BigQuery Schema Design and Optimization

Google Cloud

Welcome to the "BigQuery Schema Design and Optimization" module. This module summarizes common patterns and best practices for designing and optimizing table schemas in BigQuery, including the use of nested and repeated fields, partitioning, and clustering. Drawing upon your knowledge of Redshift, this module also provides a high-level overview of the similarities and differences in schema usage and design between Redshift and BigQuery to help you start structuring and optimizing your data in BigQuery.

Let's jump in!

BigQuery Schema Design and Optimization

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

1. Describe common patterns and best practices for designing and optimizing table schemas in BigQuery.
2. Explain how schemas in Redshift differ from BigQuery.
3. Design schemas in BigQuery using best practices.

Lesson 2 of 14

Lesson introduction

In this lesson, you will examine the similarities and differences in how schemas are used to structure, index, and query your data in Redshift and BigQuery.

Let's get started!

Similarities and differences in schema design

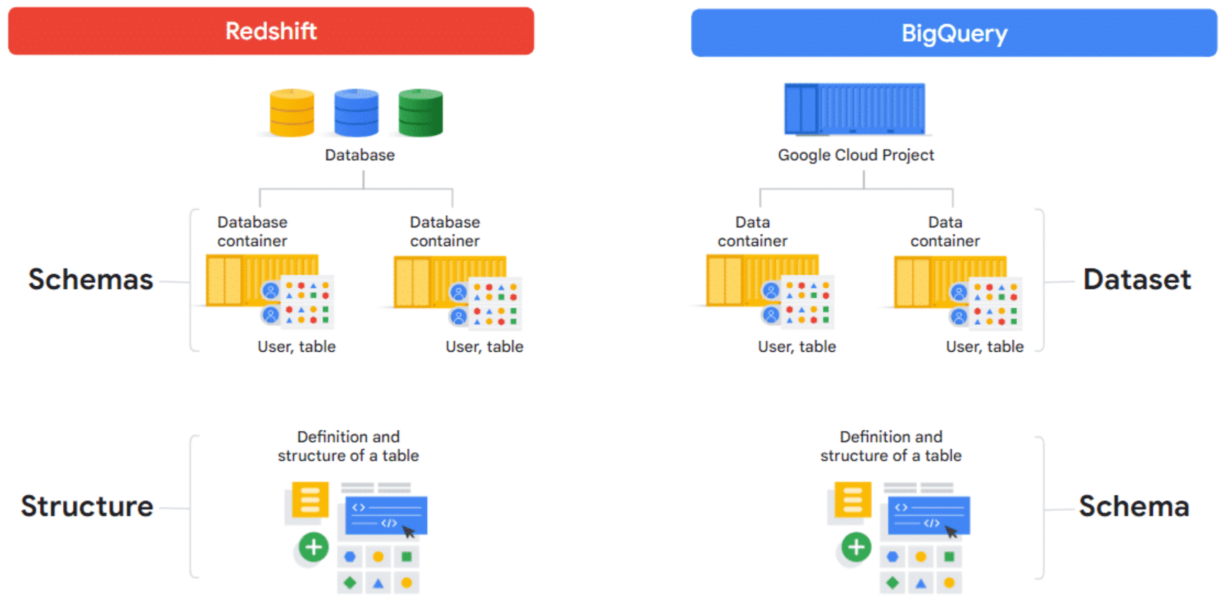
Let's start with some important similarities.

1. In both Redshift and BigQuery, you can change or alter schemas without impacting availability of the data.

2. Redshift and BigQuery data can both be configured with indexes. In Redshift, indexes can be created as primary or foreign keys; in BigQuery, indexes can be created for search optimization.

Tables and schemas relate differently in Redshift and BigQuery.

In Redshift, a schema refers to the database container used to organize and secure tables and other named objects. In BigQuery, a schema refers to the definition and structure of a table, including column names and types. The closest parallel of the Redshift schema is the BigQuery dataset, which contains BigQuery tables.



Let's explore some other key differences.

Click the + icon to expand and learn more.

Indexes

- In Redshift, indexing is applied as a constraint, not as a separate object, and is not enforced at the table level in the database.
- In BigQuery, you can create search indexes to improve search performance for specific column types including STRING, ARRAY<STRING>, STRUCT, and JSON. BigQuery does not use primary or foreign keys for indexing.

Schema creation

- In Redshift, schemas are created using SQL statements. When an object like a table is created, the schema that the object belongs to can be specified. The default schema is "public" with no optimization.
- In BigQuery, there are multiple ways to create or specify a table schema, including the bq command line tool, the Google Cloud Console, loading a JSON file, and through the BigQuery API. BigQuery also offers auto-detection of table schemas when loading new data or querying external data sources.

Views

- Redshift allows multiple schemas for each table and database. The schemas can be applied to a user's permissions, which controls what the user can see and interact with in a table or database.
- Similar functionality can be accomplished using views in BigQuery.

Lesson 4 of 14

BigQuery and Redshift

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

Topic	BigQuery	Redshift
Schema components	BigQuery schemas refer to the structure of the tables including column names and types. Required components for each table column: • Column name • Column data type Optional components for each table column: • Column description • Mode (e.g. nullable, required) • Default value	Redshift schemas refer to database containers used to organize and secure tables and other named objects. Redshift schemas Create schema Create external schema

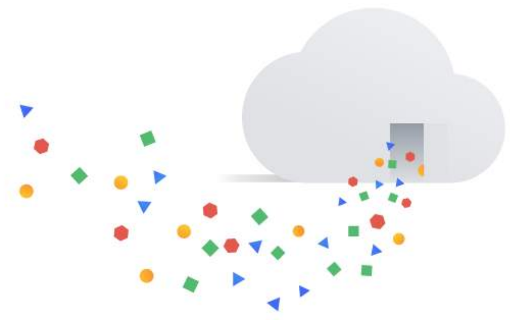
Topic	BigQuery	Redshift
Schema design for tables	Manual definition of schemas JSON schema file or using the API Auto-detect schemas Nested and repeated fields JSON for semi-structured data Geospatial data	Define table structure Semistructured data Geospatial data
Schema optimization for tables	Partitioning Clustering	Data distribution (partitioning analog) Sort keys (clustering analog)
Indexing	Search indexes	Primary and foreign keys using table constraints

Topic	BigQuery	Redshift
Resources for migrating data and schemas	BigQuery Migration Service Migration guides for schema and data: • Amazon Redshift • Oracle • Snowflake • Teradata	AWS schema conversion Schema conversion tool

Lesson 5 of 14

Lesson introduction

In this lesson, you will learn about schema design in BigQuery, including how BigQuery leverages nested and repeated fields in order to optimize schema design and query performance.



