Start Lab 02:15:00

Designing and Querying Bigtable Schemas

Lab 1 hour 30 minutes No cost Introductory

GSP1053

Overview

Bigtable is Google's fully managed, scalable NoSQL database service. Bigtable is ideal for storing large amounts of data in a key-value store and for use cases such as personalization, ad tech, financial tech, digital media, and Internet of Things (IoT). Bigtable supports high read and write throughput at low latency for fast access to large amounts of data for processing and analytics.

In Bigtable, each row represents a single entity (such as an individual user or sensor) and is labeled with a unique row key. Each column stores attribute values for each row, and column families can be used to organize related columns. At the intersection of a row and column, there can be multiple cells, with each cell representing a different version of the data at a given timestamp.

In this lab, you use the Bigtable page of the Google Cloud Console to explore a Bigtable instance and the Bigtable CLI (`cbt` CLI) to query data in a Bigtable table. You also design a table schema and row key using best practices for Bigtable.

What you'll do

Overview

Bigtable is Google's fully managed, scalable NoSQL database service. Bigtable is ideal for storing large amounts of data in a key-value store and for use cases such as personalization, ad tech, financial tech, digital media, and Internet of Things (IoT). Bigtable supports high read and write throughput at low latency for fast access to large amounts of data for processing and analytics.

In Bigtable, each row represents a single entity (such as an individual user or sensor) and is labeled with a unique row key. Each column stores attribute values for each row, and column families can be used to organize related columns. At the intersection of a row and column, there can be multiple cells, with each cell representing a different version of the data at a given timestamp.

In this lab, you use the Bigtable page of the Google Cloud Console to explore a Bigtable instance and the Bigtable CLI (`cbt` CLI) to query data in a Bigtable table. You also design a table schema and row key using best practices for Bigtable.

What you'll do

Overview

Bigtable is Google's fully managed, scalable NoSQL database service. Bigtable is ideal for storing large amounts of data in a key-value store and for use cases such as personalization, ad tech, financial tech, digital media, and Internet of Things (IoT). Bigtable supports high read and write throughput at low latency for fast access to large amounts of data for processing and analytics.

In Bigtable, each row represents a single entity (such as an individual user or sensor) and is labeled with a unique row key. Each column stores attribute values for each row, and column families can be used to organize related columns. At the intersection of a row and column, there can be multiple cells, with each cell representing a different version of the data at a given timestamp.

In this lab, you use the Bigtable page of the Google Cloud Console to explore a Bigtable instance and the Bigtable CLI (`cbt` CLI) to query data in a Bigtable table. You also design a table schema and row key using best practices for Bigtable.

What you'll do

Overview

Bigtable is Google's fully managed, scalable NoSQL database service. Bigtable is ideal for storing large amounts of data in a key-value store and for use cases such as personalization, ad tech, financial tech, digital media, and Internet of Things (IoT). Bigtable supports high read and write throughput at low latency for fast access to large amounts of data for processing and analytics.

In Bigtable, each row represents a single entity (such as an individual user or sensor) and is labeled with a unique row key. Each column stores attribute values for each row, and column families can be used to organize related columns. At the intersection of a row and column, there can be multiple cells, with each cell representing a different version of the data at a given timestamp.

In this lab, you use the Bigtable page of the Google Cloud Console to explore a Bigtable instance and the Bigtable CLI (`cbt` CLI) to query data in a Bigtable table. You also design a table schema and row key using best practices for Bigtable.

What you'll do

Overview

Bigtable is Google's fully managed, scalable NoSQL database service. Bigtable is ideal for storing large amounts of data in a key-value store and for use cases such as personalization, ad tech, financial tech, digital media, and Internet of Things (IoT). Bigtable supports high read and write throughput at low latency for fast access to large amounts of data for processing and analytics.

In Bigtable, each row represents a single entity (such as an individual user or sensor) and is labeled with a unique row key. Each column stores attribute values for each row, and column families can be used to organize related columns. At the intersection of a row and column, there can be multiple cells, with each cell representing a different version of the data at a given timestamp.

