SQL in BigQuery

Google Cloud

Welcome to the "SQL in BigQuery" module. This module summarizes the key features and operations of the Google Standard SQL dialect used in BigQuery and best practices for optimizing query performance and controlling costs in BigQuery. Drawing upon your knowledge of Redshift, this module also provides a high-level overview of the similarities and differences in the SQL dialects and features between Redshift and BigQuery to help you start running and optimizing queries in BigQuery.

Let's jump in!

SQL translation references and other migration guides

References

SQL in BigQuery

Upon completion of this module, you will be able to:

1. List the supported statements in Google Standard SQL used by BigQuery.
2. Explain how SQL in Redshift differs from BigQuery.
3. Identify best practices to optimize query performance and control costs in BigQuery.
4. Run queries in BigQuery using best practices.

Lesson introduction

In this lesson, you will examine the similarities and differences in SQL syntax, features, and operations between Redshift and BigQuery.

Let's get started!

Lesson 3 of 21

Similarities and differences in SQL

Let's start by exploring a few similarities between Redshift and BigQuery.

Click the flash cards to learn more.

Scheduled queries

Both Redshift and BigQuery can schedule queries using a query editor.

Support for transactions

Redshift and BigQuery both support transactional workloads.

Support for Machine Learning (ML) and Geospatial data

Both Redshift and BigQuery support ML and Geospatial workflows using SQL commands.

Key differences

Let's examine some key differences for developers.

Click the + icon to expand and learn more.

SQL language –

- Redshift uses Amazon Redshift SQL.
- BigQuery uses Google Standard SQL.

Data aggregation commands

- Redshift does not have a summary option for data aggregation commands. Also, Redshift data aggregation commands do not accept window functions as arguments.
- BigQuery supports data sketches, which are compact summaries of data aggregations (based on approximate values); BigQuery also supports window functions such as moving averages, rankings, and cumulative sums.

Scheduling queries using APIs

- To schedule queries using the API in Redshift, you need to use both the Redshift Data API and EventBridge service.
- BigQuery has multiple options for scheduling queries, including using the BigQuery API.

BigQuery and Redshift

Here's a summary of topics related to SQL in BigQuery and Redshift.

Topic	BigQuery	Redshift
SQL	Google Standard SQL dialect (ANSI-compliant; options for DQL, DDL, DML, DCL, TCL, procedural statements, UDFs, EXPORT/LOAD DATA) Parameterized queries BigQuery ML using SQL Geography functions	No documented alignment with ANSI SQL, though it is aligned to PostgreSQL which is aligned with ANSI SQL SQL Reference Redshift ML Geospatial queries
Support for transactions	Multi-statement transactions	Managing transactions Dynamic SQL transaction conditions

Topic	BigQuery	Redshift
Support for data aggregations	Aggregate functions Approximate aggregate functions Data sketches for data aggregation HLL++ sketch functions	Data aggregation functions
Support for scheduled queries	Scheduling recurring queries	Scheduled queries
Mechanisms for debugging queries	Debugging functions Debugging statements Capture SQL activities in BigQuery sessions Error messages Troubleshoot quota and limit errors	Troubleshooting queries UDF logging Stored procedure error handling Internal query errors (STL_ERROR) Internal load errors (STL_LOAD_ERRORS, STL_LOADERROR_DETAIL)

Topic	BigQuery	Redshift
Query optimization strategies	Query results, including interactive and batch queries, are cached in temporary tables for approximately 24 hours with some exceptions. Inspect Query Plan and Timeline Use partition filtering Materialized Views Optimize with BI Engine Search with an index Best practices for optimizing query performance Best practices for controlling costs	System views for Query details (STL_QUERY and STL_QUERY_METRICS) Analyzing and Improving Queries Query tuning/ optimization Performance improvement tips
Resources for SQL translation	SQL translation guides: • Amazon Redshift • Oracle • Snowflake • Teradata • IBM Netezza	Redshift schema translation Migrate BigQuery to Redshift

Lesson introduction

In this lesson, you will learn about the different statements and features of Google Standard SQL used by BigQuery, including data manipulation language (DML) and data definition language (DDL) statements, user-defined functions (UDFs), and stored procedures.

