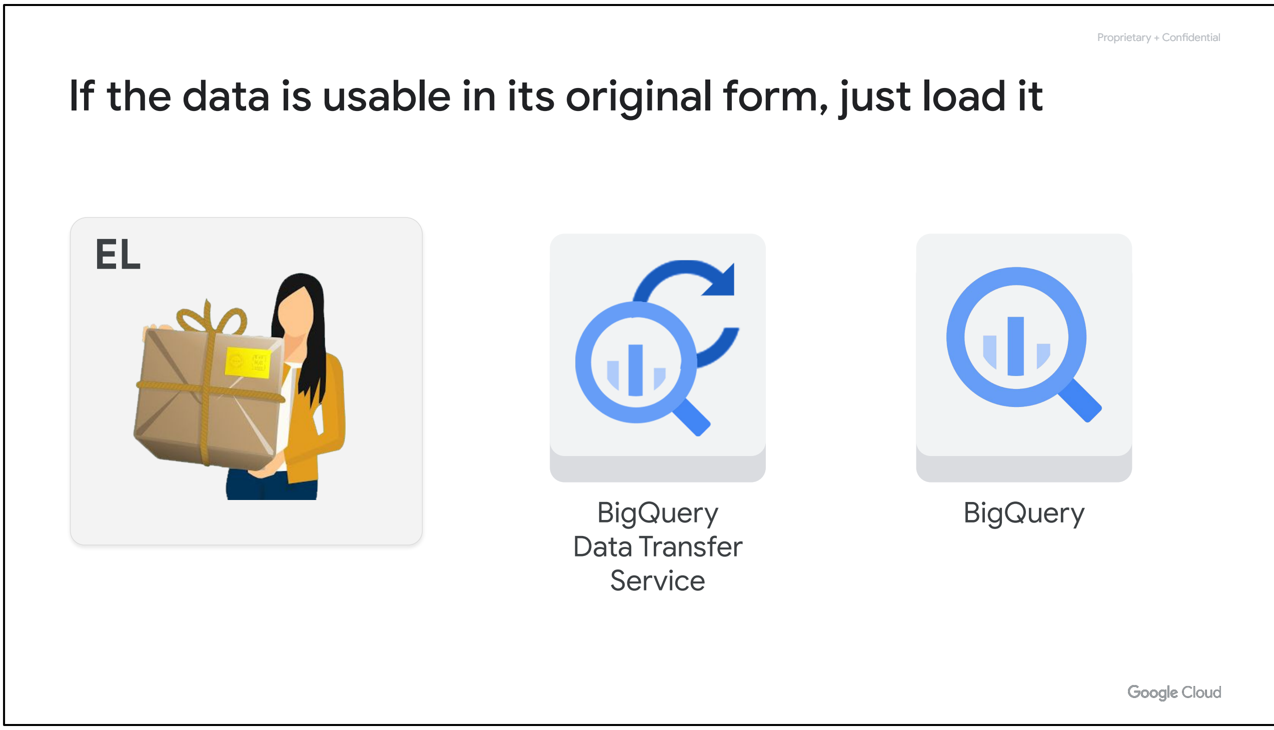
[Materialized views: https://cloud.google.com/bigquery/docs/materialized-views-intro]

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

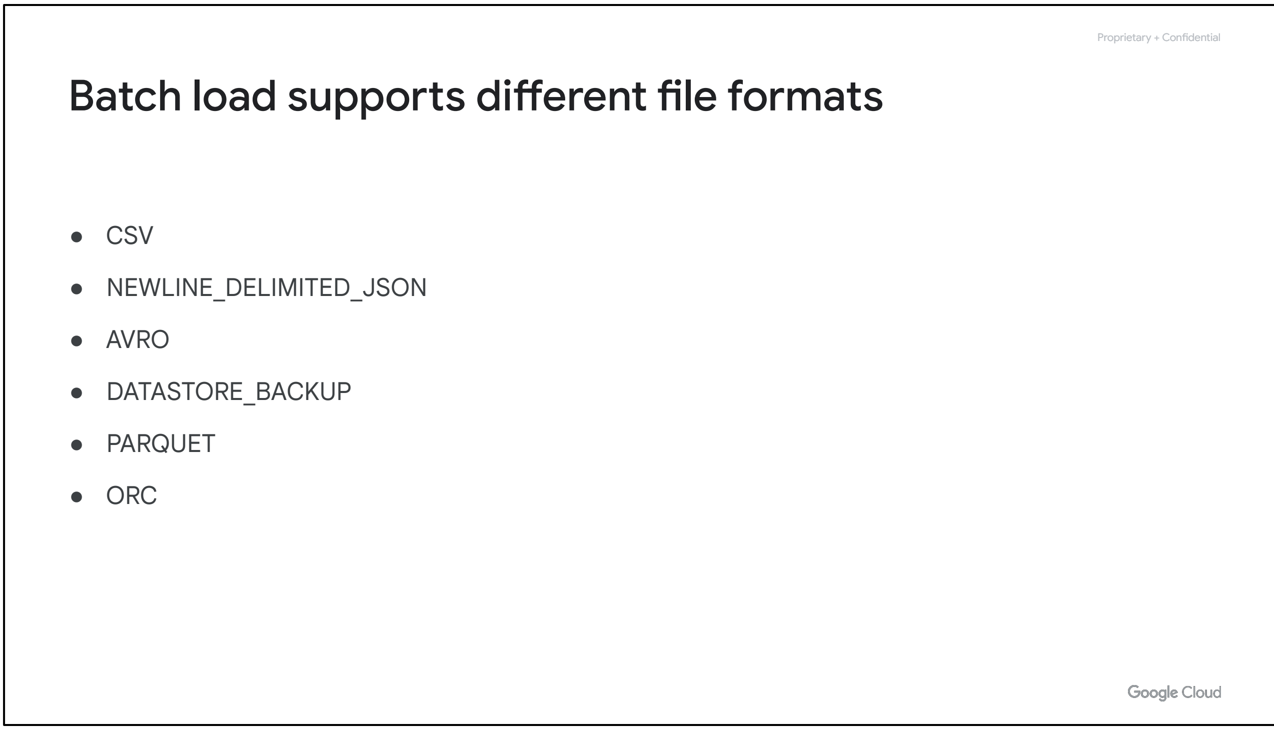


Recall from an earlier module that the method you use to load data depends on how much transformation is needed.

- **E-L, or Extract and Load**, is used when data is imported as-is where the source and target have the same schema.
- **E-L-T, or Extract, Load, Transform**, is used when raw data will be loaded directly into the target and transformed there.
- **E-T-L, 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 E-L. 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_delimiter flag.

BigQuery supports loading gzip 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-constrained, 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:

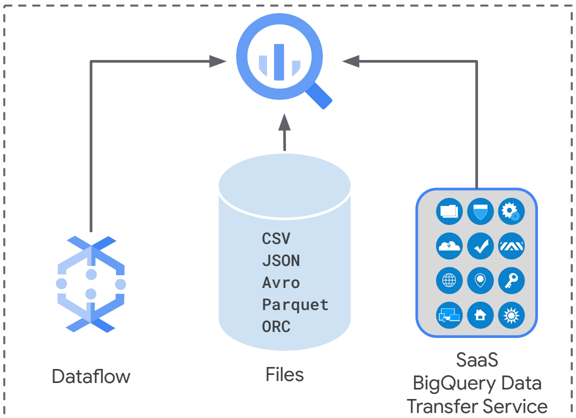
- If your data is in Avro format, which is self-describing, BigQuery can determine the schema directly.
- If the data is in JSON or CSV format, BigQuery can auto-detect the schema, but manual verification is recommended.

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_leading_rows flag to ignore the row. For details, see the documentation on BigQuery load flags.

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 load 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.

Most common is loading data into BigQuery tables (batch, periodic)



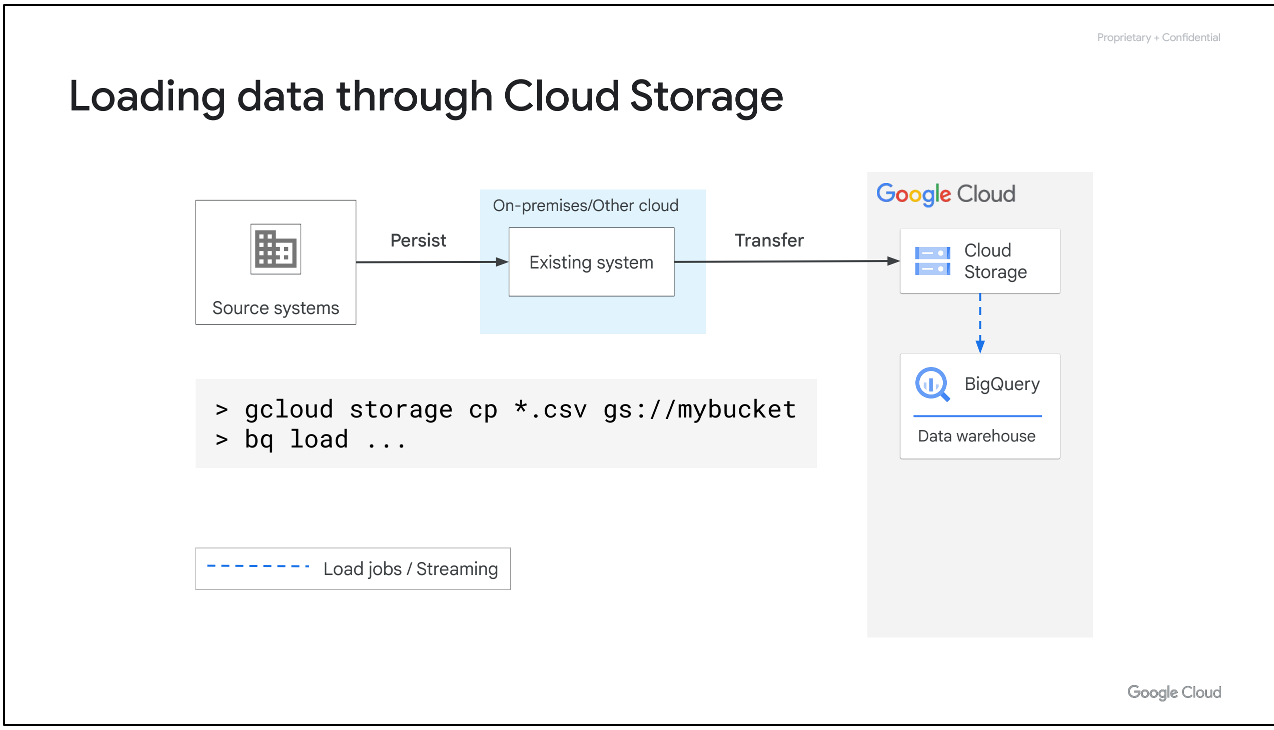
Loading data into BigQuery tables (batch, periodic) offers the best performance.

BigQuery can import data stored in the JSON file format, as long as it is newline 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.