Let's get started!

BigQuery architecture and schema design

Why is BigQuery so efficient for read performance?

BigQuery handles both the concept of scaling out into multiple different tables and the idea of building indexes to handle massive amounts of rows and tables in a fundamentally different way. BigQuery introduces three innovations that are central to its architecture.

Click the flash cards to learn more.

Column-based storage

BigQuery data are stored as individual compressed columns inside of Colossus—Google's massive scalable storage system.

All BigQuery data is actually

Sharding for distributed storage

broken up and sharded into many chunks which are individually and distributedly stored, and then accessed by BigQuery when executing queries in a parallel fashion.

Nested and repeated fields

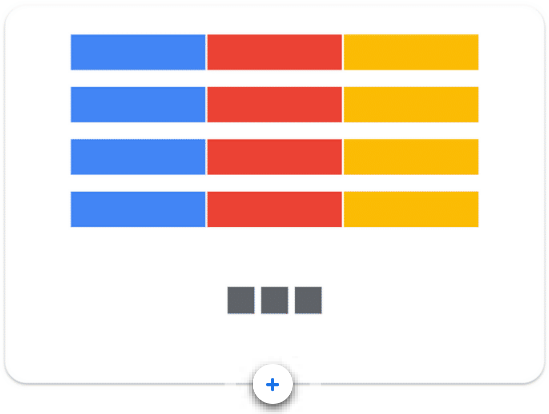
To maintain relationships while denormalizing your data, you can use nested and repeated fields instead of completely flattening your data.

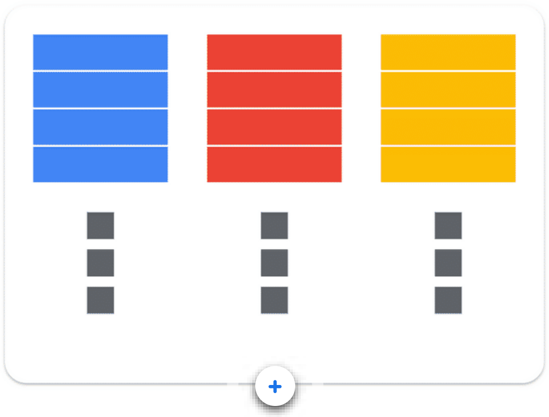
Column-oriented storage

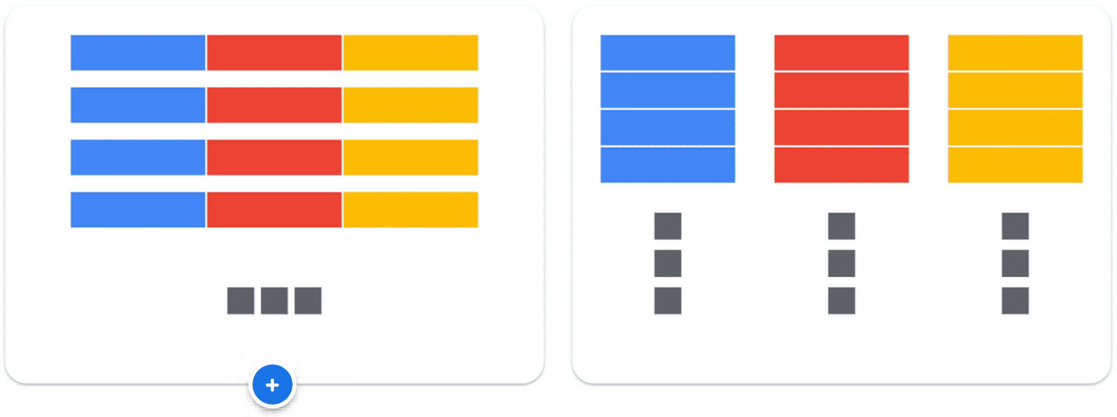
In traditional databases, you store individual records. When you query a record, you pull all of the bytes of the different columns that are present in the table. Even if you just want three out of the 200 columns, that record itself cannot be broken apart, so all 200 columns are pulled.

BigQuery uses column-oriented storage. When you query data using SQL, you are actually selecting only the individual columns specified in the query. Because the columns are stored individually, the query can return results much faster because it is only pulling the requested columns, not the entire table.

Click the + icon to expand and learn more.

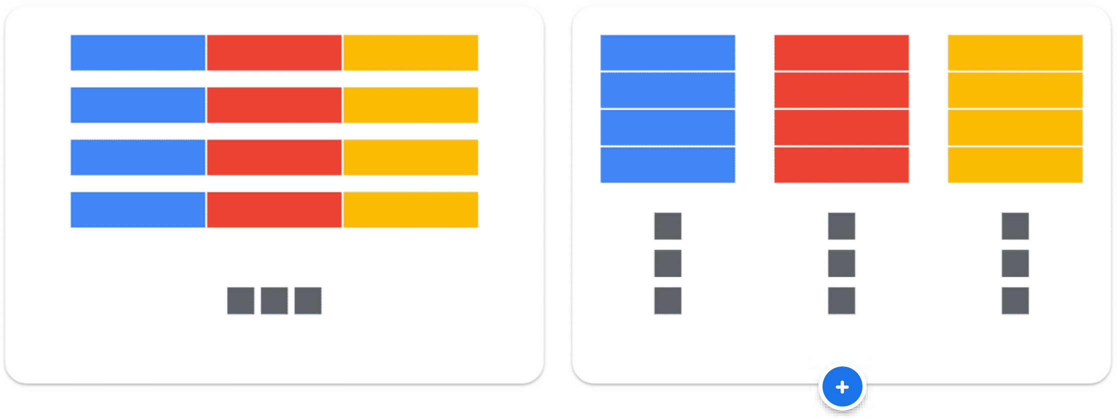






Record-oriented storage

Record-oriented, or row-oriented, storage can be efficient for looking up individual records, but less efficient at performing analytical functions across many records because the system has to read every column when accessing a record."



Column-oriented storage

Column-oriented storage is optimized for analytic workloads that aggregate data over a very large number of records. For example, to calculate the sum of a column over millions of rows, BigQuery can read only that column data without reading every column of every row.

Lesson 7 of 14

Defining schemas

Options for specifying schemas

BigQuery lets you specify a table's schema when you load data into a table, and when you create an empty table. Alternatively, you can use schema auto-detection for supported data formats when you load data into BigQuery and when you query an external data source.

