Chapter 7
Designing Databases for Reliability, Scalability, and Availability

Google Cloud Professional Data Engineer Exam objectives covered in this chapter include the following:

- **4. Ensuring solution quality**
- **✓ 4.2 Ensuring scalability and efficiency. Considerations include:**
- Building and running test suites
- Assessing, troubleshooting, and improving data representations and data processing infrastructure
- Resizing and autoscaling resources
- **✓ 4.3 Ensuring reliability and fidelity. Considerations include:**
- Performing data preparation and quality control (e.g., Cloud Dataprep)
- Verification and monitoring
- Planning, executing, and stress testing data recovery (fault tolerance, rerunning failed jobs, performing retrospective re-analysis)

A significant amount of a data engineer's time can go into working with databases. Chapter 2, "Building and Operationalizing Storage Systems," introduced the GCP databases and highlighted their features and some use cases. In this chapter, we will delve into more detail about designing for reliability, scalability, and availability of three GCP databases:

- Cloud Bigtable
- Cloud Spanner
- Cloud BigQuery

A fourth database, Cloud SQL, could be used as well, but it does not scale beyond the region level. If you do need a multi-regional relational database, you should use Cloud Spanner.

By the end of this chapter, you should have an understanding of how to apply best practices for designing schemas, querying data, and taking advantage of the physical design properties of each database.

Designing Cloud Bigtable Databases for Scalability and Reliability

Cloud Bigtable is a nonrelational database based on a sparse three-dimensional map. The three dimensions are rows, columns, and cells. By understanding this 3D map structure, you will be able to figure out the best way to design Cloud Bigtable schemas and tables to meet scalability, availability, and reliability requirements. In particular, we will consider the following:

- Data modeling with Cloud Bigtable
- Designing row-keys
- Designing for time-series data
- Using replication to improve scalability and availability

DATA MODELING WITH CLOUD BIGTABLE

The 3D map structure of Cloud Bigtable lends itself to distributed implementations. Cloud Bigtable databases are deployed on a set of resources known as an *instance*. (This is not the same thing as a virtual machine (VM) instance in Compute Engine.) An *instance* consists of a set of nodes, which are virtual machines, as well as a set of sorted string tables and log data that is stored in Google's Colossus filesystem. Figure 7.1 shows the basic architecture of Cloud Bigtable.

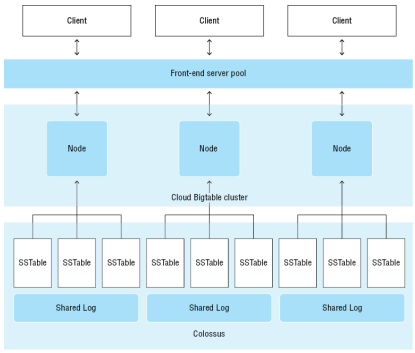

Figure 7.1 Cloud Bigtable uses a cluster of VMs and the Colossus filesystem for storing, accessing, and managing data.

Source: `https://cloud.google.com/bigtable/docs/overview`

When you create a Cloud Bigtable instance, you specify a number of nodes. These nodes manage the metadata about the data stored in the Cloud Bigtable database, whereas the actual data is stored outside of the nodes on the Colossus filesystem. Within the Colossus filesystem, data is organized into sorted string tables, or SSTables, which are known as tablets in GCP terminology. As the name implies, data is stored in a sorted order. SSTables are also immutable.

Separating metadata from data has several advantages, including the following:

- Fast rebalancing of tablets from one node to another
- The ability to recover a failed node by migrating metadata only
- The scalability, reliability, and availability of the Colossus filesystem for data

A master process balances workloads in a cluster, including splitting and merging tablets.

In Cloud Bigtable, *rows* are indexed by a *row-key*, which is analogous to a primary key in a relational database. Data is stored in a lexicographically sorted order by row-key. Data is retrieved from Bigtable by using a row-key to specify a particular row or a range of row-keys to specify a set of contiguous rows. There are no secondary indexes in Cloud Bigtable, so it is important to have a row-key that is designed to support the query patterns used with the database.

Columns are the second dimension. They are grouped into column families. This makes it more efficient when retrieving data that is frequently used together. For example, a street address, city, state, and postal code could be stored in four columns that are grouped into a single address column family. Columns are sorted lexicographically within a column family.

Cells are the third dimension, and they are used to store multiple versions of a value over time. For example, the intersection of a row and column may store the shipping address of a customer. When an update is made to the address, a new version is added and indexed using a timestamp. By default, Bigtable returns the value in the cell with the latest timestamp.

Cloud Bigtable tables are sparse—that is, if there is no data for a particular row/column/cell combination, then no storage is used.

When you are designing tables, fewer large tables is preferred to many small tables. Small tables require more backend connection overhead. In addition, different table sizes can disrupt the load balancing that occurs in the background.

All operations are atomic at the row level. This favors keeping related data in a single row so that when multiple changes are made to related data, they will all succeed or fail together. There are limits, though, on how much data should be stored in a single row. Google recommends storing no more than 10 MB in a single cell and no more than 100 MB in a single row.

DESIGNING ROW-KEYS

One of the most important design choices in Cloud Bigtable is the choice of a row-key. The reason for this is that Cloud Bigtable scales best when read and write operations are distributed across nodes and tablets. If read and write operations are concentrated in a small number of nodes and tablets, the overall performance is limited to the performance of those few resources.