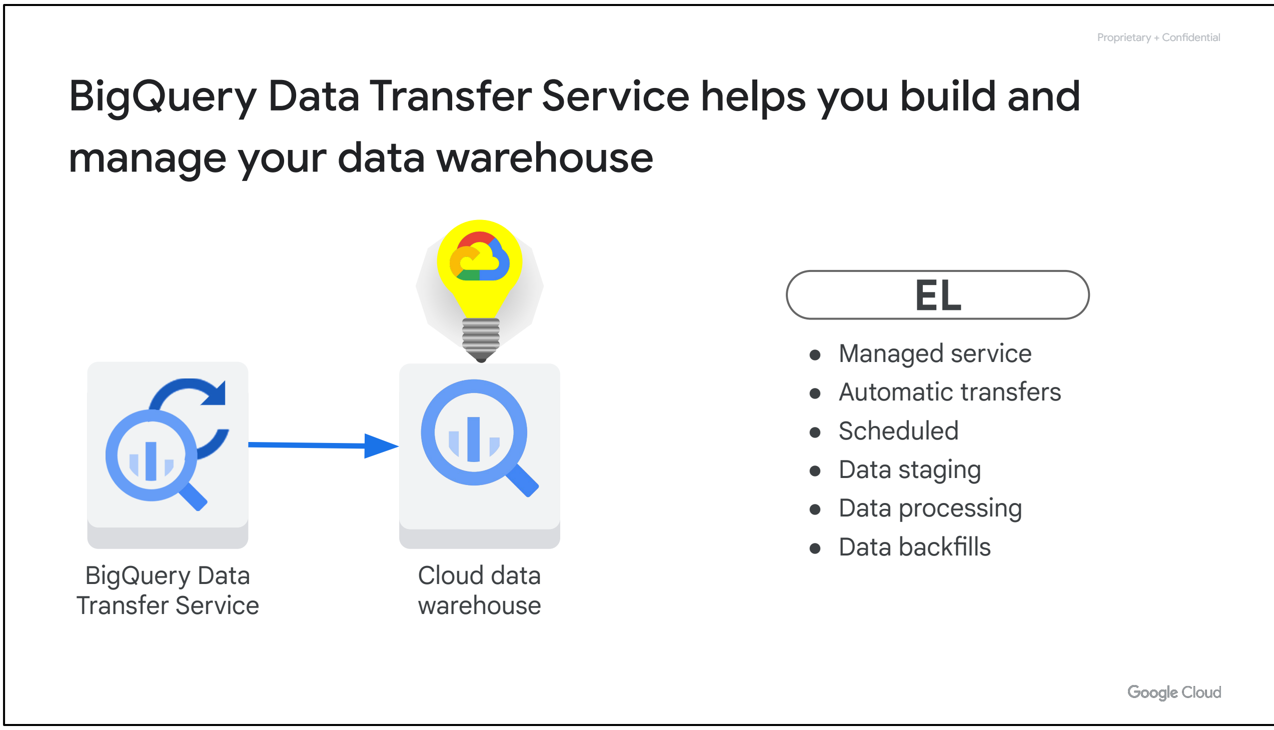
BigQuery can also directly import Firestore and Datastore export files.

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 Dataproc 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.



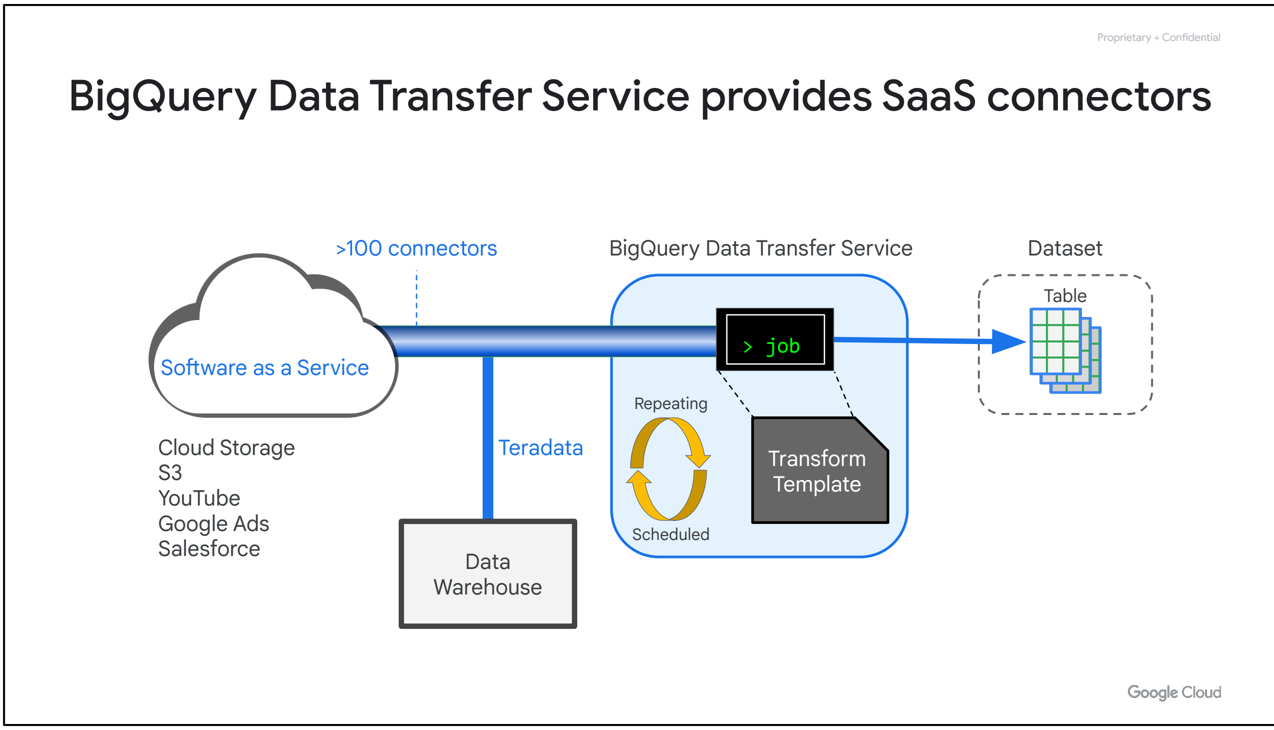
Cloud Storage can be useful in the E-L process. You can transfer files to Cloud Storage in the schema that is native to the existing on-premises data storage and then load those files into BigQuery.



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 SaaS services) in to BigQuery.

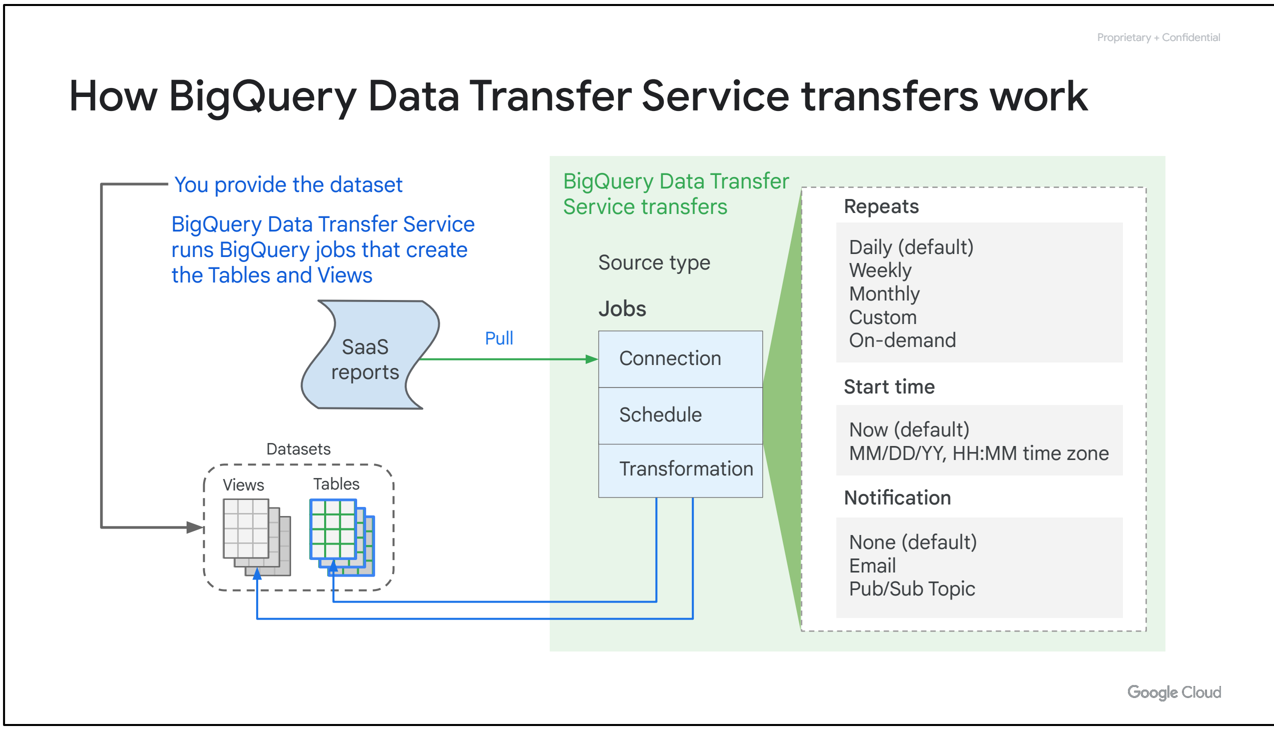
Transfering the data is only the first part of building a data warehouse. If you were assembling your own system, you would need to stage the data so that it can be cleaned (data quality), and transformed (E-L-T, extract, load, transform), and processed (put into its final and stable form). A common 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.

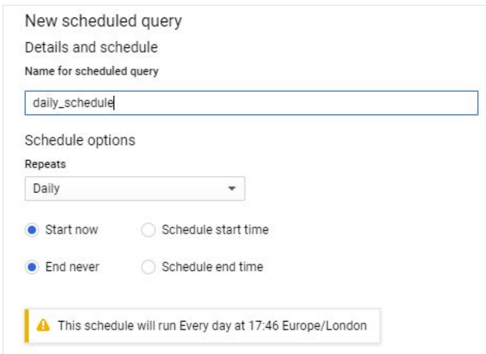
The BigQuery Data Transfer Service can also be used to efficiently move data between regions.

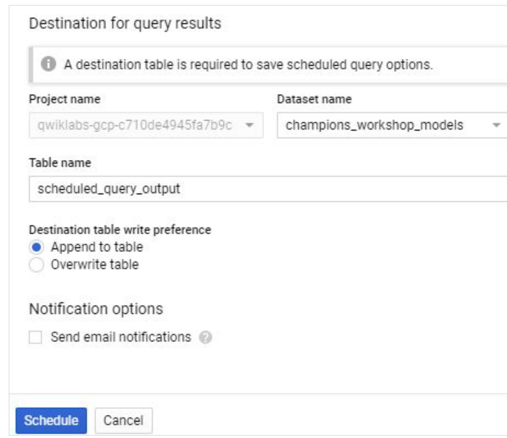


Notice that you don't need Cloud Storage buckets. BigQuery Data Transfer Service runs BigQuery jobs that transform reports from SaaS sources into BigQuery Tables and Views.

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.