When you load Avro, Parquet, ORC, Firestore export files, or Datastore export files, the schema is automatically retrieved from the self-describing source data.

You can specify a table's schema in the following ways:

1. Manually specify the schema using the Google Cloud console; inline using the bq command-line tool, or using the CREATE TABLE SQL statement.
2. Create a schema file in JSON format.
3. Call the jobs.insert method and configure the schema property in the load job configuration.
4. Call the tables.insert method and configure the schema in the table resource using the schema property.

Click the + icon to expand and learn more.

What happens when schema auto-detection is enabled?

When auto-detection is enabled, BigQuery infers the data type for each column. BigQuery scans up to the first 500 rows of data to use as a representative sample. BigQuery then examines each field and attempts to assign a data type to that field based on the values in the sample.

If you don't enable schema auto-detection for CSV, JSON, or Sheets data, then you must provide the schema manually when creating the table.

What about self-describing file formats?

You don't need to enable schema auto-detection for Avro, Parquet, ORC, Firestore export, or Datastore export files. These file formats are self-describing, so BigQuery automatically infers the table schema from the source data.

For Parquet, Avro, and Orc files, you can optionally provide an explicit schema to override the inferred schema.

You can see the detected schema for a table in the following ways:

- Use the Google Cloud console.
- Use the bq command-line tool's bq show command.

Required components for each column

When you specify a table schema, you must supply each column's name and data type. You can also supply a column's description, mode, and default value.

A column name must contain only letters (a-z, A-Z), numbers (0-9), or underscores (_), and it must start with a letter or underscore. The maximum column name length is 300 characters. A column name cannot use any of the following prefixes:

- _TABLE_
- _FILE_
- _PARTITION
- _ROW_TIMESTAMP
- __ROOT__
- _COLIDENTIFIER

Duplicate column names are not allowed even if the case differs. For example, a column named Column1 is considered identical to a column named column1.

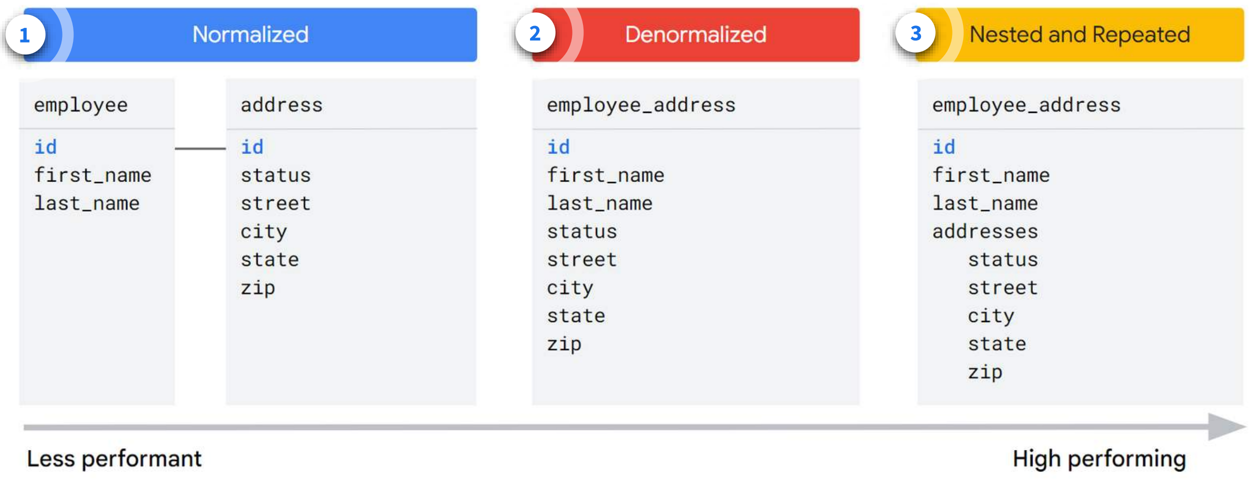
Nested and repeated fields

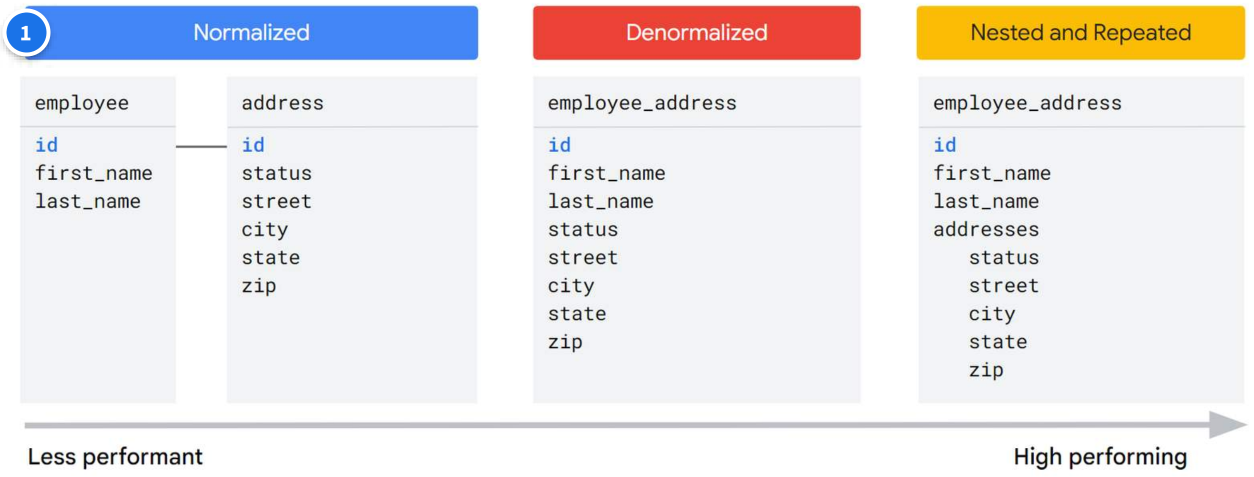
Many developers are accustomed to working with normalized data schemas that avoid duplicating any data in the tables.

Denormalization is a common strategy for increasing read performance for relational datasets that were previously normalized. So, rather than preserving a relational schema such as a star or snowflake schema and relying on joins, you flatten the data to allow some repeated data.

In BigQuery, you can use nested and repeated fields to denormalize data and maintain relationships without impacting performance. It's best to use this strategy when the relationships are hierarchical and frequently queried together, such as in parent-child relationships.