Let's get started!

Overview of Google Standard SQL

Google Standard SQL is an ANSI compliant Structured Query Language (SQL) which includes the following types of supported statements.

Click the + icon to expand and learn more.

Query statements / Data Query Language (DQL) statements

These are the primary methods to analyze data in BigQuery. They scan one or more tables or expressions and return the computed result rows.

Procedural language statements

These are procedural extensions to Google Standard SQL that allow you to execute multiple SQL statements in one request. Procedural statements can use variables and control-flow statements, and can have side effects.

Data Definition Language (DDL) statements

These let you create and modify database objects such as tables, views, functions, and row-level access policies.

Data Manipulation Language (DML) statements

These enable you to update, insert, and delete data from your BigQuery tables.

Data Control Language (DCL) statements

These let you control BigQuery system resources such as access and capacity.

Transaction Control Language (TCL) statements

These allow you to manage transactions for data modifications.

Other statements

These provide additional functionality, such as loading and exporting data. For example, the EXPORT DATA statement can be used to export the results of a query to Cloud Storage, while the LOAD DATA statement can be used to load one or more files into a table.

Lesson 7 of 21

DDL and DML statements

Data definition language (DDL) statements

In BigQuery, DDL statements let you create and modify BigQuery resources using Google Standard SQL query syntax. You can use DDL commands to create, alter, and delete resources, such as tables, table clones, table snapshots, views, user-defined functions (UDFs), and row-level access policies.

Running DDL statements

You can run DDL statements by using the Google Cloud console, or the bq command-line tool, by calling the jobs.query REST API, or programmatically using the BigQuery API client libraries.

To create a job that runs a DDL statement, you must have the *bigquery.jobs.create* permission for the project where you are running the job. The predefined IAM roles **bigquery.user**, **bigquery.jobUser**, and **bigquery.admin** include the required *bigquery.jobs.create* permission.

Data manipulation language (DML) statements

In BigQuery, DML enables you to update, insert, and delete data from your BigQuery tables.

Any transaction that modifies or adds rows to a BigQuery table is ACID-compliant. BigQuery uses snapshot isolation to handle multiple concurrent operations on a table. DML operations can be submitted to BigQuery by sending a query job containing the DML statement.

DML in BigQuery supports inserting, updating, or deleting an arbitrarily large number of rows in a table in a single job. This means you can apply changes to data in a table more frequently and keep your data warehouse up to date with the changes in data sources.

Running DML statements

You can execute DML statements just as you would a SELECT statement, with the following conditions:

- You must use Google Standard SQL.
- You cannot specify a destination table for the query.

Best practices for running DML statements

For best performance, Google recommends the following patterns:

1. Avoid submitting large numbers of individual row updates or insertions. Instead, group DML operations together when possible.
2. If updates or deletions generally happen on older data, or within a particular range of dates, consider partitioning your tables. Partitioning ensures that the changes are limited to specific partitions within the table.

3. Avoid partitioning tables if the amount of data in each partition is small and each update modifies a large fraction of the partitions.

4. If you often update rows where one or more columns fall within a narrow range of values, consider using clustered tables. Clustering ensures that changes are limited to specific sets of blocks, reducing the amount of data that needs to be read and written.

5. If you need online transactional processing (OLTP) functionality, consider using Cloud SQL federated queries, which enable BigQuery to query data that resides in Cloud SQL.

Procedures and User Defined Functions (UDFs)

The Google Standard SQL procedural language lets you execute multiple statements in one query as a multi-statement query. You can use a multi-statement query to:

- Run multiple statements in a sequence, with shared state.
- Automate management tasks such as creating or dropping tables.
- Implement complex logic using programming constructs such as IF and WHILE.
- Work with stored procedures.

Stored procedures

A stored procedure is a collection of statements that can be called from other queries or other stored procedures. A procedure can take input arguments and return values as output.