Row-key Design Best Practices

In general, it is best to avoid monotonically increasing values or lexicographically close strings at the beginning of keys because this can cause hotspots. Google notably makes an exception to this rule and suggests using reverse domain names as a row-key when the entity can be represented as a domain name and there are many domain names. For example, the domain `mysubdomain.groupsubdomain.example.com` can be reversed to create a row-key called `com.example.groupsubdomain.domain`. This is helpful if there is some data that is repeated across rows. In that case, Cloud Bigtable can compress the repeated data efficiently. Reverse domain names should not be used as row-keys if there are not enough distinct domains to distribute data adequately across nodes.

Also, when a using a multitenant Cloud Bigtable database, it is a good practice to use a tenant prefix in the row-key. This will ensure that all of a customer's data is kept together and not intermingled with other customers' data. In this situation, the first part of the row-key could be a customer ID or other customer-specific code.

String identifiers, such as a customer ID or a sensor ID, are good candidates for a row-key. *Timestamps* may be used as part of a row-key, but they should not be the entire row-key or the start of the row-key. This is especially helpful when you need to perform range scans based on time. The first part of a row-key that includes a timestamp should be a high-cardinality—that is, a large number of possible values—attribute, such as a sensor ID. For example, `1873838#1577569258` is a sensor ID concatenated with a timestamp in seconds past the epoch. This ordering will work as long as the sensors report in a fairly random order.

Moving timestamps from the front of a row-key so that another attribute is the first part of the row-key is an example of *field promotion*. In general, it is a good practice to promote, or move toward the front of the key, values that are highly varied.

Another way to avoid hotspots is to use *salting*, which is the process of adding a derived value to the key to make writes noncontiguous. For example, you could use the hash of a timestamp divided by the number of nodes in a cluster. This approach would evenly distribute the write load across all nodes. Of course, this simple salting function depends on the number of nodes in the cluster, which can change over time.

Antipatterns for Row-key Design

In addition to employing best practices for designing row-keys, you need to avoid some antipatterns. Specifically, avoid the following:

- Domain names
- Sequential numeric IDs
- Frequently updated identifiers
- Hashed values

In the case of the domain names and sequential IDs, you can create hotspots for read and write operations. Frequently updated identifiers should not be used because the update operations can overload the tablet that stores the row that is often updated.

Key Visualizer

Key Visualizer is a Cloud Bigtable tool for understanding usage patterns in a Cloud Bigtable database. This tool helps you identify the following:

- Where hotspots exist
- Rows that may contain too much data
- The distribution of workload across all rows in a table

Key Visualizer generates hourly and daily scans of tables that have at least 30 GB of data at some point in time and for which there was an average of at least 10,000 reads and writes per second over the previous 24 hours. Data from Key Visualizer is displayed in a heatmap in which dark areas have low activity and bright areas have heavy activity (see **Figure 7.2** for an example).

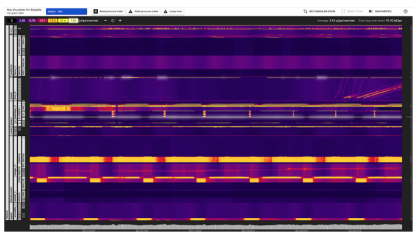

Figure 7.2 An example heatmap generated by Cloud Bigtable Key Visualizer

Source: `https://cloud.google.com/bigtable/docs/keyvis-overview`

DESIGNING FOR TIME SERIES

Time-series data, such as financial data and IoT data streams, is well suited to Cloud Bigtable. The managed database provides low-latency writes, and it is highly scalable, which is needed for large-volume time-series data. Time-series data is a natural fit for tables with many rows and

few columns. These are known as tall and narrow. Also, time-series data is often queried using time ranges as filters. Data that is likely to be queried together—for example, data from an IoT device that was generated within the same hour, day, and so on—can be stored together in Cloud Bigtable. This storage pattern can help reduce the volume of data scanned to answer a query.

Google has several recommendations for designing for time-series data:

Keep names short. This reduces the size of metadata since names are stored along with data values.

Favor tables with many rows and few columns ("tall and narrow" tables). Within each row, store few events—ideally only one event per row. This makes querying easier. Also, storing multiple events increases the chance of exceeding maximum recommended row sizes.

Using tall and narrow tables typically leads to one table per modeled entity. For example, consider two IoT sensors: one is an environmental sensor that measures temperature, humidity, and pressure, and the other is a sensor that counts the number of vehicles passing the sensor. Both devices collect data at the same location and at the same frequency. Although the two types of sensors may be part of the same application, the data from each should be stored in their own tables.

Design row-keys for looking up a single value or a range of values. Range scans are common in time-series analysis. Keep in mind that there is only one index on Cloud Bigtable tables. If you need to look up or range scan data in different orders, then you will need to denormalize the data and create another table with a different ordering. For example, if you have an IoT application that groups data by sensor type and by monitored device type, you could create one table with the sensor type in the row-key and another table with the monitored device type in the row-key. To minimize storage, only the data that is needed to answer queries should be stored in the additional table.

USE REPLICATION FOR AVAILABILITY AND SCALABILITY

Cloud Bigtable replication allows you to keep multiple copies of your data in multiple regions or multiple zones. This approach can improve availability and durability. You can also take advantage of replicated data to distribute workloads across multiple clusters.

When creating a Cloud Bigtable instance, you can create multiple clusters, as highlighted in Figure 7.3.

Cloud Bigtable supports up to four replicated clusters. Clusters can be regional or multi-regional. When a change is made to one of the clusters, it is replicated in the others. This includes the following:

- Updating data in tables
- Adding and deleting tables
- Adding and deleting column families
- Configuring garbage collection

Note that there is no single primary cluster. All replicated clusters can perform read and write operations. Cloud Bigtable is eventually consistent, so all replicas should be updated within a few seconds or minutes in most cases. Performing large updates or replicating clusters that are far apart can take longer.