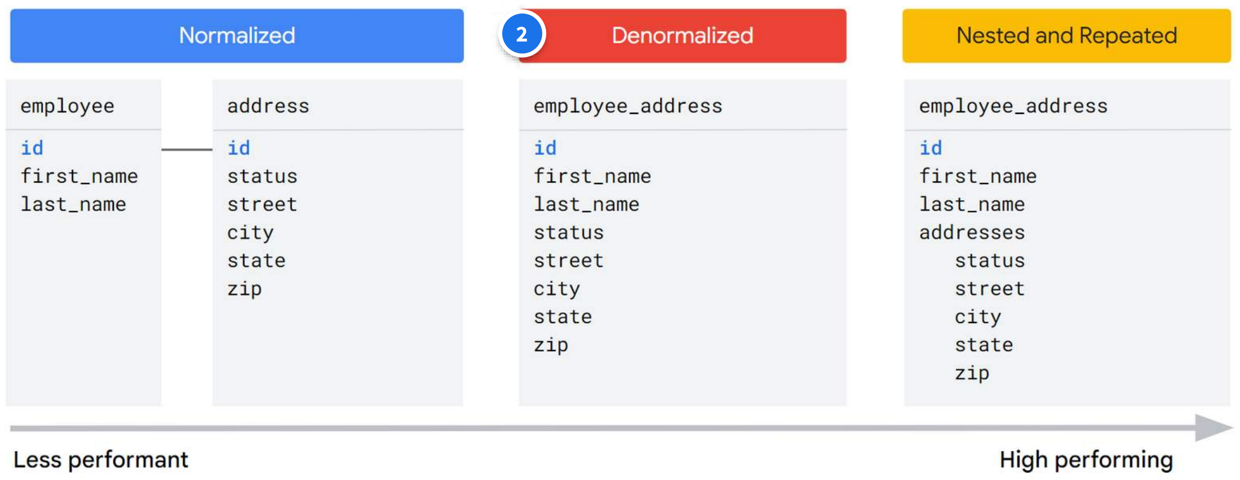
Click the numbers in sequence to learn more.





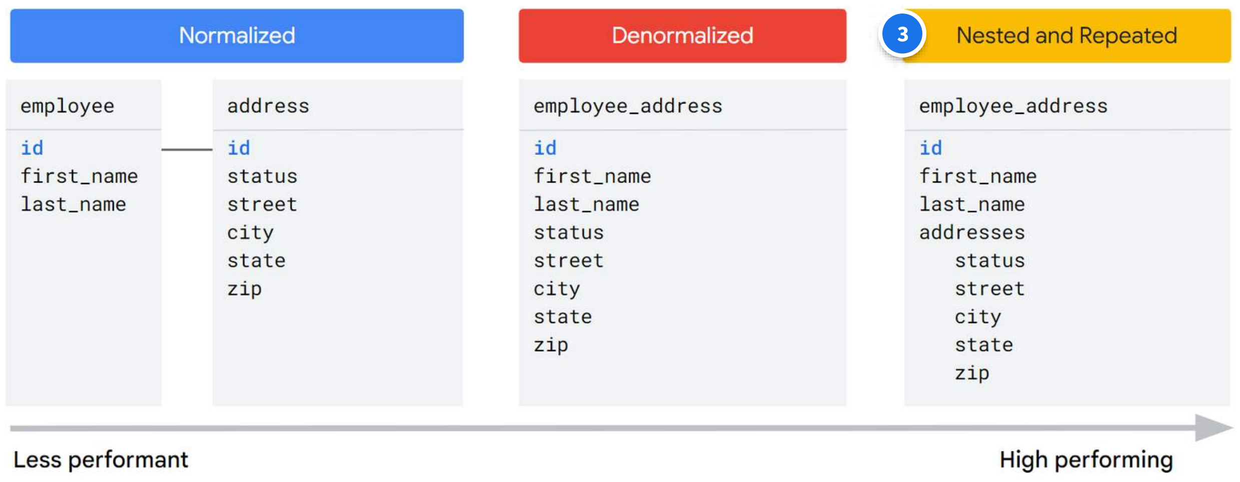
Normalized

When using the concept of normalization, you have data stored in two different tables such as employee and address. This two-table format illustrates the traditional model of relational databases: related information is stored in separate tables that can be linked with a common key such as id. In this case, you sacrifice performance due to the need for joins.



Denormalized

With denormalization, you can optimize data for reads, but as a result, you may have a lot of redundancy in the stored data, resulting in large tables. For example, one employee may require multiple rows of data, one for each address associated with that employee.



Nested and Repeated

With nested and repeated fields, you can store multiple data columns within a parent column and specify multiple values for those child columns. For example, you can store employee addresses in one table using a nested and repeated format in which the status and street information are nested under the parent column called addresses, and there can be many values for status or street, such as a home address or work address.

Using nested and repeated fields

While BigQuery performs best when data is denormalized, it doesn't require a completely flat denormalization; you can use nested and repeated fields to maintain relationships.

- Nesting data (STRUCT)
    - Nesting data lets you represent foreign entities inline.
    - Querying nested data uses "dot" syntax to reference leaf fields, which is similar to the syntax using a join.
    - Nested data is represented as a STRUCT type in Google Standard SQL.
    - To create a column with nested data, set the data type of the column to RECORD in the schema.
- Repeated data (ARRAY)

- Repeated data is represented as an ARRAY. You can use an ARRAY function in Google Standard SQL when you query the repeated data.
- To create a column with repeated data, set the mode of the column to REPEATED in the schema.

* Nested and repeated data (ARRAY of STRUCTs)

    - A RECORD column (nested column) can have REPEATED mode, which is represented as an array of STRUCT types. In addition, a field within a record (nested field) can be repeated, which is represented as a STRUCT that contains an ARRAY.
    - Creating a field of type RECORD with the mode set to REPEATED lets you preserve a one-to-many relationship directly in one table.

Lesson 9 of 14

Working with JSON

JSON is a widely used format that allows for semi-structured data, because it does not require a schema. Applications can use a "schema-on-read" approach, where the application ingests the data and then queries based on assumptions about the schema of that data. This approach differs from the STRUCT type in BigQuery, which requires a fixed schema that is enforced for all values stored in a column of STRUCT type.

By using the JSON data type, you can ingest semi-structured JSON into BigQuery without providing a schema for the JSON data upfront. This lets you store and query data that doesn't always adhere to fixed schemas and data types. By ingesting JSON data as a JSON data type, BigQuery can encode and process each JSON field individually. You can then query the values of fields and array elements within the JSON data by using the field access operator, which makes JSON queries easy to use and cost efficient.