In BigQuery, you name and store a procedure in a BigQuery dataset. A stored procedure can access or modify data across multiple datasets by multiple users. It can also contain a multi-statement query. Stored procedures also support procedural language statements, which let you do things like define variables and implement control flow.

To create a stored procedure in BigQuery, use the CREATE PROCEDURE statement.

The following example shows a procedure that contains a multi-statement query. The multi-statement query sets a variable, runs an INSERT statement, and displays the result as a formatted text string.

```
CREATE OR REPLACE PROCEDURE mydataset.create_customer()
BEGIN
 DECLARE id STRING;
 SET id = GENERATE_UUID();
 INSERT INTO mydataset.customers (customer_id)
   VALUES(id);
 SELECT FORMAT(“Created customer %s”, id);
END
```

The creation of the stored procedure begins with the statement CREATE OR REPLACE PROCEDURE which identifies the BigQuery dataset that will contain the new procedure and the name of the new procedure. Then, the steps of the procedure are specified between the BEGIN and END statements.

Note that some stored procedures are built into BigQuery and do not need to be created. These are called system procedures, and you can learn more about the options and how to use them in the System procedures reference.

User Defined Functions (UDFs)

In BigQuery, a UDF lets you create a function by using a SQL expression or JavaScript code. A UDF accepts columns of input, performs actions on the input, and returns the result of those actions as a value. UDFs allow you to extend the built-in SQL functions.

You can define UDFs as either persistent or temporary. You can reuse persistent UDFs across multiple queries, while temporary UDFs only exist in the scope of a single query.

Managing UDFs

Click the flash cards to learn more.

CREATE FUNCTION

To create a UDF, use the CREATE FUNCTION statement.

DROP FUNCTION

To delete a persistent UDF, use the DROP FUNCTION statement. Temporary UDFs expire as soon as the query finishes. The DROP FUNCTION

The following example creates a persistent SQL UDF that takes two FLOAT64 values as input and returns the product of these two values as a FLOAT64 value.

```
CREATE FUNCTION mydataset.multiplyInputs(x FLOAT64, y FLOAT64)
RETURNS FLOAT64
AS (x*y);
```

The creation of the UDF begins with the statement CREATE FUNCTION, which identifies the BigQuery dataset that will contain the new function, the function name, and the function input variables. Then, the steps of the UDF (including any outputs) are defined before the ending semi-colon.

Authorized functions

You can authorize a UDF by leveraging authorized functions to share query results with particular users or groups without giving those users or groups access to the underlying tables.

Additional helpful SQL features

Google Standard SQL for BigQuery supports many other functions, operators, and conditional expressions. Some key options are described below to highlight the versatility of BigQuery.

Click the + icon to expand and learn more.

Run parameterized queries

BigQuery supports query parameters to help prevent SQL injection when queries are constructed using user input.

You can use the @ character followed by an identifier, such as @param_name, to specify a named parameter, or you can use the placeholder value of "?" to specify a positional parameter. Note that a query can use positional or named parameters but not both.

Analyze geospatial data using geography functions

Google Standard SQL for BigQuery supports geography functions that operate on or generate Google Standard SQL GEOGRAPHY values. The signature of most geography functions starts with ST_.

Google Standard SQL for BigQuery supports many functions that can be used to analyze geographical data, determine spatial relationships between geographical features, and construct or manipulate GEOGRAPHYs.

Create machine learning models using BigQuery ML

BigQuery ML enables users to create and execute machine learning models in BigQuery by using SQL queries. This feature can help democratize machine learning in your organization by enabling SQL practitioners to build models using their existing tools and to increase development speed by eliminating the need for data movement.

BigQuery Machine Learning in a minute

References

- Learn more about query syntax for Google Standard SQL from the documentation titled "Query syntax."
- Learn about data control language (DCL) statements from the documentation titled "Data control language (DCL) statements in Google Standard SQL."
- Learn more about working with transactional control language (TCL) statements in BigQuery from the documentation titled "Transactions."
- Learn more about SQL stored procedures from the documentation titled "Work with SQL stored procedures."
- Learn more about defining UDFs using SQL and JavaScript from the documentation titled "CREATE FUNCTION statement."
- Explore open source community-contributed UDFs from the GitHub repository titled "bigquery-utils."