In this lab, you use the Bigtable page of the Google Cloud Console to explore a Bigtable instance and the Bigtable CLI (`cbt` CLI) to query data in a Bigtable table. You also design a table schema and row key using best practices for Bigtable.

What you'll do

Overview

Overview

Bigtable is Google's fully managed, scalable NoSQL database service. Bigtable is ideal for storing large amounts of data in a key-value store and for use cases such as personalization, ad tech, financial tech, digital media, and Internet of Things (IoT). Bigtable supports high read and write throughput at low latency for fast access to large amounts of data for processing and analytics.

In Bigtable, each row represents a single entity (such as an individual user or sensor) and is labeled with a unique row key. Each column stores attribute values for each row, and column families can be used to organize related columns. At the intersection of a row and column, there can be multiple cells, with each cell representing a different version of the data at a given timestamp.

In this lab, you use the Bigtable page of the Google Cloud Console to explore a Bigtable instance and the Bigtable CLI (`cbt` CLI) to query data in a Bigtable table. You also design a table schema and row key using best practices for Bigtable.

What you'll do

Overview

Bigtable is Google's fully managed, scalable NoSQL database service. Bigtable is ideal for storing large amounts of data in a key-value store and for use cases such as personalization, ad tech, financial tech, digital media, and Internet of Things (IoT). Bigtable supports high read and write throughput at low latency for fast access to large amounts of data for processing and analytics.

In Bigtable, each row represents a single entity (such as an individual user or sensor) and is labeled with a unique row key. Each column stores attribute values for each row, and column families can be used to organize related columns. At the intersection of a row and column, there can be multiple cells, with each cell representing a different version of the data at a given timestamp.

In this lab, you use the Bigtable page of the Google Cloud Console to explore a Bigtable instance and the Bigtable CLI (`cbt` CLI) to query data in a Bigtable table. You also design a table schema and row key using best practices for Bigtable.

What you'll do


```
[core]
project = qwiklabs-gcp-44776a13dea667a6
```

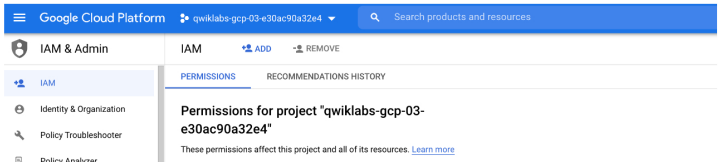
Note: Full documentation of **gcloud** is available in the gcloud CLI overview guide.

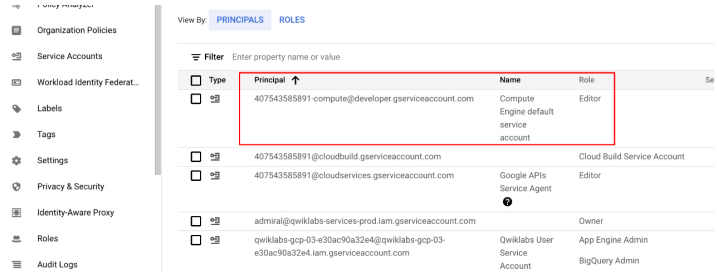
Check project permissions

Before you begin working on Google Cloud, you must ensure that your project has the correct permissions within Identity and Access Management (IAM).

1. In the Google Cloud console, on the **Navigation menu** (☰), click **IAM & Admin** > **IAM**.

2. Confirm that the default compute Service Account `{project-number}-compute@developer.gserviceaccount.com` is present and has the `editor` role assigned. The account prefix is the project number, which you can find on **Navigation menu** > **Cloud overview**.





If the account is not present in IAM or does not have the `editor` role, follow the steps below to assign the required role.

1. In the Google Cloud console, on the **Navigation menu**, click **Cloud overview**.
2. From the **Project info** card, copy the **Project number**.
3. On the **Navigation menu**, click **IAM & Admin** > **IAM**.
4. At the top of the **IAM** page, click **Add**.
5. For **New principals**, type:

```
{project-number}-compute@developer.gserviceaccount.com
```

Replace `{project-number}` with your project number.