Working with JSON data

BigQuery natively supports JSON data using the JSON data type. You can ingest JSON data into a BigQuery table in the following ways:

- Use a batch load job to load into JSON columns from CSV, Avro, and JSON files.
- Use the BigQuery Storage Write API.
- Use the legacy tabledata.insertAll streaming API.

JSON is case-sensitive and supports UTF-8 in both fields and values. You can use Google Standard SQL to query JSON in the following ways:

- Extract values as JSON
- Extract values as strings
- Extract arrays from JSON
- Work with JSON NULLs (specifically, the JSON type has a special null value that is different from the SQL NULL)

References

- Learn more about the various options for defining schemas on the documentation titled "Specifying a schema".
- Learn more about nested and repeated fields from the documentation titled "Use nested and repeated fields."
- Learn more about schema auto-detection from the documentation titled "Using schema auto-detection."
- Learn about working with JSON for semi-structured data from the documentation titled "Working with JSON data in Google Standard SQL."
- Learn more about the BigQuery Migration Service from the documentation page titled "Introduction to BigQuery Migration Service."

Lesson 11 of 14

Lesson introduction

In this lesson, you will learn about common patterns and best practices for optimizing schemas in BigQuery, including partitioning, clustering, and search indexes.

Let's get started!

Lesson 12 of 14

Partitioning and clustering

Partitioning BigQuery tables

A partitioned table is a special table that is divided into segments, called partitions. These partitions make it easier to manage and query data. By dividing a large table into smaller partitions, you can improve query performance and control costs by reducing the number of bytes read by a query. You can partition BigQuery tables in three ways.

Click the tabs to learn more.

TIME-UNIT COLUMN | INGESTION TIME | INTEGER RANGE

- Tables are partitioned based on a TIMESTAMP, DATE, or DATETIME column in the table.
- For TIMESTAMP and DATETIME columns, the partitions can have either hourly, daily, monthly, or yearly granularity. For DATE columns, the partitions can have daily, monthly, or yearly granularity.
- When you write data to the table, BigQuery automatically puts the data into the correct partition, based on the time/date values in the column.

TIME-UNIT COLUMN | INGESTION TIME | INTEGER RANGE

- Tables are partitioned based on the timestamp of when BigQuery ingests the data.
- You can choose hourly, daily, monthly, or yearly granularity for the partitions.
- When you write data to the table, BigQuery automatically assigns rows to partitions based on the time when BigQuery ingests the data.

TIME-UNIT COLUMN | INGESTION TIME | INTEGER RANGE

- Tables are partitioned based on an integer column such as customer_id.
- To create an integer range partitioned table, you can specify the starting and ending values for the range and the interval of each range.
- When you write data to the table, BigQuery automatically assigns rows to partitions based on the integer range contained in the partition.

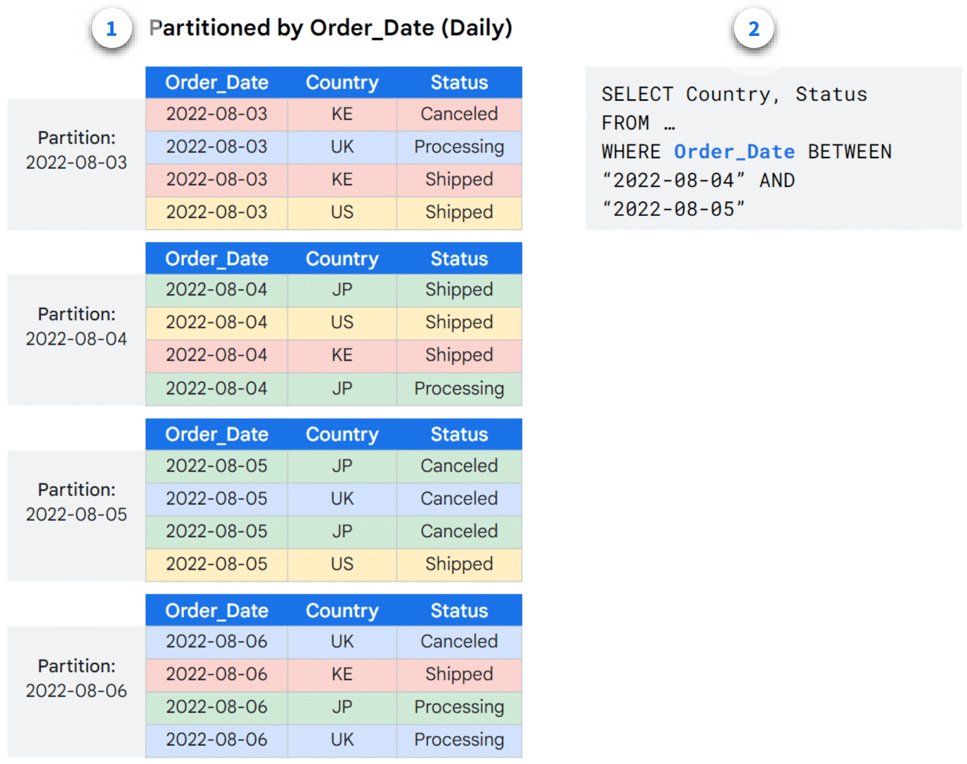
Using partitioned tables to prune data for queries

If a query uses a qualifying filter on the value of the partitioning column, BigQuery can scan the partitions that match the filter and skip the remaining partitions.

For example, consider a table that has been partitioned by the eventDate column. Based on this partition definition, BigQuery will split and store records in separate shards, depending on the value for eventDate. When you run a query on the table using a filter on eventDate, BigQuery does not have to scan the entire table to filter by date. Instead, BigQuery can completely ignore records in certain partitions that do not cover the specific date range provided in the query.

BigQuery maintains the defined partitions across all operations that read, write, or modify data in the table including query jobs, load and copy jobs, and data manipulation language (DML) and data definition language (DDL) statements. While this requires BigQuery to maintain more metadata than for a non-partitioned table, the cost and time savings for queries can be significant, particularly as tables grow larger.

Click the numbers in sequence to learn more.