If you want to ensure that a cluster will never read data that is older than its most recent writes, you can specify read-your-writes consistency by stipulating single-cluster routing in an app profile. *App profiles* are configurations that specify how to handle client requests. If you want strong consistency, you would have to specify single-cluster routing in the app configuration, and you must not use the other clusters except for failover. If you want Cloud Bigtable to fail over automatically to one region if your application cannot reach another region, you should use multicluster routing.

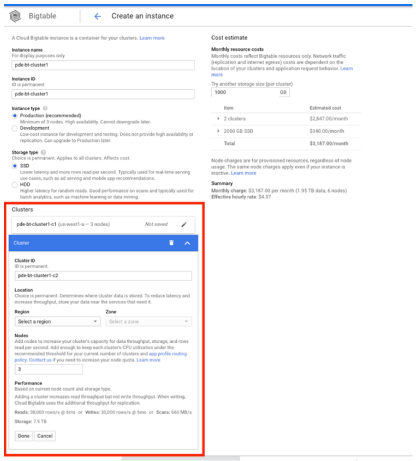

Figure 7.3 Specifying multiple clusters when creating a Cloud Bigtable instance

Designing Cloud Spanner Databases for Scalability and Reliability

Cloud Spanner is a globally scalable relational database that offers many of the scalability benefits once limited to NoSQL databases. It does so while maintaining key features of relational databases, such as support for normalized data models and transaction support. Data engineers need to understand factors that influence the performance of Cloud Spanner databases in order to avoid design choices that can limit performance and scalability.

For the purposes of the Cloud Professional Data Engineer exam, it is important to understand several aspects of Cloud Spanner, including

- Relational database features
- Interleaved tables
- Primary keys and hotspots
- Database splits
- Secondary indexes
- Query best practices

RELATIONAL DATABASE FEATURES

Many of Cloud Spanner features are common to other relational databases. The data model is centered around tables, which are composed of columns that represent attributes and rows that are collections of values for attributes about a single entity. For example, an e-commerce application may have tables about customers, orders, and inventory, with rows representing a single customer, order, and inventory item, respectively. The customer table may have columns such as customer name, address, and credit limit, whereas the order table may have an order ID, date of order, customer ID, and total value of the order. An inventory table could have columns with product names, quantity, and location of the item in a warehouse.

Relational databases are strongly typed. Cloud Spanner supports the following types:

- **Array:** An ordered list of zero or more elements of a non-array type
- **Bool:** Boolean `TRUE` or `FALSE`
- **Bytes:** Variable-length binary data
- **Date:** A calendar date
- **Float64:** Approximate numeric values with fractional components
- **INT64:** Numeric values without fractional components
- **String:** Variable-length characters
- **Struct:** Container of ordered typed fields
- **Timestamp**: A point in time with nanosecond precision

Cloud Spanner supports both primary and secondary indexes. The *primary index* is an index automatically created on the primary key of a table. Other columns or combinations of columns can be indexed as well using secondary indexes. Note that this is different from Cloud Bigtable, which does not support secondary indexes. This reduces the need to denormalize and duplicate data to support multiple query patterns. Whereas in Cloud Bigtable you would need to duplicate data to query the data efficiently without referencing the primary key, in Cloud Spanner you can create secondary indexes that support different query patterns.

INTERLEAVED TABLES

Another important performance feature of Cloud Spanner is its ability to interleave data from related tables. This is done through a parent-child relationship in which parent data, such as a row from the order table, is stored with child data, such as order line items. This makes retrieving

data simultaneously from both tables more efficient than if the data were stored separately and is especially helpful when performing joins. Since the data from both tables is co-located, the database has to perform fewer seeks to get all the needed data.

Cloud Spanner supports up to seven layers of interleaved tables. Tables are interleaved using the `INTERLEAVE IN PARENT` clause in a `CREATE TABLE` statement. For example, to interleave orders with the customers who placed the order, you could use the following commands:

```
CREATE TABLE Customers (
  CustomerId   INT64 NOT NULL,
  FirstName  STRING(1024),
  LastName   STRING(1024),
  AddressLine STRING(1024),
  City STRING(1024),
  State STRING(1024),
  PostalCode STRING(10),
) PRIMARY KEY (CustomerId);

CREATE TABLE Orders (
  CustomerId     INT64 NOT NULL,
  OrderId        INT64 NOT NULL,
  OrderDescription   STRING(MAX),
) PRIMARY KEY (CustomerId, OrderId)
,

  INTERLEAVE IN PARENT Customers ON DELETE CASCADE;
```

Interleaving is used when tables are frequently joined. By storing the related data together, joins can be performed with fewer I/O operations than if they were stored independently.

PRIMARY KEYS AND HOTSPOTS

Cloud Spanner is horizontally scalable, so it should be no surprise that it shares some characteristics with Cloud Bigtable, another horizontally scalable database. Cloud Spanner uses multiple servers, and it divides data among servers by key range. This creates the potential for hotspots just as in Cloud Bigtable. Monotonically increasing keys, for example, can cause read and write operations to happen in a few servers at once instead of being evenly distributed across all servers.

As with Cloud Bigtable, there are several recommended ways of defining primary keys to avoid hotspots. These include the following:

Using a Hash of the Natural Key Note that using a hash of the natural key is not recommended with Cloud Bigtable, but meaningless keys are regularly used in relational databases.

Swapping the Order of Columns in Keys to Promote High-Cardinality Attributes Doing so makes the start of the keys more likely to be nonsequential.

Using a Universally Unique Identifier (UUID), Specifically Version 4 or Later Some older UUIDs store timestamps in the high-order bits, and that can lead to hotspotting.