6. For **Select a role**, select **Basic** (or Project) > **Editor**.
7. Click **Save**.

Task 1. Explore a Bigtable instance in the Console

For this lab exercise, a Bigtable instance and table have been pre-created for you to explore. In this task, you access the Bigtable instance named *personalized-sales* in the Google Cloud Console and review relevant details about the instance.

1. In the Google Cloud Console, in the **Navigation menu** (☰), under **Databases**, click **Bigtable**.
2. From the list of Bigtable instances, identify the instance ID named **personalized-sales**.

   Review the details provided for **Nodes**, and answer the following question.

   What is the total number of nodes allocated to clusters in the instance?

   - ○ 2
   - ○ 1
   - ○ 10
   - ○ 0

   Submit

3. To navigate to the instance details, click on the instance ID named **personalized-sales**.

4. Click **Edit instance**.

   Review the details for the instance, and answer the following questions.

   What is the storage type for the instance?

   - SSD
   - HDFS
   - Cloud Storage
   - HDD

   Submit

   How many cluster IDs are contained in the instance?

   - 2
   - 0
   - 1
   - 3

   Submit

   What is the name of the cluster ID for the instance?

   - ps-cluster1
   - personalized-sales-cluster1
   - personalized-sales-c1
   - personalized-sales-cluster-1

   Submit

5. To close the instance edit page, click **Cancel**.

6. From the table of cluster IDs, on the line for **personalized-sales-cluster1**, click **Edit** pencil icon.

   Review the details of the cluster, and answer the following questions.

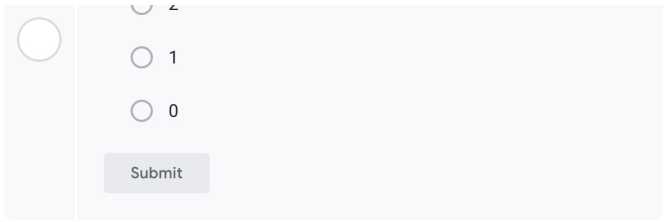
   What is the node scaling mode for the cluster?

   - Automatic scaling
   - Manual selection
   - Manual allocation
   - Autoscaling

   Submit

   How many nodes are allocated in the cluster?

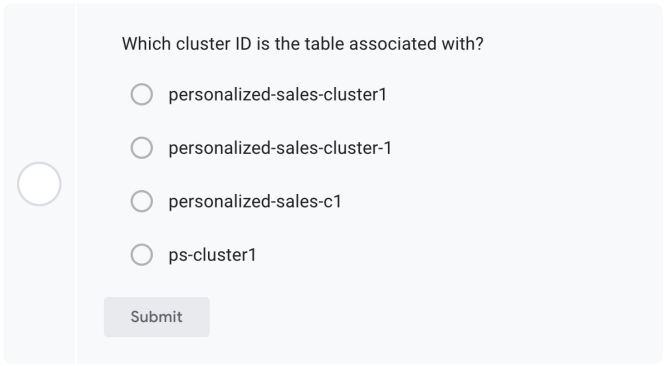
   - 0.5
   - 2



7. To close the cluster details, click **Cancel**.

8. In the navigation menu, under **Instance**, click **Tables**.

   Review the details for the table named *UserSessions*, and answer the following question.



Now that you have reviewed the details of the Bigtable instance, you can proceed to the next task to connect to the instance using the `cbt` CLI.

Task 2. Configure the Bigtable CLI

To connect to Bigtable using `cbt` CLI commands, you first need to update the `.cbtrc` configuration file with your project ID and your Bigtable instance ID using Cloud Shell.

For a review of how to access Cloud Shell, see the **Setup and requirements** section earlier in this lab guide.

While `cbt` CLI is primarily intended for debugging and exploration, it is a useful tool for learning the basics of Bigtable. To complete CRUD (Create, Read, Update, Delete) operations in production, we recommend using one of the client libraries for Bigtable.