1 **Partitioned by Order_Date (Daily)**

Partition: 2022-08-03

Order_Date	Country	Status
2022-08-03	KE	Canceled
2022-08-03	UK	Processing
2022-08-03	KE	Shipped
2022-08-03	US	Shipped

Partition: 2022-08-04

Order_Date	Country	Status
2022-08-04	JP	Shipped
2022-08-04	US	Shipped
2022-08-04	KE	Shipped
2022-08-04	JP	Processing

Partition: 2022-08-05

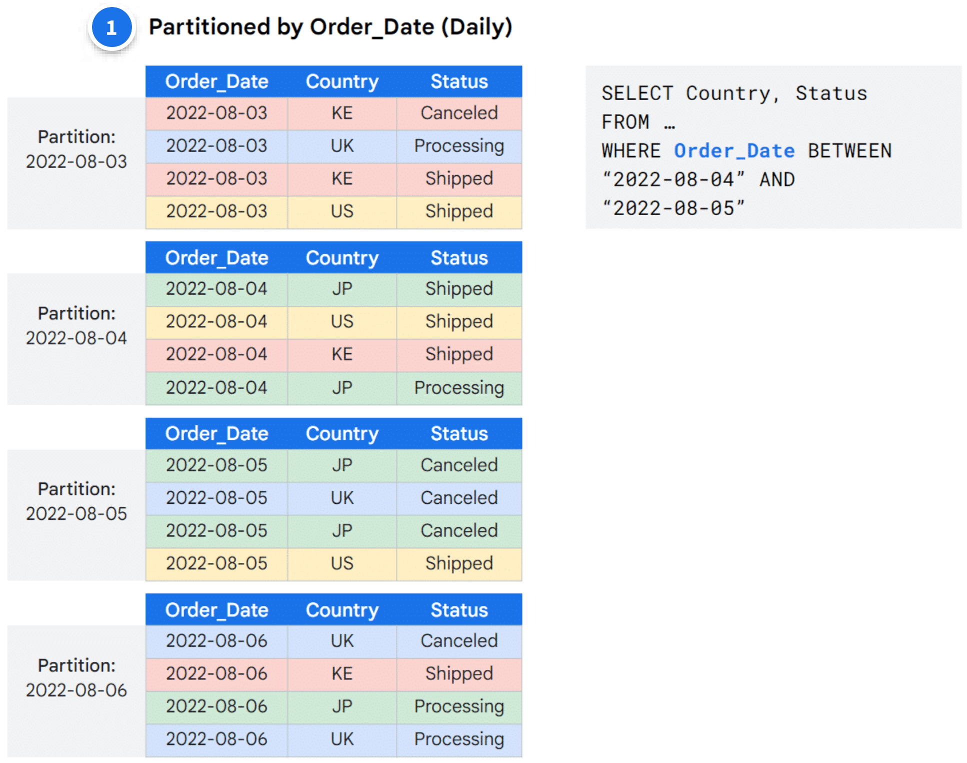
Order_Date	Country	Status
2022-08-05	JP	Canceled
2022-08-05	UK	Canceled
2022-08-05	JP	Canceled
2022-08-05	US	Shipped

Partition: 2022-08-06

Order_Date	Country	Status
2022-08-06	UK	Canceled
2022-08-06	KE	Shipped
2022-08-06	JP	Processing
2022-08-06	UK	Processing

2

```
SELECT Country, Status
FROM …
WHERE Order_Date BETWEEN
“2022-08-04” AND
“2022-08-05”
```



Order_Date	Country	Status
2022-08-03	KE	Canceled
2022-08-03	UK	Processing
2022-08-03	KE	Shipped
2022-08-03	US	Shipped

Order_Date	Country	Status
2022-08-04	JP	Shipped
2022-08-04	US	Shipped
2022-08-04	KE	Shipped
2022-08-04	JP	Processing

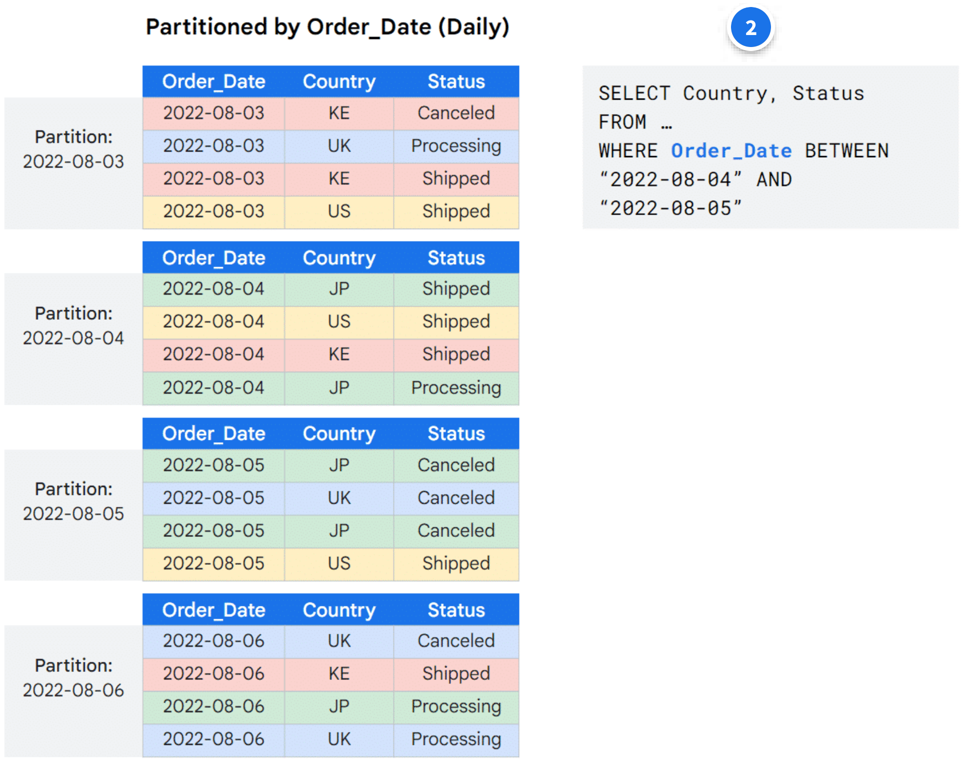
Order_Date	Country	Status
2022-08-05	JP	Canceled
2022-08-05	UK	Canceled
2022-08-05	JP	Canceled
2022-08-05	US	Shipped

Order_Date	Country	Status
2022-08-06	UK	Canceled
2022-08-06	KE	Shipped
2022-08-06	JP	Processing
2022-08-06	UK	Processing

```
SELECT Country, Status
FROM …
WHERE Order_Date BETWEEN
“2022-08-04” AND
“2022-08-05”
```

Partitioning by Order_Date

The default partitioning type for date, datetime, and timestamp columns is daily partitioning. This example table is partitioned by a column called Order_Date; there are four partitions for each day between Aug 3, 2022 and Aug 6, 2022.