Automate the execution of queries based on a schedule

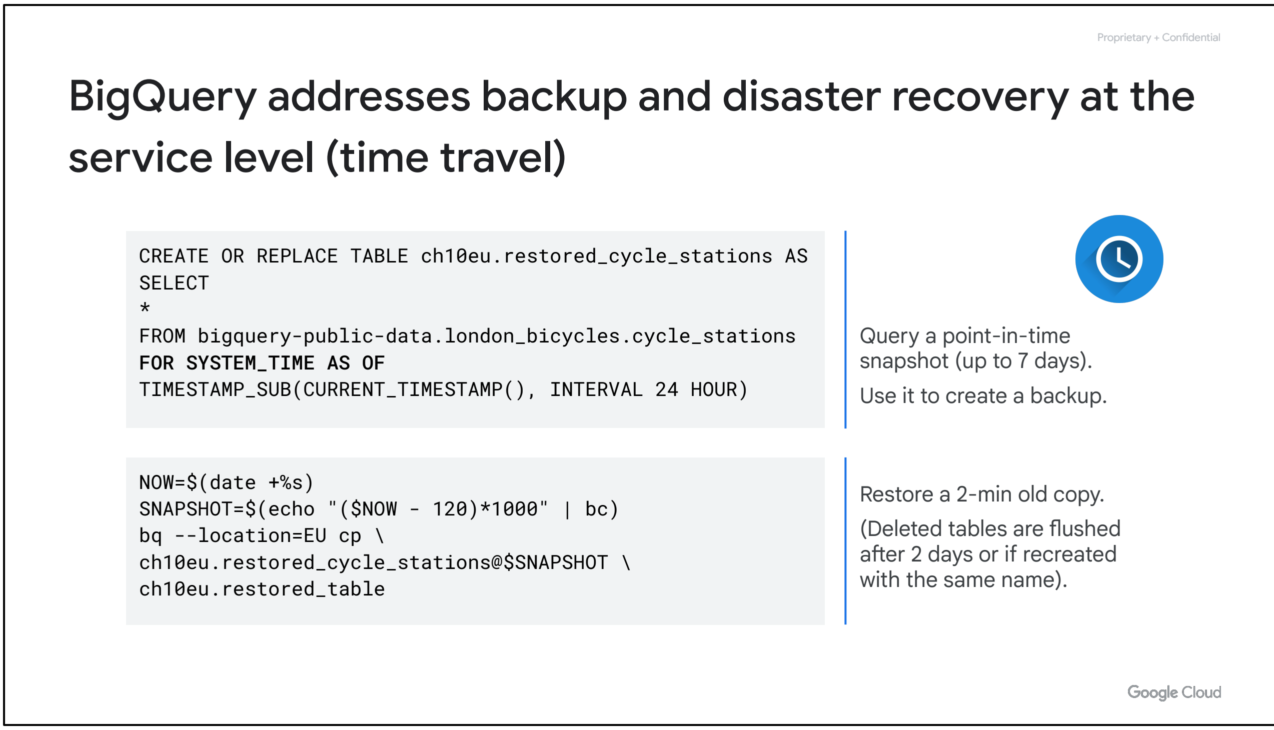




It is a common practice to automate execution of queries based on a schedule or event and cache the results for later consumption.

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.

[https://cloud.google.com/bigquery/docs/scheduling-queries]

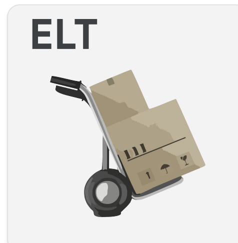


- By maintaining a complete 7-day history of changes against your tables, BigQuery allows you to query a point-in-time snapshot of your data. You can easily revert changes without having to request a recovery from backups. This slide shows how to do a SELECT query to query the table as of 24 hours ago. 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.

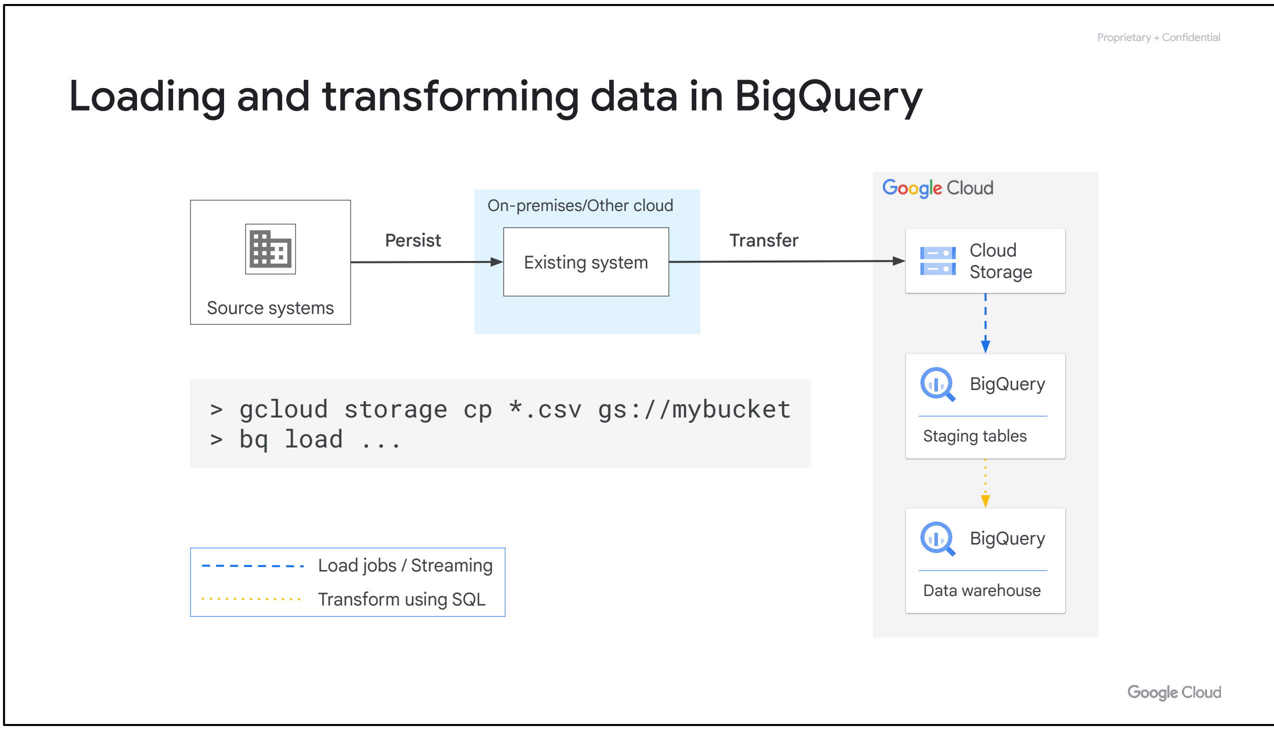
- You can also do this using the BigQuery command-line tool as shown in the second snippet. Here, we're restoring data as of 120 seconds ago.
  You can recover a deleted table only if another table with the same ID in the dataset has not been created. In particular, this means you cannot recover a deleted 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.

If the data requires simple transformations, such as scaling, maybe it can be handled in SQL

BigQuery



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.

Proprietary + Confidential

Modify table data with standard DML statements

INSERT, UPDATE, DELETE, MERGE records into tables

```
UPDATE table_A
SET
  y = table_B.y,
  z = table_B.z + 1
FROM table_B
WHERE table_A.x = table_B.x
  AND table_A.y IS NULL;

INSERT INTO table VALUES (1,2,3), (4,5,6), (7,8,9);

DELETE FROM table WHERE TRUE;
```

Google Cloud

BigQuery supports standard DML statements such as insert, update, delete, and merge. There are no limits on D-M-L statements.

However you should not treat BigQuery as an O-L-T-P system. The underlying infrastructure is not structured to perform optimally as an O-L-T-P. There are other more appropriate products on Google Cloud for such workloads.

[https://cloud.google.com/bigquery/docs/managing-table-schemas]

Create new tables from data with SQL DDL

```
SELECT
    *
FROM
    movielens.movies_raw
WHERE
    movieId < 5;
```

	movieId	title	genres
0	3	Grumpier Old Men (1995)	Comedy|Romance
1	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
2	2	Jumanji (1995)	Adventure|Children|Fantasy
3	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy

```
CREATE OR REPLACE TABLE
movielens.movies AS
SELECT
    * REPLACE(SPLIT(genres,
"|") AS genres)
FROM
    Movielens.movies_raw;
-- Execute multiple statements.
SELECT * FROM movielens.movies;
```

	movieId	title	genres
0	4	Waiting to Exhale (1995)	[Comedy, Drama, Romance]
1	3	Grumpier Old Men (1995)	[Comedy, Romance]
2	2	Jumanji (1995)	[Adventure, Children, Fantasy]
3	1	Toy Story (1995)	[Adventure, Animation, Children, Comedy, Fantasy]

Question: What's the difference between CREATE OR REPLACE TABLE and CREATE TABLE IF NOT EXISTS ? When would you use each?

Google Cloud

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.

Proprietary + Confidential

Custom transformations? BigQuery supports user-defined functions in SQL, JavaScript, and scripting

```
CREATE TEMP FUNCTION multiplyInputs(x float64,y float64)
  RETURNS float64
  LANGUAGE js AS """
    return x*y;
  """;

WITH numbers AS (
  SELECT 1 AS x, 5 AS y
  UNION ALL
  SELECT 2 AS x, 10 AS y
  UNION ALL
  SELECT 3 AS x, 15 AS y)

SELECT x, y, multiplyInputs(x,y) AS product
FROM numbers;
```

Row	x	y	product
1	1	5	5.0
2	2	10	20.0
3	3	15	45.0

Google Cloud

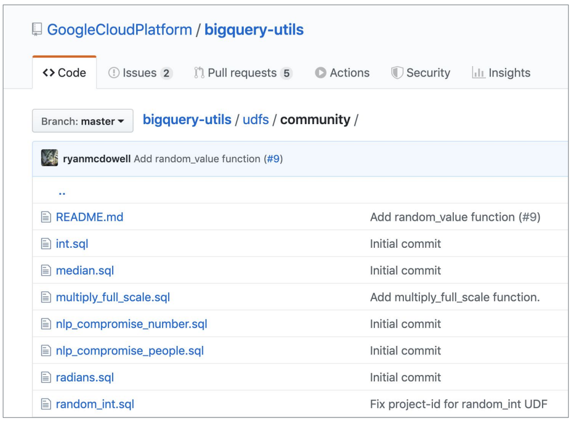
Lastly, what if your transformations went beyond what functions were currently available in BigQuery? Well, you can create your own!

BigQuery supports user-defined functions, or UDF. A UDF enables you to create a function using another SQL expression or an external programming language. JavaScript 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 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.

You can persist and share your UDF objects with other team members or publically



The BigQuery team has a public GitHub repo for common User Defined Functions
https://github.com/GoogleCloudPlatform/bigquery-utils/tree/master/udfs/community

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 publically if you wanted to. The BigQuery team has a public GitHub repo for common User Defined Functions at the link you see here.

[https://cloud.google.com/blog/products/data-analytics/new-persistent-user-defined-functions-increased-concurrency-limits-gis-and-encryption-functions-and-more]

Lab Intro

Loading Data into BigQuery

Now it's time to get some hands-on experience with a lab.