Lesson introduction

In this lesson, you will explore strategies and tools to help you optimize query performance in BigQuery including materialized views, BI Engine, and search indexes.

Let's get started!

Lesson 12 of 21

General strategies to optimize query performance

Evaluate query performance

Evaluating query performance in BigQuery involves several factors.

Click the flash cards to learn more.

Input data and data sources (I/O)

How many bytes does your query read?

Communication

How many bytes does your query pass to

between nodes (shuffling)	advance to the next stage? How many bytes does your query pass to each slot?
Computation	How much CPU work does your query require?
Outputs (materialization)	How many bytes does your query write?

Query anti-patterns

Do your queries avoid SQL anti-patterns such as self-joins and data skewing?

Many of these questions are answered in a query plan that BigQuery generates each time you run a query in BigQuery. The query plan shows execution statistics such as bytes read and slot time consumed. It also shows the different stages of execution, which can help you diagnose and improve query performance.

Best practices for enhancing query performance

Sometimes queries run more slowly than you would like. In general, queries that do less work perform better. They run faster and consume fewer resources, which can result in lower costs and fewer failures. Here are some key general strategies for reducing work by queries.

Click the + icon to expand and learn more.

Avoid repeatedly transforming data through SQL queries

Best practice: If you are using SQL to perform ETL operations, avoid situations where you are repeatedly transforming the same data.

For example, if you are using SQL to trim strings or extract data by using regular expressions, it is more performant to materialize the transformed results in a destination table. Functions like regular expressions require additional computation. Querying the destination table without the added transformation overhead is much more efficient.

Optimize your join pattern

Best practice: For queries that join data from multiple tables, optimize your join patterns. Start with the largest table.

When you create a query by using a JOIN, consider the order in which you are merging the data. The Google Standard SQL query optimizer can determine which table should be on which side of the join, but it is still recommended to order your joined tables appropriately. As a best practice, place the table with the largest number of rows first, followed by the table with the fewest rows, and then place the remaining tables by decreasing size.

Use INT64 data types in joins

Best practice: If your use case supports it, use INT64 data types in joins instead of STRING data types.

BigQuery does not index primary keys like traditional databases, so the wider the join column is, the longer the comparison takes. Therefore, using INT64 data types in joins is cheaper and more efficient than using STRING data types.

Prune partitioned queries

Best practice: When querying a partitioned table, to filter with partitions on partitioned tables, use the following columns:

- For ingestion-time partitioned tables, use the pseudo-column _PARTITIONTIME

- For partitioned tables such as the time-unit column-based and integer-range, use the partitioning column.

For time-unit partitioned tables, filtering the data with _PARTITIONTIME or partitioning column lets you specify a date or range of dates. The query will only process data in the partitions that are indicated by the date range.

Avoid duplicate usage of same CTEs

Best practice: Use procedural language, variables, temporary tables, and automatically expiring tables to persist calculations and use them later in the query.

When your query contains Common Table Expressions (CTEs) that are used in multiple places in the query, they are evaluated each time they are referenced. This may increase internal query complexity and resource consumption. You can store the result of a CTE in a scalar variable or a temporary table depending on the data that the CTE returns. You are not charged for storage of temporary tables.

Split complex queries into multiple smaller ones

Best practice: Leverage multi-statement query capabilities and stored procedures to perform the computations that were designed as one complex query as multiple smaller and simpler queries instead.

Complex queries, REGEX functions, and layered subqueries or joins can be slow and resource intensive to run. Trying to fit all computations in one huge SELECT statement, for example to make it a view, is sometimes an antipattern, and it can result in a slow, resource-intensive query.