Partitioned by Order_Date (Daily)

Partition: 2022-08-03

Order_Date	Country	Status
2022-08-03	KE	Canceled
2022-08-03	UK	Processing
2022-08-03	KE	Shipped
2022-08-03	US	Shipped

Partition: 2022-08-04

Order_Date	Country	Status
2022-08-04	JP	Shipped
2022-08-04	US	Shipped
2022-08-04	KE	Shipped
2022-08-04	JP	Processing

Partition: 2022-08-05

Order_Date	Country	Status
2022-08-05	JP	Canceled
2022-08-05	UK	Canceled
2022-08-05	JP	Canceled
2022-08-05	US	Shipped

Partition: 2022-08-06

Order_Date	Country	Status
2022-08-06	UK	Canceled
2022-08-06	KE	Shipped
2022-08-06	JP	Processing
2022-08-06	UK	Processing

2

```
SELECT Country, Status
FROM …
WHERE Order_Date BETWEEN
“2022-08-04” AND
“2022-08-05”
```

Pruning data by Order_Date

When you run a query on this table using a WHERE clause for dates between Aug 4, 2022 and Aug 5, 2022, BigQuery will only have to read one-half of the data and can ignore the other two partitions, resulting in faster and more cost-effective queries.

Benefits of partitioned tables

Click the flash card to learn more.

Accurate query cost estimate

Partition pruning is done before the query runs, so you can get the query cost after

	partitioning pruning through a dry run.
Partition-level management	You can easily manage your partitions to set a partition expiration time, load data to a specific partition, or delete partitions.
More performant than manual table sharding	With manually sharded tables, BigQuery must maintain a copy of the schema and metadata for each table. BigQuery might also need to verify

Same table access control options

Access control for partitioned tables is the same as access control for standard tables in BigQuery.

Many no-cost operations

In BigQuery, many partitioned table operations can be executed at no cost, including loading data into partitions, copying partitions, and exporting data

Clustering within BigQuery tables

Clustered tables in BigQuery are tables that have a user-defined column sort order using clustered columns. Clustered tables can improve query performance and reduce query costs.

In BigQuery, a clustered column is a user-defined table property that sorts storage blocks based on the values in the clustered columns. Queries that filter or aggregate by the clustered columns only scan the relevant blocks based on the clustered columns instead of the entire table or table partition. BigQuery supports clustering for both partitioned and non-partitioned tables.

Benefits of clustered tables

Click the flash card to learn more.

Faster filtering on many column types

In BigQuery, you can cluster columns that uses the following data types: STRING, INT64, NUMERIC/BIGNUMERIC, DATE, DATETIME, TIMESTAMP, BOOL, and GEOGRAPHY. Clustering accelerates queries

Faster filtering on high cardinality columns

When your queries filter on columns that have many distinct values (high cardinality), clustering accelerates these queries by providing BigQuery with detailed metadata on which blocks of records match the filter. Once

Automatic reclustering	As data in the clustered table is modified, BigQuery performs automatic reclustering in the background to maintain the
Same table access control options	Access control for clustered tables is the same as access control for standard tables in BigQuery.
Many no-cost operations	Like other BigQuery table operations, clustered table operations take advantage of no-cost

operations in

BigQuery such as

Combining partitioning and clustering

You can combine table clustering with table partitioning to achieve finely grained sorting for further query optimization. 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.

Click the + icon to expand and learn more.

Does using clustering and partitioning together improve performance? –

Yes. When you combine partitioning with clustering, data is first partitioned, and then data in each partition is clustered by the clustering columns.

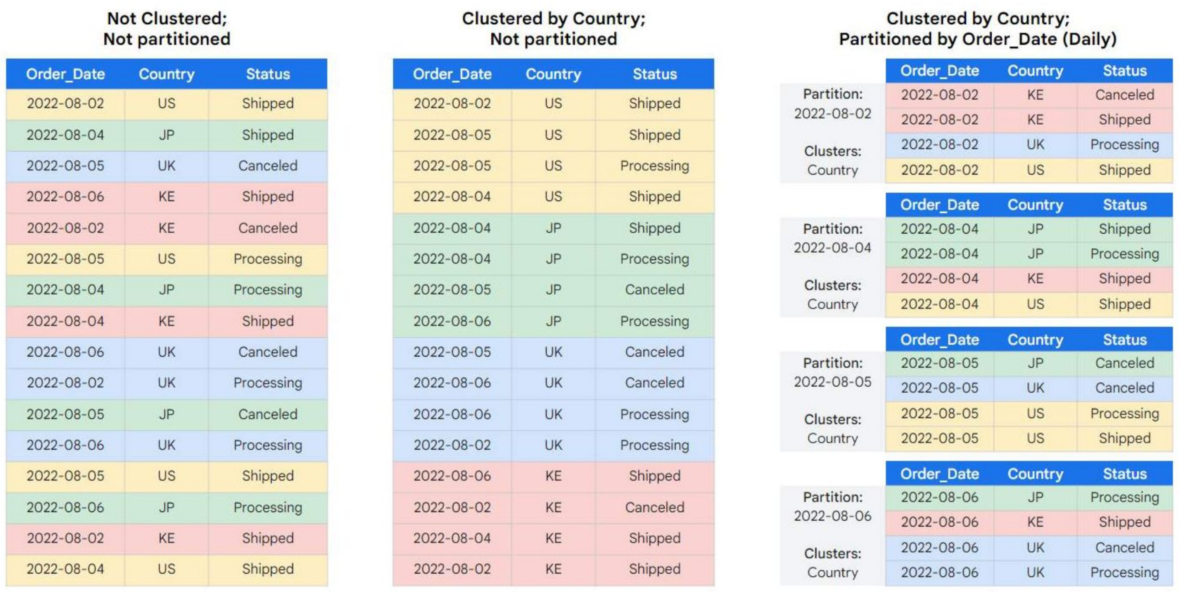
When the table is queried, partitioning sets an upper bound of the query cost based on partition pruning. In addition, there may be other query cost savings when the query actually runs because of cluster pruning.

Do clustering and partitioning need to be executed together? –

No. Both clustering and partitioning can be implemented separately in BigQuery.

For example, if your partitions would be very small (approximately less than 1 GB) or if you need to frequently update most partitions in the table (for example, every few minutes), then it’s more performant to just use clustering.