Lab objectives

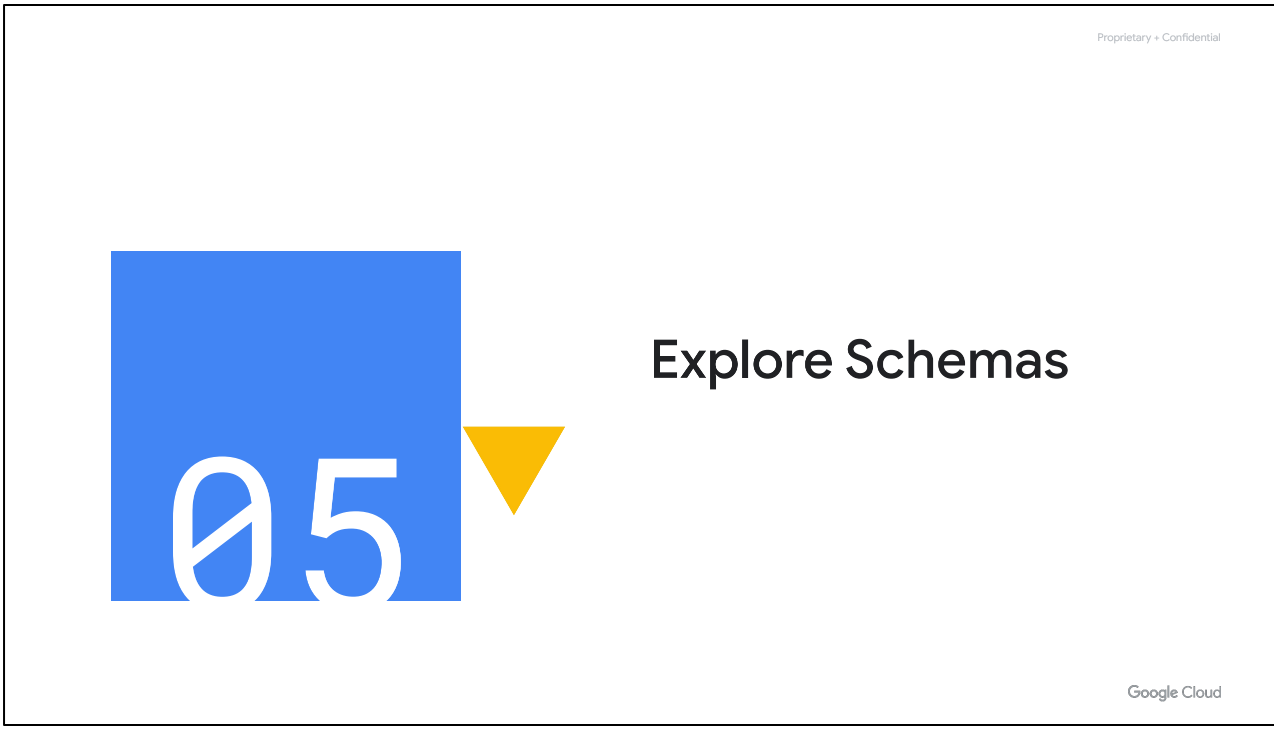
01 Load data into BigQuery from various sources.

02 Load data into BigQuery using the CLI and the Cloud Console.

03 Use DDL to create tables.

Google Cloud

In this lab, you're going to practice loading data into BigQuery. The primary objective of this lab is to load data into BigQuery using both the command-line interface and the Cloud Console. You'll also get experience loading several datasets into BigQuery and using the Data Description Language or DDL.



Now let’s dive into the world of data warehouse schemas.

Designing schemas that scale is a core job of data engineers -- let's explore BigQuery Public Dataset schemas

Public datasets include flights, taxi cab logs, weather recordings, and many more.

https://cloud.google.com/bigquery/public-data/

Google Cloud

Designing efficient schemas that scale is a core job responsibility of any data engineering team. BigQuery hosts many public datasets and schemas for you to explore on popular topics like daily weather readings, taxi cab logs, health data and more.

Let's explore some of these public dataset schemas using SQL.

[https://cloud.google.com/bigquery/public-data/]

Next, we will talk about efficient data warehouse schema design.

Proprietary + Confidential

Transactional databases often use normal form

Original data

Customer	OrderID	Date	Items	
Doug	1600p	8/20/19		
			Product	Quantity
			Caulk	3 boxes
			Soffit	34 meters
			Sealant	2 liters
Tom	221b	10/29/19		
			Product	Quantity
			Sealant	1 liter
			Soffit	17 meters
			Caulk	4 tubes

Normalized data

Orders

Date	OrderID
8/20/2019	1600p
10/29/2018	221b

Order_Items

OrderID	Product	Quantity
1600p	Caulk	3 boxes
221b	Sealant	1 liter
1600p	Soffit	34 meters
221b	Soffit	17 meters
221b	Caulk	4 tubes
1600p	Sealant	2 liters

Google Cloud

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 merged 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.

Data warehouses often denormalize

Normalized data

Orders	
Date	OrderID
08/20/2019	1600p
10/29/2018	221b

Order_Items		
OrderID	Product	Quantity
1600p	Caulk	3 boxes
221b	Sealant	1 liter
1600p	Soffit	34 meters
221b	Soffit	17 meters
221b	Caulk	4 tubes
1600p	Sealant	2 liters

Denormalized flattened data

Customer	OrderID	Date	Product	Quantity
Doug	1600p	08/20/2019	Siding	3 boxes
Doug	1600p	08/20/2019	Caulk	12 tubes
Tom	221b	10/29/2019	Soffit	17 meters
Tom	221b	10/29/2019	Sealant	1 liter
Doug	1600p	08/20/2019	Soffit	34 meters
Tom	221b	10/29/2019	Siding	2 boxes
Tom	221b	10/29/2019	Caulk	4 tubes
Doug	1600p	08/20/2019	Sealant	2 liters

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.

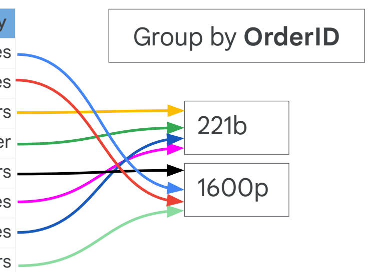
Specifically, denormalizing data enables BigQuery to more efficiently distribute processing among slots, resulting in more parallel processing and better query performance.

You would usually denormalize data before loading it into BigQuery.

Grouping on a 1-to-many field in flattened data can cause shuffling of data over the network

Denormalized flattened table

Customer	OrderID	Date	Product	Quantity
Doug	1600p	08/20/2019	Siding	3 boxes
Doug	1600p	08/20/2019	Caulk	12 tubes
Tom	221b	10/29/2019	Soffit	17 meters
Tom	221b	10/29/2019	Sealant	1 liter
Doug	1600p	09/09/2018	Soffit	34 meters
Tom	221b	10/29/2019	Siding	2 boxes
Tom	221b	10/29/2019	Caulk	4 tubes
Doug	1600p	08/20/2018	Sealant	2 liters



However, there are cases where denormalizing data is bad for performance. Specifically, if you have to group by a column with a 1-to-many relationship. In the example shown, OrderID 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 is slow.

Fortunately, BigQuery supports a method to improve this situation.

Nested and repeated columns improve the efficiency of BigQuery with relational source data

Denormalized flattened table

Customer	OrderID	Date	Product	Quantity
Doug	1600p	08/20/2019	Siding	3 boxes
Doug	1600p	08/20/2019	Caulk	12 tubes
Tom	221b	10/29/2019	Soffit	17 meters
Tom	221b	10/29/2019	Sealant	1 liter
Doug	1600p	8/20/2019	Soffit	34 meters
Tom	221b	10/29/2019	Siding	2 boxes
Tom	221b	10/29/2019	Caulk	4 tubes
Doug	1600p	08/20/2019	Sealant	2 liters

Denormalized with nested and repeated data

Order.ID	Order.Date	Order.Product	Order.Quantity
1600p	08/20/2019	Siding	3 boxes
		Caulk	12 tubes
		Soffit	34 meters
		Sealant	2 liters
221b	10/29/2019	Soffit	17 meters
		Sealant	1 liter
		Siding	2 boxes
		Caulk	4 tubes

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. OrderID 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 colocated, 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.

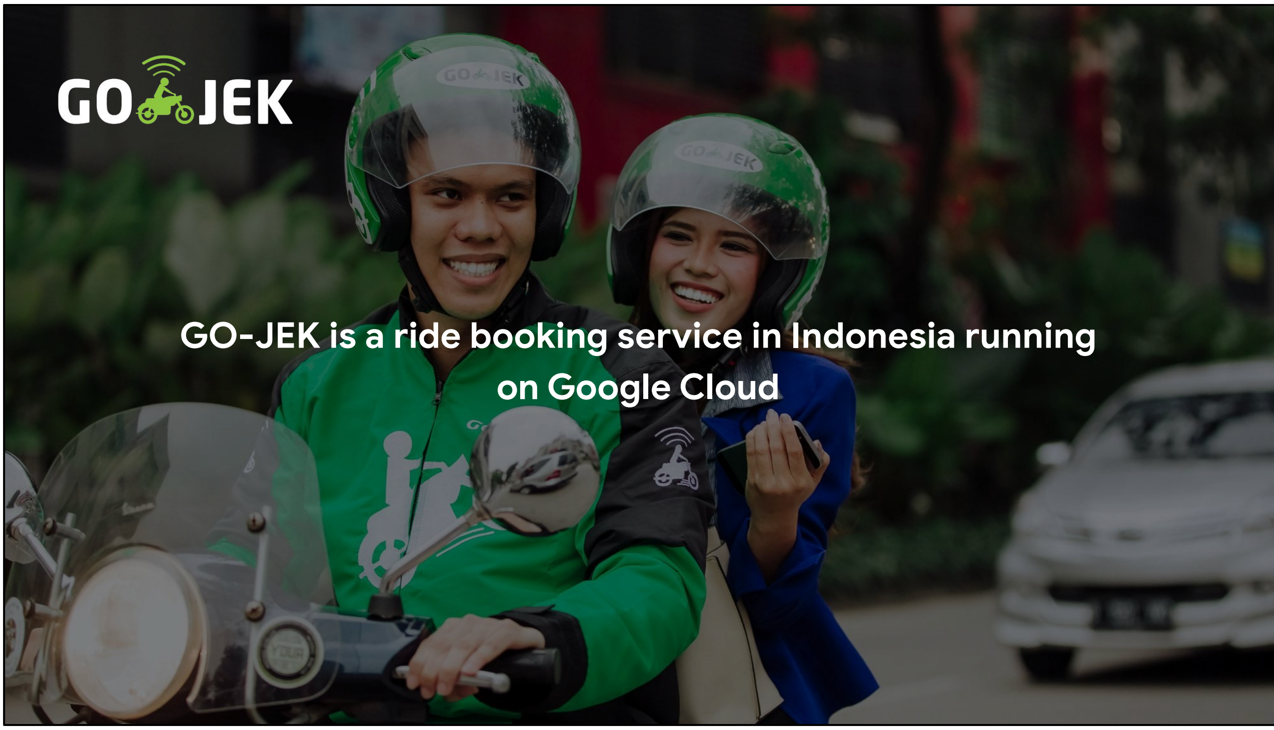
Nested columns can be understood as a form of repeated field. It preserves the relational qualities of the original data and schema while enabling columnar and parallel processing of the repeated nested fields. It is the best alternative for data that already has a relational pattern to it. Turning the relation into a nested or repeated field improves BigQuery performance.