Using Bit-Reverse Sequential Values Bit-reverse sequential values can be used when the best candidate primary key is monotonically increasing. By reversing the value, you can eliminate the potential for hotspots because the start of the reversed value will vary from one value to the next.

Also, like Cloud Bigtable, Cloud Spanner has to manage breaking down datasets into manageable-sized units that can be distributed across servers.

DATABASE SPLITS

Cloud Spanner breaks data into chunks known as *splits*. Splits are up to about 4 GB in size and move independently of one another. A split is a range of rows in a top-level table—that is, a table that is not an interleaved part of another table. Rows in a split are ordered by primary key, and the first and last keys are known as the *split boundaries*.

Rows that are interleaved are kept with their parent row. For example, if there is a customer table with an interleaved order table and below that an interleaved order line item table, then all order and order line item rows are kept in the same split as the corresponding customer row. Here it is important to keep splits and split limits in mind when creating interleaved tables. Defining a parent-child relationship collocates data, and that could cause more than 4 GB of data to be interleaved. Four GB is the split limit, and exceeding it can lead to degraded performance.

Cloud Spanner creates splits to alleviate hotspots. If Cloud Spanner detects a large number of read/write operations on a split, it will divide the rows into two splits so that they can be placed on different servers. Figure 7.4 shows a Cloud Spanner instance with six nodes and varying amounts of data in each (shown in gray). Cloud Spanner may have placed data into this distribution to balance the read/write workload across nodes, not just the amount of data across nodes.

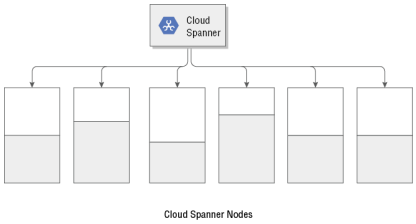

Figure 7.4 Cloud Spanner distributes splits across servers to avoid hotspots.

SECONDARY INDEXES

Cloud Spanner provides for secondary indexes on columns or groups of columns other than the primary key. An index is automatically created for the primary key, but you have to define any secondary indexes specifically.

Secondary indexes are useful when filtering in a query using a `WHERE` clause. If the column referenced in the `WHERE` clause is indexed, the index can be used for filtering rather than scanning the full table and then filtering. Secondary indexes are also useful when you need to return rows in a sort order other than the primary key order.

When a secondary index is created, the index will store the following:

- All primary key columns from the base table
- All columns included in the index
- Any additional columns specified in a `STORING` clause

The `STORING` clause allows you to specify additional columns to store in an index but not include those columns as part of the index. For example, if you wanted a secondary index on the customer table by postal code, you could create it as follows:

```
CREATE INDEX CustomerByPostalCode ON Customers(PostalCode);
```

If you frequently need to refer to the customer's last name when you look up by postal code, you can have the secondary index store the last name along with the postal code by specifying the `LastName` column in the `STORING` clause of the `CREATE INDEX` command, such as in the following line of code:

```
CREATE INDEX CustomerByPostalCode ON Customers(PostalCode)STORING (Last
```

Since the index has the primary key columns, the secondary index columns, and the additional `LastName` column, any query that references only those columns can be performed by retrieving data from the index only and not from the base table as well.

When Cloud Spanner develops an execution plan to retrieve data, it will usually choose the optimal set of indexes to use. If testing indicates that the best option is not being selected by the query, you can specify a `FORCE_INDEX` clause in the `SELECT` query to have Cloud Spanner use the specified index. Consider the following query:

```
SELECT CustomerID, LastName, PostalCode
FROM Customers
WHERE PostalCode  = '99221'
```

This query would benefit from using a secondary index on the `PostalCode` column. If the Cloud Spanner execution plan builder does not choose to use that index, you can force its use by specifying a `FORCE_INDEX` clause using the index directive syntax:

```
SELECT CustomerID, LastName, PostalCode
FROM Customers@{FORCE_INDEX=CustomerByPostalCode}
WHERE PostalCode  = '99221'
```

QUERY BEST PRACTICES

Here are some other best practices to keep in mind when using Cloud Spanner.

Use Query Parameters

When a query is executed, Cloud Spanner builds an execution plan. This involves analyzing statistics about the distribution of data, existing secondary indexes, and other characteristics of the tables in the query. If a query is repeated with changes to only some filter criteria, such as a postal code, then you can avoid rebuilding the query execution plan each time the query is executed by using *parameterized queries*.

Consider an application function that returns a list of customer IDs for a given postal code. The query can be written by using literal values for the postal code as in the following snippet:

```
SELECT CustomerID
FROM Customers
WHERE PostalCode  = '99221'
```

If this query is repeated with changes only to the literal value, you can use a query parameter instead of the literal values. A parameter is specified using the @ sign followed by the parameter name, such as the following:

```
SELECT CustomerID
FROM Customers
WHERE PostalCode  = @postCode
```

The value of the parameter is specified in the `params` field of the `ExecuteSQL` or `ExecuteStreamingSQL` API request.

Use *EXPLAIN PLAN* to Understand Execution Plans

A *query execution plan* is a series of steps designed to retrieve data and respond to a query. Since Cloud Spanner stores data in splits that are distributed across a cluster, execution plans have to incorporate local execution of subplans—that is, subplans are executed on each node, and then the results are aggregated across all nodes.

Figure 7.5 shows a logical example of an execution plan for a query that selects the name of songs from a table of songs.