Splitting up a complex query allows for materializing intermediate results in variables or temporary tables. You can then use these intermediate results in other parts of the query. It is increasingly useful when these results are needed in more than one place of the query. You are not charged for storage of temporary tables.

Lesson 13 of 21

Optimize queries with Materialized Views

In BigQuery, materialized views are precomputed views that periodically cache the results of a query for increased performance and efficiency. Queries that use materialized views are generally faster and consume fewer resources than queries only from the base tables.

Materialized views can improve query performance if you frequently require the following:

- Pre-aggregate data: Create aggregations of streaming data.
- Pre-filter data: Run queries that only read a particular subset of the table.
- Pre-join data: Create query joins especially between large and small tables.
- Recluster data: Run queries that would benefit from a clustering scheme that differs from the base tables. Materialized views can be clustered on different columns from the original base table.

The following are key characteristics of materialized views.

Zero maintenance

Materialized views are recomputed in the background when the base tables change. Any incremental data changes from the base tables are automatically added to the materialized views, with no user action required.

Fresh data

Materialized views return fresh data. If changes to base tables might invalidate the materialized view, then data is read directly from the base tables. If the changes to the base tables do not invalidate the materialized view, then rest of the data is read from the materialized view and only the changes are read from the base tables.

Smart tuning

If any part of a query against the source table can be resolved by querying the materialized view, then BigQuery reroutes the query to use the materialized view for better performance and efficiency.

Creating Materialized Views

You can create BigQuery materialized views through the Google Cloud console, the bq command-line tool, or the BigQuery API.

To create materialized views, you need the *bigquery.tables.create* IAM permission.

Each of the following predefined IAM roles includes the permissions that you need in order to create a materialized view:

- **bigquery.dataEditor**
- **bigquery.dataOwner**
- **bigquery.admin**

Lesson 14 of 21

Optimize queries with BI Engine

BigQuery BI Engine is a fast, in-memory analysis service that accelerates many SQL queries in BigQuery by intelligently caching the data you use most frequently. BI Engine can accelerate SQL queries from any source, including those written by data visualization tools, and can manage cached tables for on-going optimization. This lets you improve query performance without manual tuning or data tiering.

You can use clustering and partitioning to further optimize the performance of large tables with BI Engine. For example, if your dashboard only displays the last quarter's data, then consider partitioning your tables by time so only the latest partitions are loaded into memory.

You can also combine the benefits of materialized views and BI Engine. This works particularly well when the materialized views are used to join and flatten data to optimize their structure for BI Engine.

What are the advantages of BI Engine?

Click the + icon to expand and learn more.

BigQuery API –

BI Engine directly integrates with the BigQuery API. Any BI solution or custom application that works with the BigQuery API through standard mechanisms such as REST or JDBC and ODBC drivers can use BI Engine without modification.

Vectorized runtime

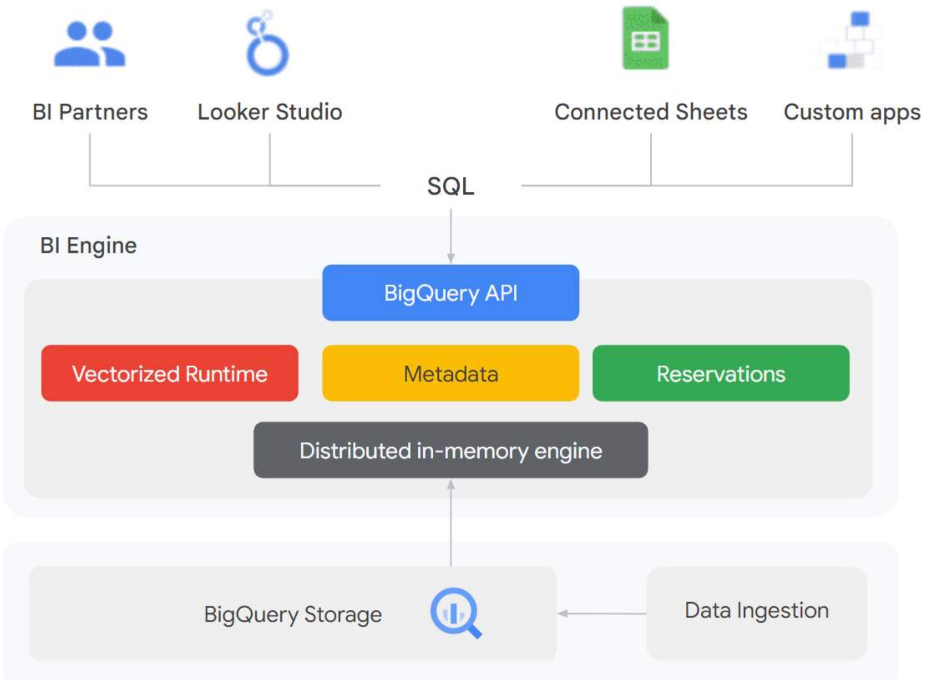
With the BI Engine SQL interface, BI Engine introduces a more modern technique called vectorized processing. Using vectorized processing in an execution engine makes more efficient use of modern CPU architecture, by operating on batches of data at a time. BI Engine also uses advanced data encodings, specifically, dictionary and run-length encoding, to further compress the data that's stored in the in-memory layer.