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

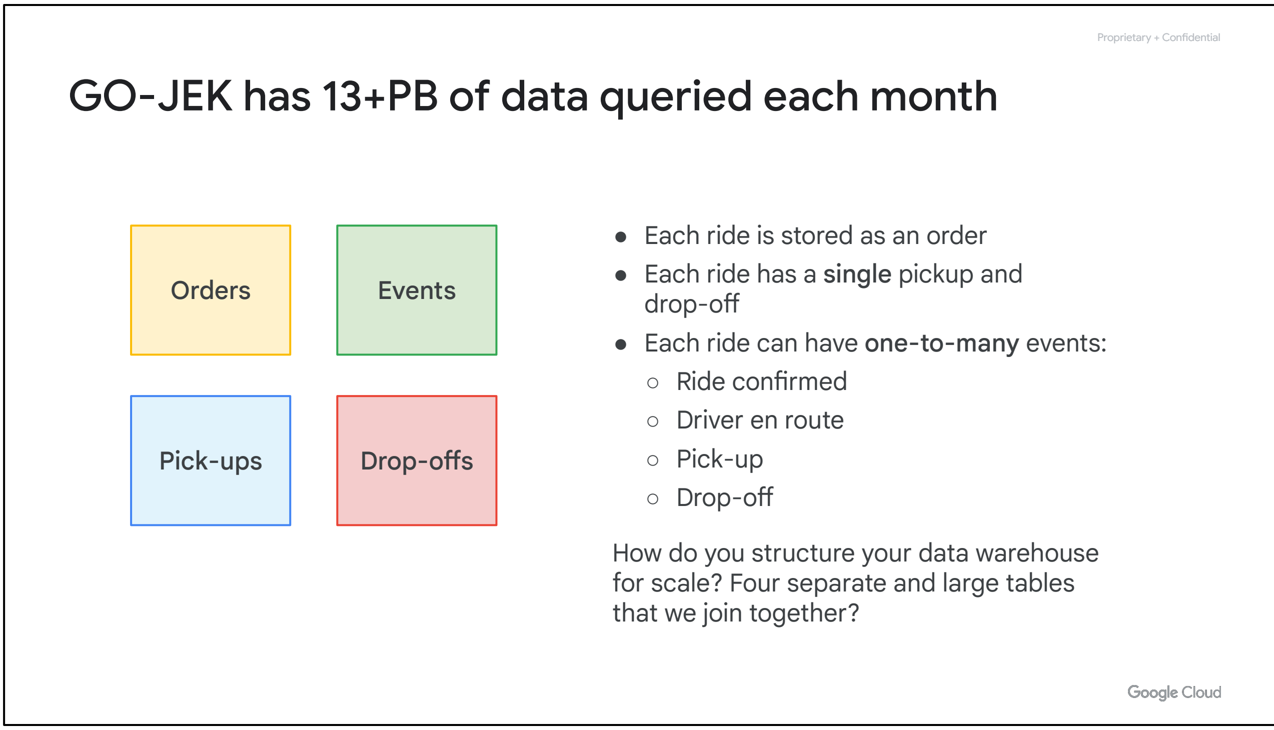
Look for nested and repeated fields whenever BigQuery is used in a hybrid solution in conjunction with traditional databases, 07

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.



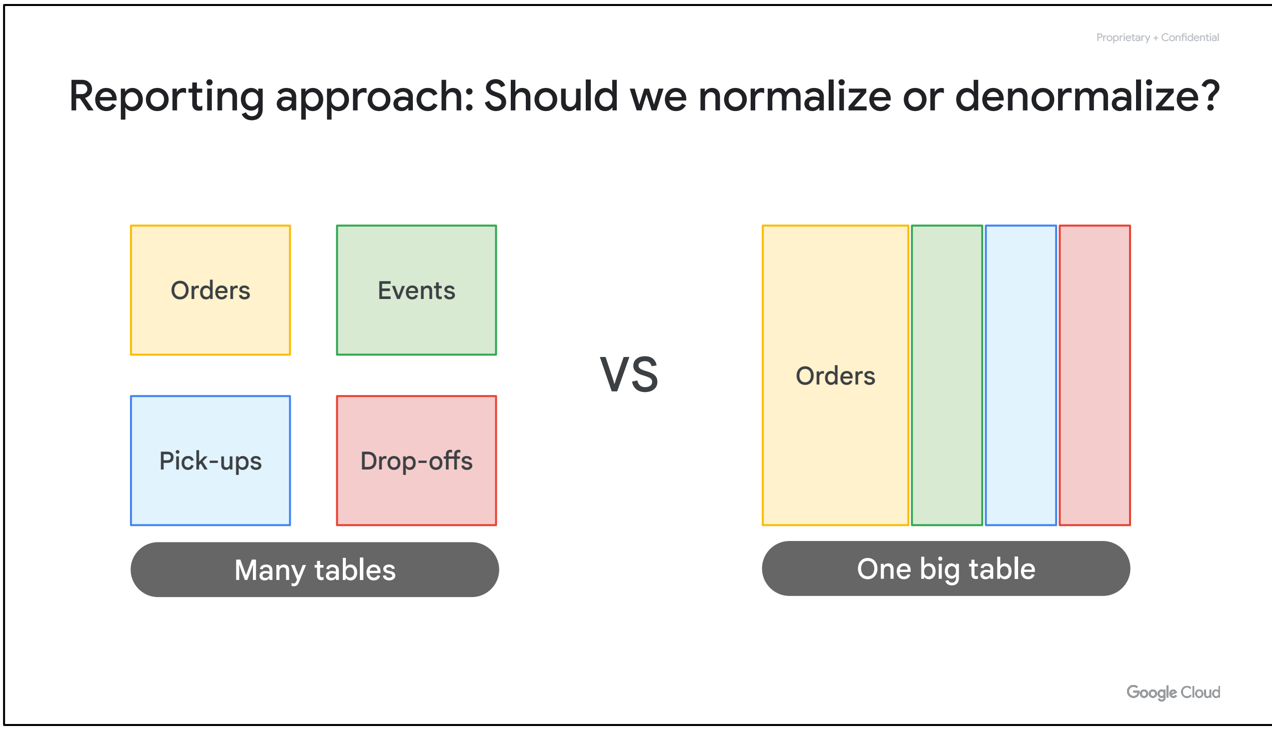
I’ll illustrate by using an example from a real business running on Google Cloud. GO-JEK is a company in Indonesia that is well known for its ride booking service…



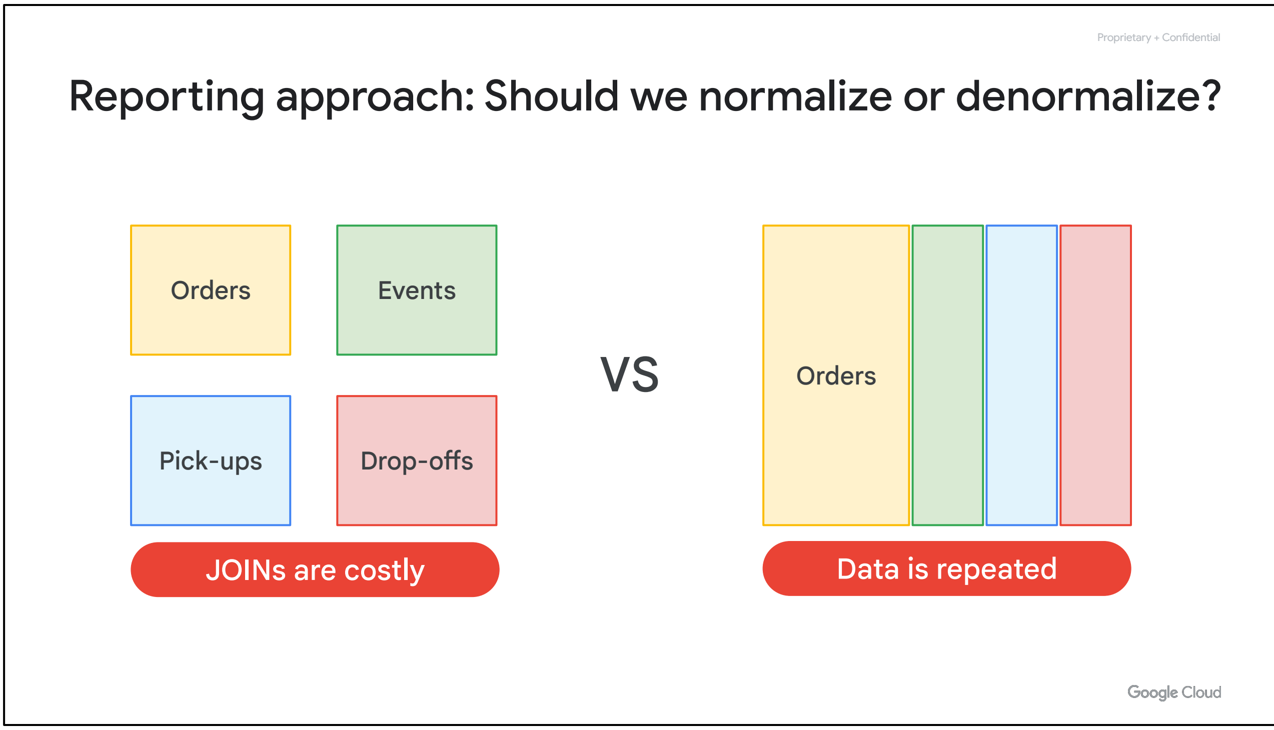
… and they process over a 13 petabyte of data on BigQuery per month from queries to support business decisions. What kind of decisions?

For GO-JEK, they track whenever a new customer places an order like hails a ride with their mobile app. That order is stored in an orders table. Each order has a single pick-up location and drop-off destination. For a single order, you could have one or many events like “Ride Ordered”, “Ride Confirmed”, “Drive En Route”, “Drop off Complete” etc.

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.