Clustered and partitioned tables



Not Clustered; Not partitioned

Order_Date	Country	Status
2022-08-02	US	Shipped
2022-08-04	JP	Shipped
2022-08-05	UK	Canceled
2022-08-06	KE	Shipped
2022-08-02	KE	Canceled
2022-08-05	US	Processing
2022-08-04	JP	Processing
2022-08-04	KE	Shipped
2022-08-06	UK	Canceled
2022-08-02	UK	Processing
2022-08-05	JP	Canceled
2022-08-06	UK	Processing
2022-08-05	US	Shipped
2022-08-06	JP	Processing
2022-08-02	KE	Shipped
2022-08-04	US	Shipped

Clustered by Country; Not partitioned

Order_Date	Country	Status
2022-08-02	US	Shipped
2022-08-05	US	Shipped
2022-08-05	US	Processing
2022-08-04	US	Shipped
2022-08-04	JP	Shipped
2022-08-04	JP	Processing
2022-08-05	JP	Canceled
2022-08-06	JP	Processing
2022-08-05	UK	Canceled
2022-08-06	UK	Canceled
2022-08-06	UK	Processing
2022-08-02	UK	Processing
2022-08-06	KE	Shipped
2022-08-02	KE	Canceled
2022-08-04	KE	Shipped
2022-08-02	KE	Shipped

Clustered by Country; Partitioned by Order_Date (Daily)

Partition: 2022-08-02

Clusters: Country

Order_Date	Country	Status
2022-08-02	KE	Canceled
2022-08-02	KE	Shipped
2022-08-02	UK	Processing
2022-08-02	US	Shipped

Partition: 2022-08-04

Clusters: Country

Order_Date	Country	Status
2022-08-04	JP	Shipped
2022-08-04	JP	Processing
2022-08-04	KE	Shipped
2022-08-04	US	Shipped

Partition: 2022-08-05

Clusters: Country

Order_Date	Country	Status
2022-08-05	JP	Canceled
2022-08-05	UK	Canceled
2022-08-05	US	Processing
2022-08-05	US	Shipped

Partition: 2022-08-06

Clusters: Country

Order_Date	Country	Status
2022-08-06	JP	Processing
2022-08-06	KE	Shipped
2022-08-06	UK	Canceled
2022-08-06	UK	Processing

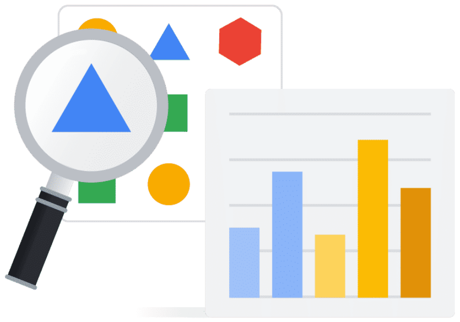
Lesson 13 of 14

Search indexes

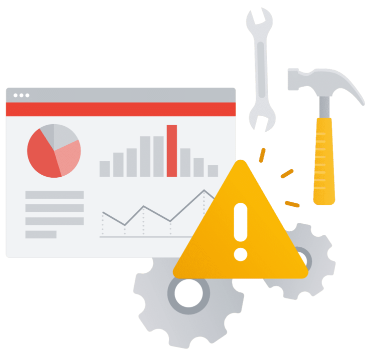
BigQuery search indexes let you use Google Standard SQL to easily find unique data elements that are buried in unstructured text and semi-structured JSON data. This is possible without having to know the table schemas in advance.

With search indexes, BigQuery provides a powerful columnar store and text search in one platform, enabling efficient row lookups when you need to find individual rows of data. BigQuery search indexes help you perform many tasks.

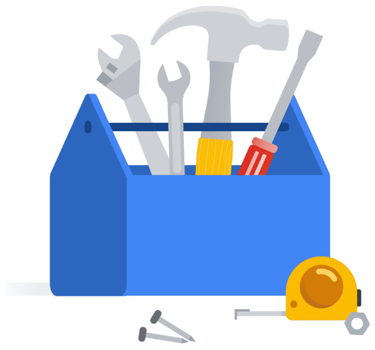
Click the flash cards to learn more.



Search system, network, or application logs stored in BigQuery tables.



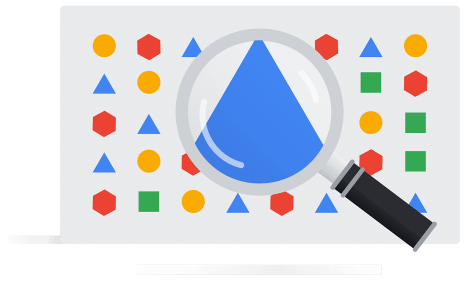
Identify data elements for deletion to comply with regulatory processes.



Support developer troubleshooting.

Perform security audits.

Create a dashboard that requires highly selective search filters.



Search pre-processed data for exact matches.

Creating a search index

To create a search index, use the CREATE SEARCH INDEX DDL statement. You can create a search index on these column types:

- STRING
- ARRAY<STRING>
- STRUCT containing at least one nested field of type STRING or ARRAY<STRING>
- JSON

When you create a search index, you can specify the type of text analyzer to use. The text analyzer controls how data is tokenized for indexing and searching. There are two options for the text analyzer:

- LOG_ANALYZER is the default. This analyzer works well for machine generated logs and has special rules around tokens commonly found in observability data, such as IP addresses or emails.
- NO_OP_ANALYZER is useful when you have pre-processed data that you want to match exactly. No tokenization or normalization is applied to the text.

Search indexes are fully managed by BigQuery and automatically refresh when the table changes. The following schema changes can trigger a full refresh:

- A new indexable column is added to a table with a search index on ALL COLUMNS.
- An indexed column is updated due to a table schema change.

If you delete the only indexed column in a table or rename the table itself, then the search index is deleted automatically.

Best practices for creating search indexes

Click the + icon to expand and learn more.

Apply on large tables –

Search indexes are designed for large tables. The performance gains from a search index increase with the size of the table.

Avoid applying search indexes on all columns –

Avoid indexing columns that contain only a very small number of unique values or that you never intend to call the SEARCH function on.

Use a slot reservation for index management in production environments –

Instead of using the default shared slot pool, you can optionally designate your own reservation to index your tables. This allows you to ensure predictable and consistent performance of index-management jobs, such as creation, refresh, and background optimizations.

Retrieving your data

BigQuery stores and manages the indexes, so when data becomes available in BigQuery, you can immediately retrieve it with the SEARCH function. Much like the index you'd find in the back of a book, a search index for a column of string data acts like an auxiliary table that has one column for unique words and another for where those words occur in the data.

References

- Learn more about partitioning from the documentation page titled "Introduction to partitioned tables."
- Learn more about clustering from the documentation titled "Introduction to clustered tables."
- Learn more about search indexes from the documentation titled "Manage search indexes."
- Learn more about text analyzers in BigQuery from the documentation titled "Text analyzer rules."