Seamless integration

BI Engine works with BigQuery features and metadata, including authorized views, column and row level security, and data masking.

Reservations

BI Engine reservations manage memory allocation at the project location level. BI Engine caches specific columns or partitions that are queried, prioritizing those in tables marked as preferred.



The BI Engine SQL interface expands BI Engine to integrate with other business intelligence (BI) tools such as Looker Studio, Connected Sheets, and custom applications to accelerate exploration and analysis of BigQuery data.

Best practices for using BI Engine

Consider the following when deciding how to configure BI Engine.

Ensure acceleration for specific queries

You can ensure a particular set of queries always gets accelerated by creating a separate project with a BI Engine reservation. To do so, you should ensure that the BI Engine reservation in that project is large enough to match the size of all tables used in those queries and designate those tables as preferred tables for BI Engine. Only those queries that need to be accelerated should be run in that project.

Minimize your joins

BI Engine works best with pre-joined or pre-aggregated, and with data in a small number of joins. This is particularly true when one side of the join is large and the others are much smaller such as when you query a large fact table joined with a small dimension table. You can combine BI Engine with materialized views that perform joins to produce a single large, flat table. In this way, the same joins don't have to be performed on every query.

Understand the impact of BI Engine

You can better understand how your workloads benefit from BI Engine by reviewing the usage statistics in Cloud Monitoring or querying the INFORMATION_SCHEMA in BigQuery. Be sure to disable the "Use cached results" option in BigQuery to get the most accurate comparison.

Optimize queries with a search index

The SEARCH function

The SEARCH function provides tokenized search on data. SEARCH is designed to be used with an index to optimize lookups. The SEARCH function always returns correct results from all ingested data, even if some of the data isn't indexed yet. However, its performance is greatly improved with the creation of a search index on the table.

Using a search index

You can create a search index on a table using the following example command:

```
CREATE  SEARCH  INDEX  my_index  ON my_dataset.my_table(ALL COLUMNS) ;
```

After you create this search index, you can use it to quickly search across all columns of the table for a value and return the rows that contain the value. You can also search a subset of columns or exclude columns from a search.

This example query uses the default text analyzer (other options available) to search for the value `bar` and return the rows that contain this value, regardless of capitalization:

```
SELECT *  FROM  my_dataset.my_table  WHERE  SEARCH(my_table,   ' bar') ;
```

A simple modification to include backticks allows for an exact search match to return only rows with lowercase `bar`.

```
SELECT * FROM my_dataset.my_table WHERE SEARCH(my_table, '`bar`');
```

Best practices for searching with an index

Click the + icon to expand and learn more.

Search selectively

Searching works best when your search has few results. Make your searches as specific as possible.

Scope your search

If you know which columns of a table should contain your search terms, then restrict your search to only those columns. This improves performance and reduces the number of bytes that need to be scanned.