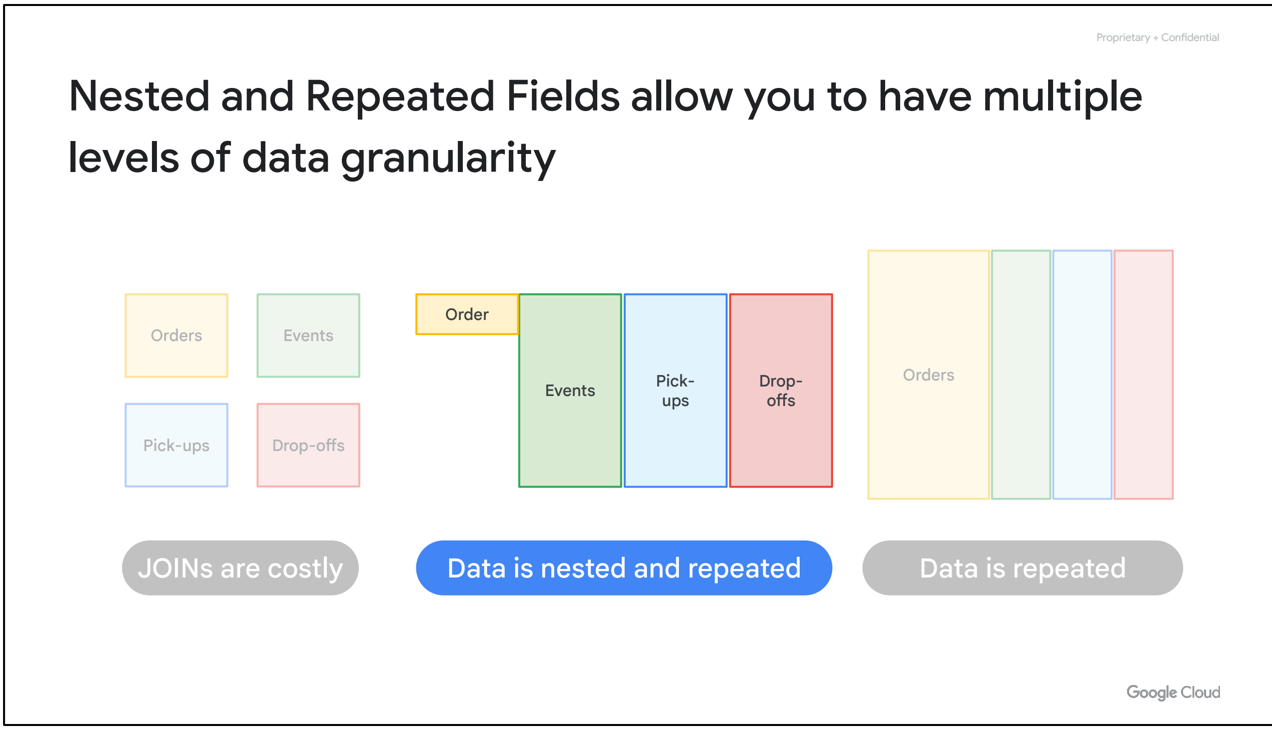


Well as you saw earlier, we could store one fact in one place with the normalization route which is typical for relational systems. Or we could go the fully denormalized route and just store all levels of granularity in a single big table where you would have one OrderID like ‘123’ repeated in a row for each event that happens on that order. Faster for querying sure but what are the drawbacks?



For relational schemas (normalized schemas) often the most intensive computational workloads are JOINS across very large tables. Remember RDBMS' 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 a 10+ 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 OrderID like '123'? It is duplicated for each event on 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 OrderIDs. 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.

Store complex data with nested fields (ARRAYS)

Row	order_id	service_type	payment_method	event.status	event.time	pickup.latitude	pickup.longitude	destination.latitude	destination.longitude	total_distance_km	pricing_type
151	FD-5117	GO_FOOD	GOPAY	CREATED	2018-12-31 04:44:02.545210 UTC	-7.75105	110.410561	-7.7430367	110.4046433	1.56	regular
				COMPLETED	2018-12-31 05:06:27.897769 UTC						
				PICKED_UP	2018-12-31 04:48:25.945331 UTC						
				DRIVER_FOUND	2018-12-31 04:44:06.869010 UTC						
152	FD-6834	GO_FOOD	CASH	PICKED_UP	2018-12-31 12:49:52.518880 UTC	1.121272	104.049739	1.1368655	104.03322	4.84	surge
				DRIVER_FOUND	2018-12-31 12:40:14.214843 UTC						
				COMPLETED	2018-12-31 13:04:00.291780 UTC						
				CREATED	2018-12-31 12:40:13.431094 UTC						
153	FD-6293	GO_FOOD	PARTIAL_PAYMENT	PICKED_UP	2018-12-31 04:33:11.856445 UTC	-7.9657554	112.6247491	-7.9384084	112.6227862	4.68	regular
				COMPLETED	2018-12-31 04:56:05.885521 UTC						
				CREATED	2018-12-31 04:16:24.356539 UTC						
				DRIVER_FOUND	2018-12-31 04:16:25.643766 UTC						
154	FD-7817	GO_FOOD	CASH	COMPLETED	2018-12-31 09:14:44.897136 UTC	-6.353915	106.247312	-6.368896	106.25787	3.51	regular
				PICKED_UP	2018-12-31 09:01:11.471274 UTC						
				CREATED	2018-12-31 08:40:31.821796 UTC						
				DRIVER_FOUND	2018-12-31 08:40:32.910319 UTC						

Table JSON

First < Prev Rows 151 - 154 of 1137 Next > Last

Google Cloud

Here you see it clearly. Shown here on screen are just 4 rows for 4 unique OrderIDs. Notice all that grey 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 if 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.status and event.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?

Report on all data in once place with STRUCTS

Row	order_id	service_type	payment_method	event.status	event.time	pickup.latitude	pickup.longitude	destination.latitude	destination.longitude	total_distance_km	pricing_type
151	FD-5117	GO_FOOD	GOPAY	CREATED	2018-12-31 04:44:02.545210 UTC	-7.75105	110.410561	-7.7430367	110.4046433	1.56	regular
				COMPLETED	2018-12-31 05:06:27.897769 UTC						
				PICKED_UP	2018-12-31 04:48:25.945331 UTC						
				DRIVER_FOUND	2018-12-31 04:44:06.869010 UTC						
152	FD-6834	GO_FOOD	CASH	PICKED_UP	2018-12-31 12:49:52.518880 UTC	1.121272	104.049739	1.1368655	104.03322	4.84	surge
				DRIVER_FOUND	2018-12-31 12:40:14.214843 UTC						
				COMPLETED	2018-12-31 13:04:00.291780 UTC						
				CREATED	2018-12-31 12:40:13.431094 UTC						
153	FD-6293	GO_FOOD	PARTIAL_PAYMENT	PICKED_UP	2018-12-31 04:33:11.856445 UTC	-7.9657554	112.6247491	-7.9384084	112.6227862	4.68	regular
				COMPLETED	2018-12-31 04:56:05.885521 UTC						
				CREATED	2018-12-31 04:16:24.356539 UTC						
				DRIVER_FOUND	2018-12-31 04:16:25.643766 UTC						
154	FD-7817	GO_FOOD	CASH	COMPLETED	2018-12-31 09:14:44.897136 UTC	-6.353915	106.247312	-6.368896	106.25787	3.51	regular
				PICKED_UP	2018-12-31 09:01:11.471274 UTC						
				CREATED	2018-12-31 08:40:31.821796 UTC						
				DRIVER_FOUND	2018-12-31 08:40:32.910319 UTC						

Table JSON

First < Prev Rows 151 - 154 of 1137 Next > Last

Google Cloud

Event, pickup, and destination are what are called STRUCT or structure 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.

Nested ARRAY fields and STRUCT fields allow for differing data granularity in the same table

Row	order_id	service_type	payment_method	event.status	event.time	pickup.latitude	pickup.longitude	destination.latitude	destination.longitude	total_distance_km	pricing_type
151	FD-5117	GO_FOOD	GOPAY	CREATED	2018-12-31 04:44:02.545210 UTC	-7.75105	110.410561	-7.7430367	110.4046433	1.56	regular
				COMPLETED	2018-12-31 05:06:27.897769 UTC						
				PICKED_UP	2018-12-31 04:48:25.945331 UTC						
				DRIVER_FOUND	2018-12-31 04:44:06.869010 UTC						
152	FD-6834	GO_FOOD	CASH	PICKED_UP	2018-12-31 12:49:52.518880 UTC	1.121272	104.049739	1.1368655	104.03322	4.84	surge
				DRIVER_FOUND	2018-12-31 12:40:14.214843 UTC						
				COMPLETED	2018-12-31 13:04:00.291780 UTC						
				CREATED	2018-12-31 12:40:13.431094 UTC						
153	FD-6293	GO_FOOD	PARTIAL_PAYMENT	PICKED_UP	2018-12-31 04:33:11.856445 UTC	-7.9657554	112.6247491	-7.9384084	112.6227862	4.68	regular
				COMPLETED	2018-12-31 04:56:05.885521 UTC						
				CREATED	2018-12-31 04:16:24.356539 UTC						
				DRIVER_FOUND	2018-12-31 04:16:25.643766 UTC						
154	FD-7817	GO_FOOD	CASH	COMPLETED	2018-12-31 09:14:44.897136 UTC	-6.353915	106.247312	-6.368896	106.25787	3.51	regular
				PICKED_UP	2018-12-31 09:01:11.471274 UTC						
				CREATED	2018-12-31 08:40:31.821796 UTC						
				DRIVER_FOUND	2018-12-31 08:40:32.910319 UTC						

Table JSON

First < Prev Rows 151 - 154 of 1137 Next > Last

Google Cloud

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 its much faster to query.

People often ask -- with really wide schemas (like a hundred 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 a COUNT(order_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.

Your turn

- Practice reading the new schema
- Spot the STRUCTS
- Type RECORD = STRUCTS

Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event.**status**	STRING	NULLABLE
event.**time**	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE
pickup.**latitude**	FLOAT	NULLABLE
pickup.**longitude**	FLOAT	NULLABLE
destination	RECORD	NULLABLE
destination.**latitude**	FLOAT	NULLABLE
destination.**longitude**	FLOAT	NULLABLE
total_distance_km	FLOAT	NULLABLE
pricing_type	STRING	NULLABLE
duration	RECORD	NULLABLE
duration.**booking_to_dispatch**	FLOAT	NULLABLE
duration.**booking_to_pickup**	FLOAT	NULLABLE

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?

Practice reading the new schema

- Practice reading the new schema
- Spot the STRUCTS
- Type RECORD = STRUCTS

Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event.**status**	STRING	NULLABLE
event.**time**	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE
pickup.**latitude**	FLOAT	NULLABLE
pickup.**longitude**	FLOAT	NULLABLE
destination	RECORD	NULLABLE
destination.**latitude**	FLOAT	NULLABLE
destination.**longitude**	FLOAT	NULLABLE
total_distance_km	FLOAT	NULLABLE
pricing_type	STRING	NULLABLE
duration	RECORD	NULLABLE
duration.**booking_to_dispatch**	FLOAT	NULLABLE
duration.**booking_to_pickup**	FLOAT	NULLABLE

Events

Pick-ups

Destination

Duration

Google Cloud

Here are the four STRUCTs 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.

Your turn

- Practice reading the new schema
- Spot the ARRAYS
- Hint: Look at Mode

Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event.**status**	STRING	NULLABLE
event.**time**	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE

Find the ARRAY data types in this schema. As a hint, look at the Mode and find the REPEATED values.

Got them?

Proprietary + Confidential

Practice reading the new schema

- Practice reading the new schema
- Spot the ARRAYS
- REPEATED = ARRAY

Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event.**status**	STRING	NULLABLE
event.**time**	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE

Events

Row	order_id	service_type	payment_method	event.status	event.time
151	FD-5117	GO_FOOD	GOPAY	CREATED	2018-12-31 04:44:02.545210 UTC
				COMPLETED	2018-12-31 05:06:27.897769 UTC

Status and Time in an ARRAY of Event STRUCTs

Google Cloud

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 it's 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.