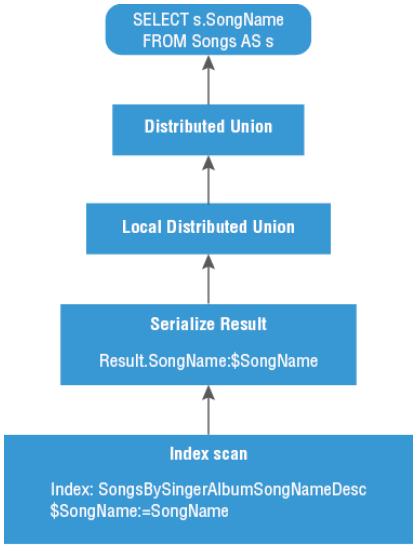

Figure 7.5 An example query execution plan

Source: `https://cloud.google.com/spanner/docs/query-execution-plans`

To see an execution plan from the GCP console, select a Cloud Spanner instance and database, run a query from the console, and use the `Explanation` option to list the plan.

Avoid Long Locks

When executing a transaction, the database may use locks to protect the integrity of data. There are some situations in which you should use locking read/write transactions, including

- If a write depends on the result of one or more reads, the reads and the write should be in a read/write transaction.
- If several (more than one) write operations have to be committed together, the writes should be in a read/write transaction.
- If there may be one or more writes depending on the results of a read, the operations should be in a read/write transaction.

When using read/write transactions, you want to keep the number of rows read to a minimum. This is because during a read/write transaction, no other processes can modify those rows until the transaction commits or rolls back. Avoid full table scans or large joins within a read/write transaction. If you do need to perform a large read, do it within a read-only transaction that does not lock rows.

Designing BigQuery Databases for Data Warehousing

BigQuery is an analytical database designed for data warehousing, machine learning (ML), and related analytic operations. Although BigQuery uses standard SQL as a query language, it is not a relational database. Some relational data modeling practices are applicable to BigQuery databases, and others are not. In this section, you'll see how to design BigQuery databases that scale while remaining performant and cost-effective. Specifically, this section covers the following:

- Schema design for data warehousing
- Clustered and partitioned tables
- Querying data
- External data access

- BigQuery ML

The focus here is on leveraging BigQuery features to support large-scale data warehousing. Additional features of BigQuery are discussed in Chapter 2.

SCHEMA DESIGN FOR DATA WAREHOUSING

Data warehousing is the practice of creating and using databases for primarily analytic operations. This is distinct from databases designed for transaction processing, such as an e-commerce application for selling products or an inventory management system for tracking those products.

Types of Analytical Datastores

The term *data warehousing* is sometimes used broadly to include a range of data stores for supporting analysis. These include the following:

Data Warehouses These are centralized, organized repositories of analytical data for an organization.

Data Marts These are subsets of data warehouses that focus on particular business lines or departments.

Data Lakes These are less structured data stores for raw and lightly processed data.

BigQuery is designed to support data warehouses and data marts. Data lakes may be implemented using object storage, such as Cloud Storage, or NoSQL databases, such as Cloud Bigtable.

Projects, Datasets, and Tables

At the highest levels, BigQuery data is organized around projects, datasets, and tables.

Projects are the high-level structure used to organize the use of GCP services and resources. The use of APIs, billing, and permissions are managed at the project level. From a BigQuery perspective, projects contain datasets and sets of roles and permissions for working with BigQuery.

Datasets exist within a project and are containers for tables and views. When you create a dataset, you specify a geographic location for that dataset. The location of a dataset is immutable. Access to tables and views are defined at the dataset level. For example, a user with read access to a dataset will have access to all tables in the dataset.

Tables are collections of rows and columns stored in a columnar format, known as *Capacitor format*, which is designed to support compression and execution optimizations. After data is encoded in Capacitor format, it is written to the Colossus distributed filesystem.

BigQuery tables are subject to the following limitations:

- Table names within a dataset must be unique.
- When you are copying a table, the source and destination table must be in the same location.
- When you are copying multiple tables to a single destination table, all source tables must have the same schema.

Tables support the following data types for columns:

- **Array:** An ordered list of zero or more elements of a non-array type
- **Bool:** Boolean `TRUE` or `FALSE`
- **Bytes:** Variable-length binary data
- **Date:** A calendar date
- **Datetime:** Represents a year, month, day, hour, minute, second, and subsecond; this data type does not include time zone
- **Numeric:** An exact numeric value with 38 digits of precision and 9 decimal digits
- **Float64:** Double-precision approximate numeric values with fractional components
- **INT64:** Numeric values without fractional components
- **Geography:** A collection of points, lines, and polygons representing a point set or subset of the earth's surface
- **String:** Variable-length characters; must be UTF-8 encoded
- **Struct:** Container of ordered typed fields
- **Time:** A time independent of a date
- **Timestamp:** A point in time with microsecond precision and that includes time zone

Most of these are atomic units or scalar values, but the array and struct types can store more complex structures.

Although BigQuery supports joins and data warehouses often use joins between fact tables and dimension tables, there are advantages to denormalizing a data model and storing related data together. In BigQuery, you can do this with nested and repeated columns.

A column that contains nested and repeated data is defined as a `RECORD` data type and is accessed as a `STRUCT` in SQL. Data that is in object-based schema formats, such as JavaScript Object Notation (JSON) and Apache

Avro files, can be loaded directly into BigQuery while preserving their nested and repeated structures.

BigQuery supports up to 15 levels of nested structs.

CLUSTERED AND PARTITIONED TABLES

BigQuery is most cost-efficient when you can minimize the amount of data scanned to retrieve queries. BigQuery does not have indexes like relational databases or document databases, but it does support partitioning and clustering, both of which can help limit the amount of data scanned during queries.

Partitioning

Partitioning is the process of dividing tables into segments called *partitions*. By segmenting data, BigQuery can take advantage of metadata about partitions to determine which of them should be scanned to respond to a query. BigQuery has three partition types:

- Ingestion time partitioned tables
- Timestamp partitioned tables
- Integer range partitioned tables

When a table is created with *ingestion time partitioning*, BigQuery loads data into a daily partition and creates new partitions each day. A date-based timestamp is stored in a pseudo-column called `_PARTITIONTIME`, which can be referenced in `WHERE` clauses to limit scanning to partitions created within a specified time period.