1. To modify the `.cbtrc` file with the project ID, in Cloud Shell, run the following commands:

```
echo project = `gcloud config get-value project` \
    >> ~/.cbtrc
```

2. To see a list of available Bigtable instances in the project, run the following command:

```
cbt listinstances
```

The output confirms that there is one instance named *personalized-sales*.

3. To modify the `.cbtrc` file with the Bigtable instance ID, run the following commands:

```
echo instance = personalized-sales \
    >> ~/.cbtrc
```

4. To verify that you have successfully modified the `.cbtrc` file with the project ID and instance ID, run the following command:

```
cat ~/.cbtrc
```

The output should resemble the following:

```
project = <project-id>
instance = personalized-sales
```

5. To see a list of available tables in the Bigtable instance named **personalized-sales**, run the following command:

```
cbt ls
```

The output confirms that the instance already contains one table named *UserSessions*. You will work with this table in a later task.

Task 3. Design a schema and row key in Bigtable

In this task, you create a test table to explore schema and row key design principles in Bigtable.

Review raw data to help design schema

To design a schema and row key in Bigtable, it is helpful to first answer key questions about the data that will be stored.

Question	Purpose
What does an individual row represent? (for example, an individual user or sensor)	To identify the row structure
What will be the most common queries to this data?	To create a row key
What values are collected for each row?	To identify the columns (referred to as *column qualifiers*)
Are there related columns that can be grouped or organized together?	To identify the column families

For example, consider a dataset that captures online shopping sessions for all users of an ecommerce company website. Each row represents an individual online shopping session with a timestamp. The most common queries to the dataset will retrieve details about individual sessions and the associated user ID. The values stored for each shopping session are all items that the user interacted with and purchased during the session as well as a color preference for the user.

The raw data could be organized as follows, with more columns for additional products (such as blue_jacket or purple_bag):

timestamp	user_id	preferred_color	red_skirt	red_hat	orange_shoes	sale
1638940844260	1939	green		seen	seen	
1638940844260	2466	blue	seen	seen		

1638940844260	2466	blue	seen	seen		
1638940844260	1679	blue		seen		blue_blouse#blue_
1638940844260	2737	blue			seen	blue_dress#blue_ja
1638940844260	582	yellow				yellow_skirt

Note: The user id provided in this example is intended as a simple example of an identifier. In a typical application of Bigtable, you would likely generate a universally unique identifier (UUID) for each user.

Create test table

A best practice for Bigtable is to store data with similar schemas in the same table, rather than in separate tables. For example, all data for the online shopping sessions can be stored in one table for easy retrieval.

- To create an empty table named **test-sessions**, run the following command:

```
cbt createtable test-sessions
```

Create column families

In Bigtable, the best practices for columns and column families include:

- Using column qualifiers as data, so that you do not repeat the value for each row.
- Organizing related columns in the same column family.
- Choosing short but meaningful names for your column families.

Following best practices for column families, which option efficiently organizes related columns into the same column family?

- red_skirt, red_hat, and orange_shoes stored in one column family, and sale stored in another column family
- red_skirt and red_hat stored in one column family, orange_shoes stored in another column family, and sale stored in another column family
- All columns in one column family with the exception of sale
- All columns in one column family

Submit

For this dataset, the column qualifiers that store product interactions can be grouped into one column family named **Interactions**, while the column qualifier that stores purchases can be organized by itself into another column family named **Sales**. The resulting schema would be organized as follows:

...	...	...	Interactions	---	---	Sales
timestamp	user_id	preferred_color	red_skirt	red_hat	orange_shoes	sale
1638940844260	1939	green		seen	seen	
1638940844260	2466	blue	seen	seen		
1638940844260	1679	blue		seen		blue_blouse#blue_
1638940844260	2737	blue			seen	blue_dress#blue_ja

1638940844260	582	yellow				yellow_skirt

1. To add a column family named **Interactions** to the **test_sessions** table, run the following command:

```
cbt createfamily test-sessions Interactions
```

2. To add another column family named **Sales**, repeat the previous command and specify the new column family name:

```
cbt createfamily test-sessions Sales
```

3. To see a list of column families in the **test_sessions** table, run the following command:

```
cbt ls test-sessions
```

The command returns the following output:

```
Family Name     GC Policy
-----------     ---------
Interactions    <never>
Sales           <never>
```

Click **Check my progress** to verify the objective.