Proprietary + Confidential

Recap

- STRUCTS (RECORD)
- ARRAYS (REPEATED)
- ARRAYS can be part of regular fields or STRUCTS
- A single table can have many STRUCTS

Field name	Type	Mode
order_id	STRING	NULLABLE
service_type	STRING	NULLABLE
payment_method	STRING	NULLABLE
event	RECORD	REPEATED
event.**status**	STRING	NULLABLE
event.**time**	TIMESTAMP	NULLABLE
pickup	RECORD	NULLABLE
pickup.**latitude**	FLOAT	NULLABLE
pickup.**longitude**	FLOAT	NULLABLE
destination	RECORD	NULLABLE
destination.**latitude**	FLOAT	NULLABLE
destination.**longitude**	FLOAT	NULLABLE
total_distance_km	FLOAT	NULLABLE
pricing_type	STRING	NULLABLE
duration	RECORD	NULLABLE
duration.**booking_to_dispatch**	FLOAT	NULLABLE
duration.**booking_to_pickup**	FLOAT	NULLABLE

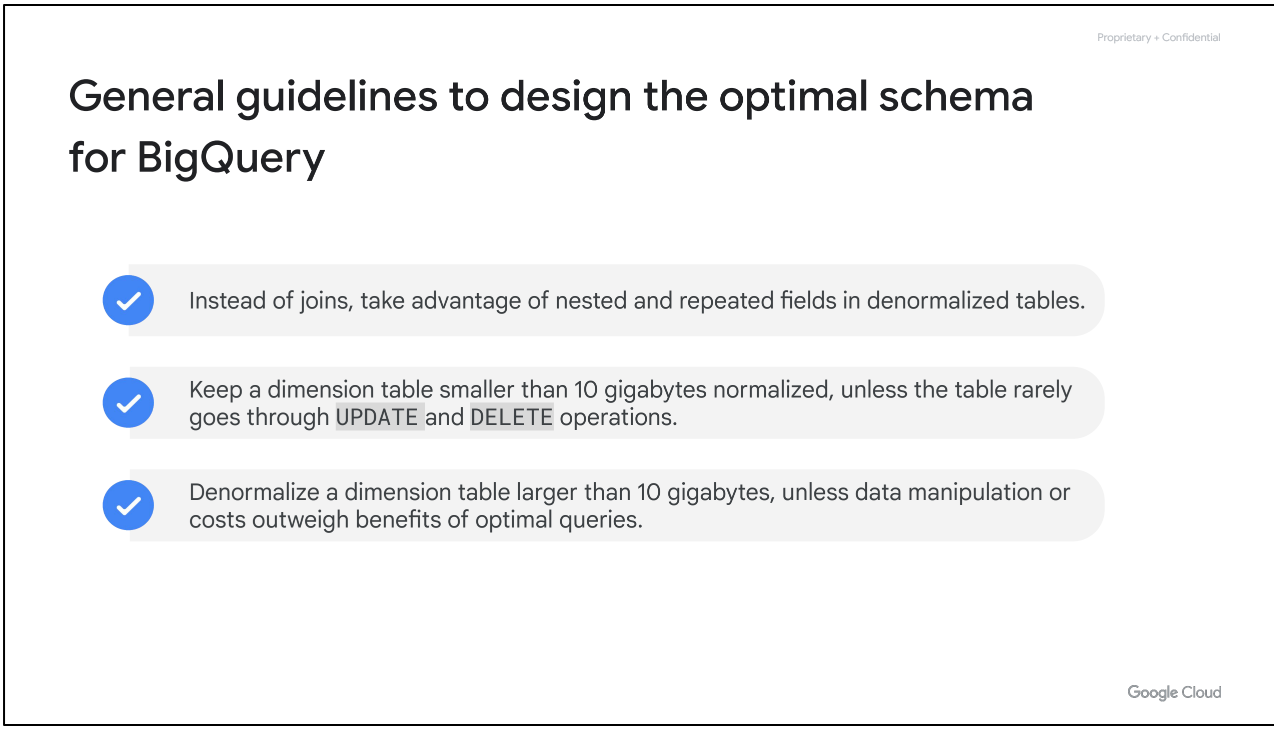
Google Cloud

So here's the cheatsheet.

- STRUCTs are of type 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, etc.
- 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.

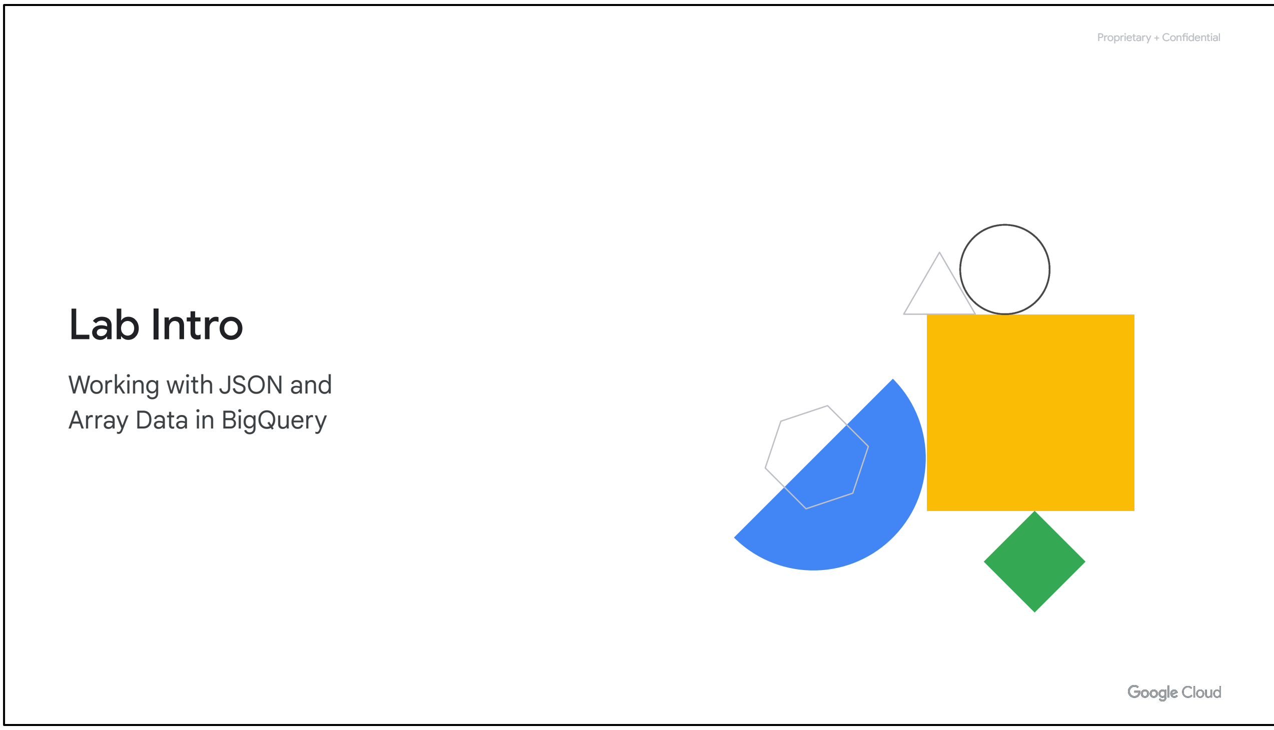
We've been talking a lot about nested and repeated fields so you're probably wondering what to do with your existing star-schema, snowflake and third normal form data. The great news is that BigQuery also works well with those schema types!

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.



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 dataset's 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 GB. If your tables are less than 10 GB, keep the tables separate and do a join.



In the next lab, you'll get some experience working with JSON and array data in BigQuery.

Lab objectives

01 Load semi-structured JSON data into BigQuery

02 Learn how to create and query arrays and structs

03 Query nested and repeated fields

Google Cloud

The objectives of this lab are to load semi-structured JSON data into BigQuery, and to learn how to create and query arrays and structs. You will also query nested and repeated fields.

Proprietary + Confidential

08

Optimize with Partitioning and Clustering

Google Cloud

Next up is optimizing with partitioning and clustering.

Spot the difference!

```sql
SELECT
  wiki,
  SUM(views) AS views
FROM
  `fh-bigquery.wikipedia_v2.pageviews_2017`
WHERE
  datehour >= '2017-01-01'
  AND wiki IN('en','en.m')
  AND title = 'Kubernetes'
GROUP BY
  wiki
ORDER BY
  wiki
```

v2 is partitioned

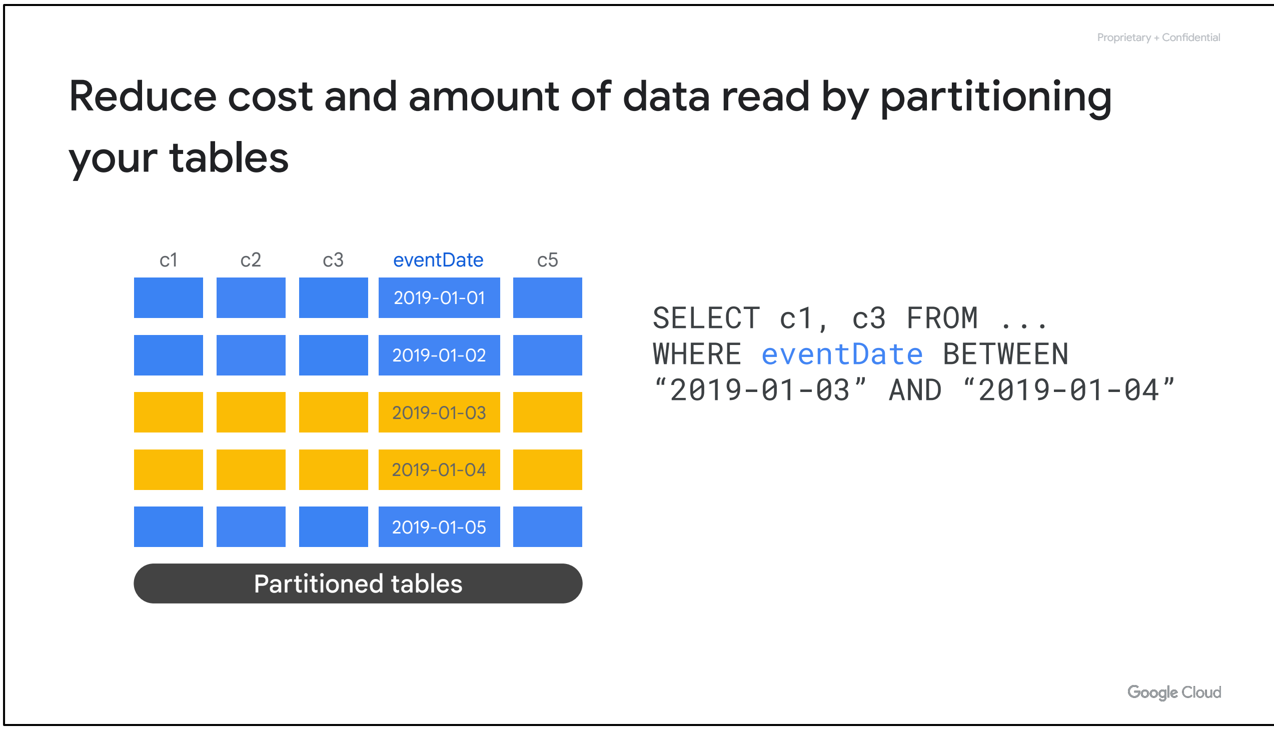
Duration	20 sec
Bytes processed	2.2 TB

```sql
SELECT
  wiki,
  SUM(views) AS views
FROM
  `fh-bigquery.wikipedia_v3.pageviews_2017`
WHERE
  datehour >= '2017-01-01'
  AND wiki IN('en','en.m')
  AND title = 'Kubernetes'
GROUP BY
  wiki
ORDER BY
  wiki
```

v3 is partitioned and clustered

Duration	5 sec
Bytes processed	227.76 GB

In a table partitioned by a date or 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 partitioned 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 eventDate column. BigQuery will then change its internal storage so the dates are stored in separate shards.