Timestamp partitioned tables partition based on a `DATE` or `TIMESTAMP` column in a table. Partitions have data from a single day. Rather than using `_PARTITIONTIME`, queries over timestamp partition tables reference the `DATE` or `TIMESTAMP` column used to partition data. Rows with null values in the `DATE` or `TIMESTAMP` column are stored in a `__NULL__` partition, and rows that have dates outside the allowed range are stored in a partition called `__UNPARTITIONED__`.

Timestamp partitioned tables have better performance than sharded tables, which use separate tables for each subset of data rather than separate partitions.

Integer range partition tables are partitioned based on a column with an `INTEGER` data type. Integer range partitions are defined with the following:

- An `INTEGER` column
- Start value of the range of partitions
- End value of the range of partitions
- Interval of each range within a partition

The total number of partitions created is the difference between the end value and start value divided by the interval size. For example, if the start value is 0, the end value is 500, and the interval is 25, then there will be 20 partitions. As with timestamped partitioned tables, the `__NULL__` and `__UNPARTITIONED__` partitions are used for rows that do not fit in the other partitions.

When you create any of type of partitioned table, you can specify that any query against that table must include a filter based on the partition structure.

Clustering

In BigQuery, *clustering* is the ordering of data in its stored format. When a table is clustered, one to four columns are specified along with an ordering of those columns. By choosing an optimal combination of columns, you can have data that is frequently accessed together collocated in storage. Clustering is supported only on partitioned tables, and it is used when filters or aggregations are frequently used.

Once a clustered table is created, the clustering columns cannot be changed. Clustering columns must be one of the following data types:

- Date
- Bool
- Geography
- INT64
- Numeric
- String
- Timestamp

Both partitioning and clustering can significantly improve the efficiency of filtering and aggregation queries.

QUERYING DATA IN BIGQUERY

BigQuery supports two types of queries: interactive and batch queries. *Interactive queries* are executed immediately, whereas *batch queries* are queued up and run when resources are available. BigQuery runs queries interactively by default.

The advantage of using batch queries is that resources are drawn from a shared resource pool and batch queries do not count toward the concurrent rate limit, which is 100 concurrent queries. *Dry-run queries*, which

just estimate the amount of data scanned, also do not count against that limit. Queries can run up to six hours but not longer.

Queries are run as jobs, similar to jobs run to load and export data. To run queries, users will need the `bigquery.jobs.create` permission, which can be granted by assigning one of the `bigquery.user`, `bigquery.jobUser`, or `bigquery.admin` roles.

Like Cloud Spanner, BigQuery supports the use of parameterized queries. BigQuery uses a similar syntax, too. A parameter is specified by an `@` character followed by the name of the parameter.

It is possible to query multiple tables at once in BigQuery by using a wildcard in a table name within the `FROM` clause of a SQL query. The wildcard character is the `*` character. For example, consider a dataset that includes the following tables:

- `myproject.mydataset.dwtable1`
- `myproject.mydataset.dwtable2`
- `myproject.mydataset.dwtable3`
- `myproject.mydataset.dwtable4`
- `myproject.mydataset.dwtable5`

A single query can take into account all of these tables by using the `FROM` clause:

```
FROM `myproject.mydataset.dwtable*`
```

Wildcard queries work only on tables and not views. External tables are not supported either.

EXTERNAL DATA ACCESS

BigQuery can access data in external sources, known as *federated sources*. Instead of first loading data into BigQuery, you can create a reference to an external source. External sources can be Cloud Bigtable, Cloud Storage, and Google Drive.

When accessing external data, you can create either permanent or temporary external tables. Permanent tables are those that are created in a dataset and linked to an external source. Dataset-level access controls can be applied to these tables. When you are using a temporary table, a table is created in a special dataset and will be available for approximately 24 hours. Temporary tables are useful for one-time operations, such as loading data into a data warehouse.

The following permissions are required to query an external table in BigQuery:

- `bigquery.tables.create`
- `bigquery.tables.getdata`
- `bigquery.jobs.create`

Querying Cloud Bigtable Data from BigQuery

To query data stored in Cloud Bigtable, you will first need to specify a URI for the table you want to query. The URI includes the project ID of the project containing the Cloud Bigtable instance, the instance ID, and the table names. When working with external Cloud Bigtable data, users must have the following roles:

- `bigquery.dataViewer` at the dataset level or higher
- `bigquery.user` role at the project level or higher to run query jobs
- `bigtable.reader` role, which provides read-only access to table metadata

Performance of reading from Cloud Bigtable tables will depend on the number of rows read, the amount of data read, and the level of parallelization.

When an external Cloud Bigtable has column families, the column families are represented within BigQuery as an array of columns.

Querying Cloud Storage Data from BigQuery

BigQuery supports several formats of Cloud Storage data:

- Comma-separated values
- Newline-delimited JSON
- Avro
- Optimized Row Columnar (ORC)
- Parquet
- Datastore exports
- Firestore exports

Data files may be in Regional, Multi-Regional, Nearline, or Coldline storage.

When you create a reference to a Cloud Storage data source, you specify a URI that identifies a bucket name and a filename.

To query data from Cloud Storage, users must have the `storage.objects.get` permission. If they are querying using wildcards

in the table name, they will also need the `storage.objects.list` permission.