Use backticks

Enclosing your search query in backticks forces an exact match. This is helpful if your search is case-sensitive or contains characters that shouldn't be interpreted as delimiters. For example, to search for the IP address 192.0.2.1, use `192.0.2.1`. Without the backticks, the search returns any row that contains the individual tokens 192, 0, 2, and 1, in any order.

Optimize your queries

Queries that use the SEARCH function on a very large partitioned table are optimized when you use an ORDER BY clause on the partitioned field and a LIMIT clause. The optimization is applied whether or not the table is indexed. This works well if you're searching for a common term.

Control query costs in BigQuery

BigQuery has two pricing models for running queries:

- On-demand pricing: You pay for the number of bytes processed by each query.
- Flat-rate pricing: You pay for dedicated query processing capacity, measured in slots.

If you are using on-demand pricing, then you can directly reduce costs by reducing the number of bytes that a query processes. With flat-rate pricing, your cost is fixed based on the number of slots that you purchase and the slot commitment plan that you select. However, optimizing your queries can help to reduce slot usage.

Google recommends the following best practices to control query costs in BigQuery.

Click the + icon to expand and learn more.

Avoid SELECT *

Query only the columns that you need.

Using SELECT * is the most expensive way to query data. When you use SELECT *, BigQuery does a full scan of every column in the table.

Instead, query only the columns you need. For example, use SELECT * EXCEPT to exclude one or more columns from the results.

If you do require queries against every column in a table, but only against a subset of data, consider:

- Materializing results in a destination table and querying that table instead.
- Partitioning your tables and querying the relevant partition. For example, use WHERE _PARTITIONDATE="2017-01-01" to query only the January 1, 2017 partition.

Don’t run queries to preview data —

Don't run queries to explore or preview table data. Use table preview options to view data for free and without affecting quotas.

BigQuery supports the following data preview options:

1. In the Google Cloud console, on the table details page, click the Preview tab to sample the data.
2. In the bq command-line tool, use the bq head command and specify the number of rows to preview.
3. In the API, use tabledata.list to retrieve table data from a specified set of rows.

Estimate query costs before running —

Before running queries, preview them to estimate costs.

View the query validator in the Google Cloud console to estimate the number of bytes read. Or, perform a dry run to estimate the number of bytes read by using the:

- --dry_run flag in the bq command-line tool
- dryRun parameter when submitting a query job using the API

Input the estimated number of bytes read into the Google Cloud Pricing Calculator to estimate query costs.

Set max bytes limits

Use the maximum bytes billed setting to limit query costs.

You can limit the number of bytes billed for a query using the maximum bytes billed setting. When you set maximum bytes billed, the number of bytes that the query will read is estimated before the query execution. If the number of estimated bytes is beyond the limit, then the query fails without incurring a charge.

If a query fails because of the maximum bytes billed setting, an error like the following is returned: Error: Query exceeded limit for bytes billed: 1000000. 10485760 or higher required.

Use clustered and partitioned tables

Use clustering and partitioning to reduce the amount of data scanned.

Clustering and partitioning can help to reduce the amount of data processed by queries. To limit the number of partitions scanned when querying clustered or partitioned tables, use a predicate filter.

If you run a query against a clustered table, and the query includes a filter on the clustered columns, then BigQuery uses the filter expression and the block metadata to prune the blocks scanned by the query. For more information, see Querying clustered tables.

When querying partitioned tables, filters on the partitioning column are used to prune the partitions and therefore can reduce the query cost. For more information, see Querying partitioned tables.

Use LIMIT with caveats

For non-clustered tables, do not use a LIMIT clause as a method of cost control.

For non-clustered tables, applying a LIMIT clause to a query does not affect the amount of data that is read. You are billed for reading all bytes in the entire table as indicated by the query, even though the query returns only a subset. With a clustered table, a LIMIT clause can reduce the number of bytes scanned, because scanning stops when enough blocks are scanned to get the result. You are billed for only the bytes that are scanned.