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.

BigQuery supports three ways of partitioning tables

Ingestion time

```
bq query --destination_table mydataset.mytable
--time_partitioning_type=DAY
...
```

Any column that is of type DATETIME, DATE or TIMESTAMP

```
bq mk --table --schema a:STRING,tm:TIMESTAMP
--time_partitioning_field tm
```

Integer-typed column

```
bq mk --table --schema "customer_id:integer,value:integer"
--range_partitioning=customer_id,0,100,10
my_dataset.my_table
```


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 datetime column, or based on a range of an integer column. Here, we are partitioning customer_id in the range 0 to 100 in increments of 10.

Proprietary + Confidential

Partitioning can improve query cost and performance by reducing data being queried

```
SELECT
  field1
FROM
  mydataset.table1
WHERE
  _PARTITIONTIME > TIMESTAMP_SUB(TIMESTAMP('2016-04-15'), INTERVAL 5 DAY)
```

Isolate the partition field in the left-hand side of the query expression!

```
bq query \
--destination_table mydataset.mytable
--time_partitioning_type=DAY --require_partition_filter
...
```

Google Cloud

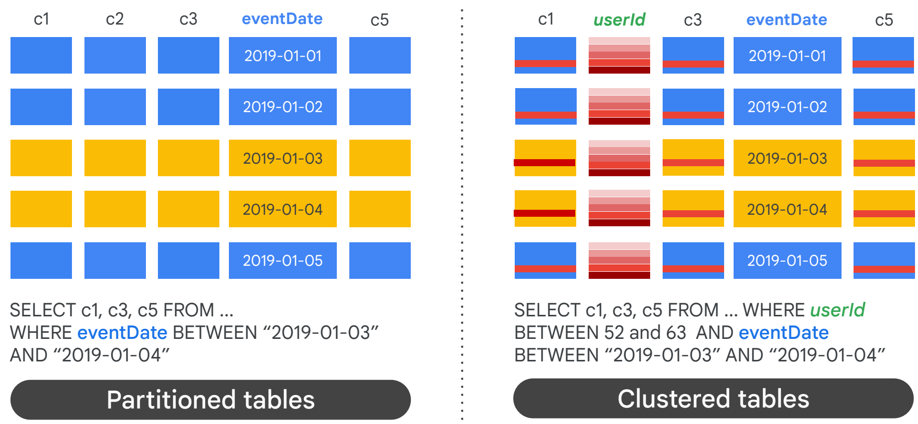
Although more metadata must be maintained, by ensuring that data is partitioned globally, BigQuery can more accurately estimate the bytes processed by a query before you run it. This cost calculation provides an upper bound on the final cost of the query.

A good practice is to require that queries always include the partition filter.

Make sure that the partition field is isolated on the left-hand-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*
https://medium.com/google-cloud/bigquery-optimized-cluster-your-tables-65e2f684594b

BigQuery automatically sorts the data based on values in the clustering columns

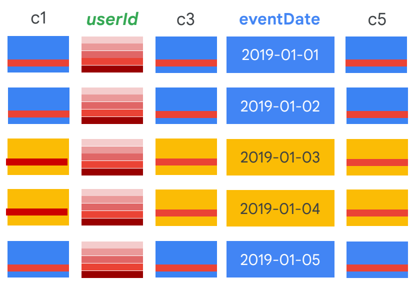


Clustering can improve the performance of certain types of queries, such as queries that use filter clauses and those that aggregate data. When data is written to a clustered table by a query or load job, BigQuery sorts the data using the values in the clustering columns. These values are used to organize 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 in the clustering columns, performance is improved because the sorted blocks co-locate rows with similar values.

In this example, the table is partitioned by eventDate and clustered by userId. Now, because the query looks for partitions in a specific range, only 2 of the 5 partitions are considered. Because the query looks for userId in a specific range, BigQuery can jump to the row range and read only those rows for each of the columns needed.

Set up clustering at table creation time



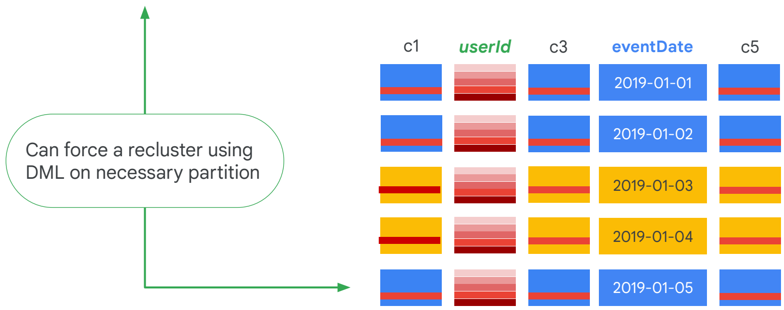

```
CREATE TABLE mydataset.myclusteredtable
 (
   c1 NUMERIC,
   userId STRING,
   c3 STRING,
   eventDate TIMESTAMP,
   C5 GEOGRAPHY
  )
 PARTITION BY DATE(eventDate)
 CLUSTER BY userId
 OPTIONS (
   partition_expiration_days=3,
   description="cluster")
 AS SELECT * FROM mydataset.myothertable
```


You set up clustering at table creation time. Here, we are creating the table, partitioning by eventDate, and clustering by userId. We are also telling BigQuery to expire partitions that are more than 3 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.

In streaming tables, the sorting fails over time, and so BigQuery has to recluster

```
UPDATE ds.table
SET c1 = 300
WHERE c1 = 300
AND eventDate > TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 DAY)
```



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 * query that selects from and overwrites the table, but guess what! You don't need to do that anymore.

Proprietary + Confidential

BigQuery will automatically recluster your data

Automatic re-clustering

free

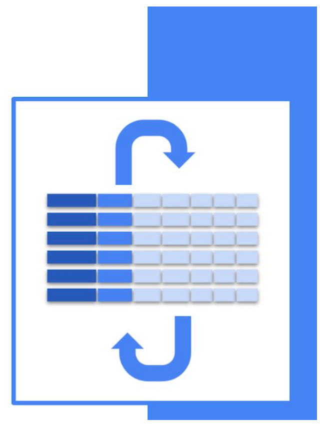
Doesn't consume your query resources

maintenance-free

Requires no setup or maintenance

autonomous

Automatically happens in the background



Google Cloud

The great news is that BigQuery now periodically does auto-reclustering 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 it.

[https://cloud.google.com/blog/products/data-analytics/whats-happening-bigquery-adding-speed-and-flexibility-10x-streaming-quota-cloud-sql-federation-and-more
https://cloud.google.com/bigquery/docs/clustered-tables#automatic_re-clustering]

Proprietary + Confidential

Organize data through managed tables

Partitioning

Filtering storage before query execution begins to reduce costs.

Reduces a full table scan to the partitions specified.

A single column results in lower cardinality (e.g., thousands of partitions).

- Ingestion-based partitioning (Pseudocolumn)
- Time partitioning (User Date/Time column)
- Integer range partitioning

Clustering

Storage optimization within columnar segments to improve filtering and record colocation.

Clustering performance and cost savings can't be assessed before query begins.

Prioritized clustering of up to 4 columns, on more diverse types (but no nested columns).

Google Cloud

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.

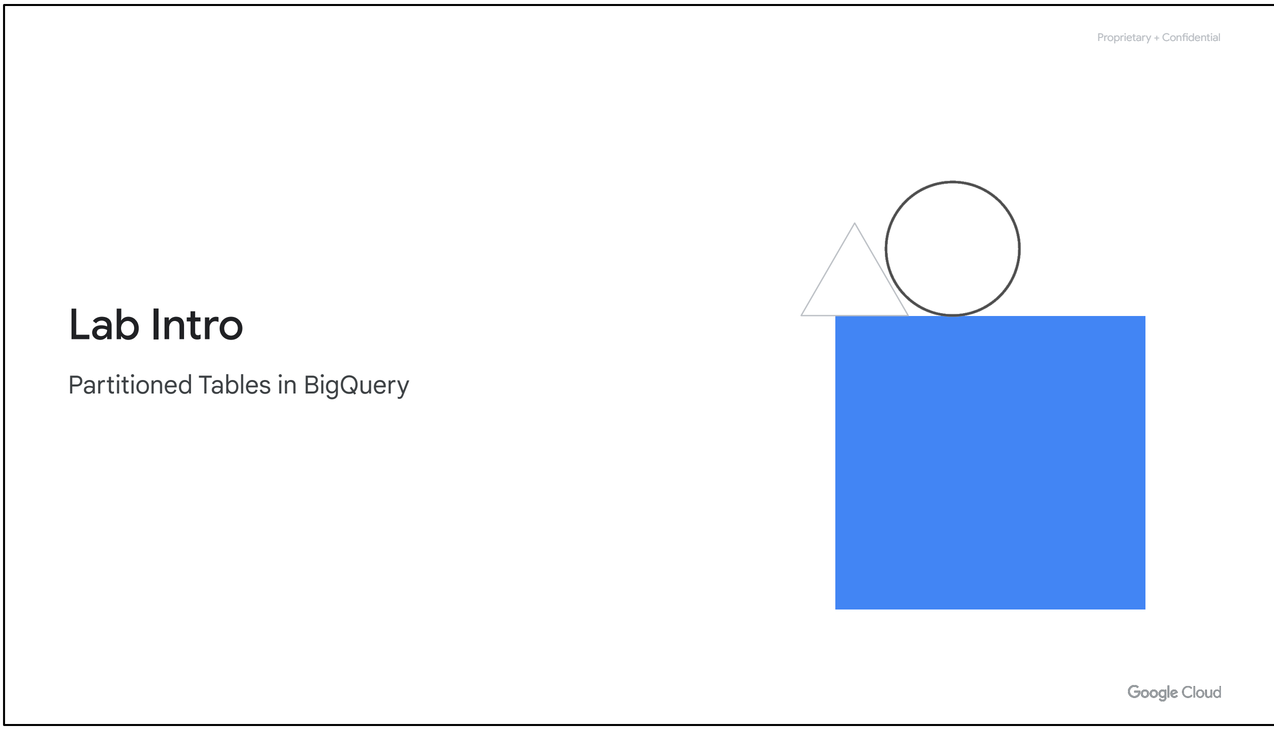
When to use clustering

- Your data is already partitioned on a `DATE`, `DATETIME`, `TIMESTAMP` or Integer Range.
- You commonly use filters or aggregation against particular columns in your queries.

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, datetime, 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_date column of type DATE and have all the values be NULL.



In this lab, you practice creating date-partitioned tables in BigQuery. Specifically, you query a partitioned dataset, and then you'll create dataset partitions to improve the query performance and reduce the overall cost.

Proprietary + Confidential

Review

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

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

There are several ways to ingest data into BigQuery.

BigQuery lets you specify a table's schema when you load data into a table.

Google Cloud

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 set up table partitioning and clustering to reduce the amount of data scanned and speed up your queries.