Querying Google Drive Data from BigQuery

Data from Google Drive can be queried from the following formats:

- Comma-separated values
- Newline-delimited JSON
- Avro
- Google Sheets

Data sources are referenced using a URI that includes the Google Drive file ID of the data file. An OAuth scope is required to register and query an external data source in Google Drive. Users must be granted View access to the Google Drive file that is the source of external data.

BIGQUERY ML

BigQuery extends standard SQL with the addition of machine learning (ML) functionality. This allows BigQuery users to build ML models in BigQuery rather than programming models in Python, R, Java, or other programming languages outside of BigQuery.

Currently, BigQuery supports several ML algorithms, including the following:

- Linear regression
- Binary logistic regression
- Multiclass logistic regression
- K-means clustering
- TensorFlow models

The basic steps for using BigQuery ML are as follows:

1. Create a dataset to store the model.
2. Examine and preprocess data to map it to a form suitable for use with an ML algorithm.
3. Divide your dataset into training, validation, and testing datasets.
4. Create a model using training data and the `CREATE MODEL` command.
5. Evaluate the model's precision, recall, accuracy, and other properties with the `ML.EVALUATE` function.
6. When the model is sufficiently trained, use the `ML.PREDICT` function to apply the model to make decisions.

There is much more to machine learning than this brief summary of BigQuery ML. For additional details on preparing data, building models, and evaluating them, see Chapter 9, "Deploying Machine Learning Pipelines," and Chapter 11, "Measuring, Monitoring, and Troubleshooting Machine Learning Models."

Exam Essentials

Understand Cloud Bigtable is a nonrelational database based on a sparse three-dimensional map. The three dimensions are rows, columns, and cells. When you create a Cloud Bigtable instance, you specify a number of type of nodes. These nodes manage metadata about the data stored in the Cloud Bigtable database, whereas the actual data is stored outside of the nodes on the Colossus filesystem. Within the Colossus filesystem, data is organized into sorted string tables, or SSTables, which are called *tablets*.

Understand how to design row-keys in Cloud Bigtable. In general, it is best to avoid monotonically increasing values or lexicographically close strings at the beginning of keys. When a using a multitenant Cloud Bigtable database, it is a good practice to use a tenant prefix in the row-key. String identifiers, such as a customer ID or a sensor ID, are good candidates for a row-key. Timestamps may be used as part of a row-key, but they should not be the entire row-key or the start of the row-key. Moving timestamps from the front of a row-key so that another attribute is the first part of the row-key is an example of *field promotion*. In general, it is a good practice to promote, or move toward the front of the key, values that are highly varied. Another way to avoid hotspots is to use salting.

Know how to use tall and narrow tables for time-series databases. Keep names short; this reduces the size of metadata since names are stored along with data values. Store few events within each row, ideally only one event per row; this makes querying easier. Also, storing multiple events increases the chance of exceeding maximum recommended row sizes. Design row-keys for looking up a single value or a range of values. Range scans are common in time-series analysis. Keep in mind that there is only one index on Cloud Bigtable tables.

Know when to use interleaved tables in Cloud Spanner. Use interleaved tables with a parent-child relationship in which parent data is stored with child data. This makes retrieving data from both tables simultaneously more efficient than if the data were stored separately and is especially helpful when performing joins. Since the data from both tables is co-located, the database has to perform fewer seeks to get all the needed data.

Know how to avoid hotspots by designing primary keys properly. Monotonically increasing keys can cause read and write op-

erations to happen in few servers simultaneously instead of being evenly distributed across all servers. Options for keys include using the hash of a natural key; swapping the order of columns in keys to promote higher-cardinality attributes; using a universally unique identifier (UUID), specifically version 4 or later; and using bit-reverse sequential values.

Know the differences between primary and secondary indexes. Primary indexes are created automatically on the primary key. Secondary indexes are explicitly created using the `CREATE INDEX` command. Secondary indexes are useful when filtering in a query using a `WHERE` clause. If the column referenced in the `WHERE` clause is indexed, the index can be used for filtering rather than scanning the full table and then filtering. Secondary indexes are also useful when you need to return rows in a sort order other than the primary key order. When a secondary index is created, the index will store all primary key columns from the base table, all columns included in the index, and any additional columns specified in a `STORING` clause.

Understand the organizational structure of BigQuery databases. Projects are the high-level structure used to organize the use of GCP services and resources. Datasets exist within a project and are containers for tables and views. Access to tables and views are defined at the dataset level. Tables are collections of rows and columns stored in a columnar format, known as *Capacitor format*, which is designed to support compression and execution optimizations.

Understand how to denormalize data in BigQuery using nested and repeated fields. Denormalizing in BigQuery can be done with nested and repeated columns. A column that contains nested and repeated data is defined as a `RECORD` datatype and is accessed as a `STRUCT` in SQL. BigQuery supports up to 15 levels of nested `STRUCT`s.

Know when and why to use partitioning and clustering in BigQuery. Partitioning is the process of dividing tables into segments called *partitions*. BigQuery has three partition types: ingestion time partitioned tables, timestamp partitioned tables, and integer range partitioned tables. In BigQuery, clustering is the ordering of data in its stored format. Clustering is supported only on partitioned tables and is used when filters or aggregations are frequently used.

Understand the different kinds of queries in BigQuery. BigQuery supports two types of queries: interactive and batch queries. Interactive queries are executed immediately, whereas batch queries are queued and run when resources are available. The advantage of using these batch queries is that resources are drawn from a shared resource pool and batch queries do not count toward the concurrent rate limit, which is 100 concurrent queries. Queries are run as jobs, similar to jobs run to load and export data.