Monitor billing data

Create a dashboard to view your billing data so you can make adjustments to your BigQuery usage. Also consider streaming your audit logs to BigQuery so you can analyze usage patterns.

You can export your billing data to BigQuery and visualize it in a tool such as Looker Studio. You can also stream your audit logs to BigQuery and analyze the logs for usage patterns such as query costs by user.

Use date-based partitions

Partition your tables by date.

Partitioning your tables lets you query relevant subsets of data which improves performance and reduces costs.

For example, when you query partitioned tables, use the _PARTITIONTIME pseudo column to filter for a date or a range of dates. The query processes data only in the partitions that are specified by the date or range.

Materialize complex queries in stages

Materialize your query results in stages.

Rather than creating a large, multi-stage query, break your query into stages where each stage materializes the query results by writing them to a destination table. Querying the smaller destination table reduces the amount of data that is read and lowers costs. The cost of storing the materialized results is much less than the cost of processing large amounts of data.

Apply table expirations

If you are writing large query results to a destination table, use the default table expiration time to remove the data when it's no longer needed.

Keeping large result sets in BigQuery storage has a cost. If you don't need permanent access to the results, use the default table expiration to automatically delete the data for you.

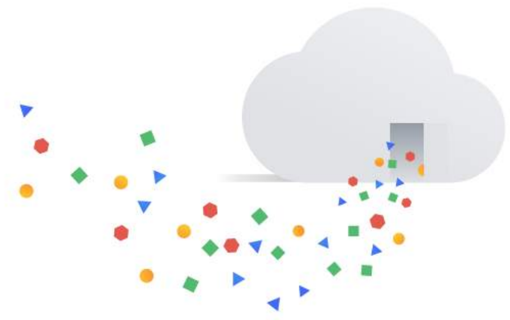
References

- Learn more about best practices for optimizing compute and query performance from the documentation titled:
  - Reduce data processed in queries
  - Optimize query computation
  - Manage query outputs
  - Use temporary tables in a multi-statement query
  - Introduction to materialized views
- Learn more about using and monitoring BI Engine from the documentation titled:
  - BI Engine preferred tables
  - INFORMATION_SCHEMA.BI_CAPACITIES view
  - Monitor BI Engine
- Learn more about querying clustered and partitioned tables from the documentation titled:
  - Querying clustered tables
  - Query partitioned tables

Lesson 18 of 21

Lesson introduction

In this lesson, you will learn about services and resources for SQL translation and data migration to BigQuery.



Let's get started!

BigQuery Migration Service

The BigQuery Migration Service is a comprehensive resource for migrating your data warehouse to BigQuery. It includes many tools at no cost to help you with each phase of migration, including assessment and planning, SQL translation for more than 10 dialects, data transfer, and data validation.

The assessment and planning phase

In the assessment and planning phase, you can use the BigQuery migration assessment feature to understand your existing data warehouse. Then, you can use the batch SQL translator and the interactive SQL translator to prepare your SQL queries and scripts to work in BigQuery. The batch and interactive SQL translators support translation from a wide range of SQL dialects.

Moving your data

When you're ready to move your data, you can use the BigQuery Data Transfer Service to automate and manage the migration from your data warehouse to BigQuery. After you migrate your data, you can use the Data Validation Tool to validate that the migration succeeded.

There is no charge to use the BigQuery Migration API. However, storage used for input and output files does incur storage pricing fees for BigQuery. BigQuery quotas and limits also apply to the number of jobs and the size of files.

Lesson 20 of 21

SQL translation references and other migration guides

SQL translation references are available for the following:

- Amazon Redshift
- IBM Netezza
- Oracle
- Snowflake
- Teradata

There are also comprehensive migration guides available for the following:

- Amazon Redshift
- Apache Hive
- IBM Netezza

- Oracle
- Snowflake
- Teradata

References

- Learn more about how to prepare your SQL queries and scripts to work in BigQuery from the documentation titled "Migrate code with the batch SQL translator" and "Translate queries with the interactive SQL translator".