Know that BigQuery can access external data without you having to import it into BigQuery first. BigQuery can access data in external sources, known as federated sources. Instead of first loading data into BigQuery, you can create a reference to an external source. External sources can be Cloud Bigtable, Cloud Storage, and Google Drive. When accessing external data, you can create either permanent or temporary external tables. Permanent tables are those created in a dataset and linked to an external source. Temporary tables are useful for one-time operations, such as loading data into a data warehouse.

Know that BigQuery ML supports machine learning in BigQuery using SQL. BigQuery extends standard SQL with the addition of machine learning functionality. This allows BigQuery users to build machine learning models in BigQuery rather than programming models in Python, R, Java, or other programming languages outside of BigQuery.

Review Questions

You can find the answers in the appendix.

1. You are investigating long latencies in Cloud Bigtable query response times. Most queries finish in less than 20 ms, but the 99th percentile queries can take up to 400 ms. You examine a Key Visualizer heatmap and see two areas with bright colors indicating hotspots. What could be causing those hotspots?

   1. Improperly used secondary index
   2. Less than optimal partition key
   3. Improperly designed row-key
   4. Failure to use a read replica

2. An IoT startup has hired you to review their Cloud Bigtable design. The database stores data generated by over 100,000 sensors that send data every 60 seconds. Each row contains all the data for one sensor sent during an hour. Hours always start at the top of the hour. The row-key is the sensor ID concatenated to the hour of the day followed by the date. What change, if any, would you recommend to this design?

   1. Use one row per sensor and 60-second datasets instead of storing multiple datasets in a single row.
   2. Start the row keyrow-key with the day and hour instead of the sensor ID.

3. Allow hours to start an any arbitrary time to accommodate differences in sensor clocks.
4. No change is recommended.

3. Your company has a Cloud Bigtable database that requires strong consistency, but it also requires high availability. You have implemented Cloud Bigtable replication and specified single-cluster routing in the app profile for the database. Some users have noted that they occasionally receive query results inconsistent with what they should have received. The problem seems to correct itself within a minute. What could be the cause of this problem?

1. Secondary indexes are being updated during the query and return incorrect results when a secondary index is not fully updated.
2. You have not specified an app configuration file that includes single-cluster routing and use of replicas only for failover.
3. Tablets are being moved between nodes, which can cause inconsistent query results.
4. The row-key is not properly designed.

4. You have been tasked with migrating a MongoDB database to Cloud Spanner. MongoDB is a document database, similar to Cloud Firestore. You would like to maintain some of the document organization of the MongoDB design. What data type, available in Cloud Spanner, would you use to define a column that can hold a document-like structure?

1. Array
2. String
3. `STRUCT`
4. JSON

5. An application using a Cloud Spanner database has several queries that are taking longer to execute than the users would like. You review the queries and notice that they all involve joining three or more tables that are all related hierarchically. What feature of Cloud Spanner would you try in order to improve the query performance?

1. Replicated clusters
2. Interleaved tables
3. `STORING` clause
4. Execution plans

6. A Cloud Spanner database is using a natural key as the primary key for a large table. The natural key is the preferred key by users because the values are easy to relate to other data. Database administrators notice that these keys are causing hotspots on Cloud Spanner nodes and are adversely affecting performance. What would you recommend in order to improve performance?

1. Keep the data of the natural key in the table but use a hash of the natural key as the primary key
2. Keep the natural key and let Cloud Spanner create more splits to improve performance
3. Use interleaved tables
4. Use more secondary indexes

7. You are using a UUID as the primary key in a Cloud Spanner database. You have noticed hotspotting that you did not anticipate. What could be the cause?

1. You have too many secondary indexes.
2. You have too few secondary indexes.
3. You are using a type of UUID that has sequentially ordered strings at the beginning of the UUID.
4. You need to make the maximum length of the primary key longer.

8. You are working for a financial services firm on a Cloud Bigtable database. The database stores equity and bond trading information from approximately 950 customers. Over 10,000 equities and bonds are tracked in the database. New data is received at a rate of 5,000 data points per minute. What general design pattern would you recommend?

1. Tall and narrow table
2. One table for each customer
3. One table for equities and one for bonds
4. Option A and Option B
5. Option A and Option C

9. You have been brought into a large enterprise to help with a data warehousing initiative. The first project of the initiative is to build a repository for all customer-related data, including sales, finance, inventory, and

logistics. It has not yet been determined how the data will be used. What Google Cloud storage system would you recommend that the enterprise use to store that data?

1. Cloud Bigtable
2. BigQuery
3. Cloud Spanner
4. Cloud Storage

10. Data is streaming into a BigQuery table. As the data arrives, it is added to a partition that was automatically created that day. Data that arrives the next day will be written to a different partition. The data modeler did not specify a column to use as a partition key. What kind of partition is being used?

1. Ingestion time partitioned tables
2. Timestamp partitioned tables
3. Integer range partitioned tables
4. Clustered tables

11. You are designing a BigQuery database with multiple tables in a single dataset. The data stored in the dataset is measurement data from sensors on vehicles in the company's fleet. Data is collected on each vehicle and downloaded at the end of each shift. After that, it is loaded into a partitioned table. You want to have efficient access to the most interesting data, which you define as a particular measurement having a value greater than 100.00. You want to cluster on that measurement column, which is a FLOAT64. When you define the table with a timestamped partitioned table and clustering on the measurement column, you receive an error. What could that error be?

1. You cannot use clustering on an external table.
2. You cannot use clustering with a FLOAT64 column as the clustering key.
3. The table is not the FLOAT64 partition type.
4. The clustering key must be an integer or timestamp.

12. What data formats are supported for external tables in Cloud Storage and Google Drive?

1. Comma-separated values only
2. Comma-separated values and Avro
3. Comma-separated values, Avro, and newline-delimited JSON
4. Comma-separated values, Avro, newline-delimited JSON, and Parquet