Create a Bigtable table.

Check my progress

Create row key

In Bigtable, a best practice is to store all information for a single entity (such as an individual online shopping session) in a single row. A related best practice is to create a row key that makes it possible to easily query and retrieve a defined range of rows.

To apply best practices for row keys in Bigtable, it is recommended that you:

- Design your row key based on the queries you will use to retrieve the data.
- Avoid row keys that start with a timestamp or sequential numeric IDs or that cause related data to not be grouped.
- Design row keys that start with a more common value (such as country) and end with a more granular value (such as city).
- Store multiple delimited values in each row key using human-readable string values (such as user ID followed by timestamp).

Following best practices for row keys, which row key design supports queries that request records for each online shopping session, including timestamp, user id, and color preference?

- start with user_id, followed by timestamp, and preferred_color
- start with timestamp, followed by preferred_color, and user_id
- start with preferred_color, followed by user_id, and timestamp
- start with user_id, followed by preferred_color, and

timestamp

Submit

Following best practices for row keys, which row key stores multiple human-readable string values separated by a delimiter?

- user_id
- timestamp
- preferred_color#user_id#timestamp
- preferred_coloruser_idtimestamp

Submit

In the previous section, timestamp, user_id, and preferred_color were not organized under a column family. Recall from the questions about the raw data that most of the queries to this dataset will retrieve details about individual sessions and the associated user ID.

To support these queries, a good row key for this table would be a combination of the user ID plus the timestamp of the session. In addition, the row key can include a prefix to label the color preference for each user, such as `green1939#1638940844260` for user ID 1939, to make it easy to retrieve all users with a specific color preference.

...	Interactions	---	---	Sales
row_key	red_skirt	red_hat	orange_shoes	sale
green1939#1638940844260		seen	seen	
blue2466#1638940844260	seen	seen		
blue1679#1638940844260		seen		blue_blouse#blue_jacket
blue2737#1638940844260			seen	blue_dress#blue_jacket
yellow582#1638940844260				yellow_skirt

1. To use a row key to add data to the **Interactions** column family, run the following command:

```
cbt set test-sessions green1939#1638940844260
Interactions:red_hat=seen
```

2. To use a row key to add data to the **Sales** column family, run the following command:

```
cbt set test-sessions blue2737#1638940844260
Sales:sale=blue_dress#blue_jacket
```

3. To see the data stored in the table, run the following command:

```
cbt read test-sessions
```

Notice that although the data for **blue2737#1638940844260** was added second, it is sorted higher in the results than **green1939#1638940844260**. The records are returned in this order because in Bigtable, rows are sorted and stored lexicographically by row key. This order is similar to alphabetical order; however, rows that begin with numbers will not be sorted as smallest to largest (such as 1, 13, 2, 25, 6, and 70).

Clean up test data

- To delete the test table, run the following command:

```
cbt deletetable test-sessions
```

Click **Check my progress** to verify the objective.

Delete a Bigtable table.

Check my progress

Task 4. Query data in Bigtable

In this task, you use the `cbt` CLI retrieve data from a pre-created, fully populated version of your test table (the existing table named *UserSessions*) and examine how the table applies best practices for designing schemas and row keys in Bigtable.

Query rows with limit

In this step, you review how the UserSessions table follows Bigtable best practices by storing all user interactions with products and purchases of products in one table that contains one row for each online shopping session.

- To see the data for the first five rows of the table, run the following command:

```
cbt read UserSessions \
    count=5
```

The output is structured as follows:

```
----------------------------------------
ROW KEY
 COLUMN_FAMILY:COLUMN_QUALIFIER       @ TIMESTAMP
    VALUE
 COLUMN_FAMILY:COLUMN_QUALIFIER       @ TIMESTAMP
    VALUE
 ...
 ----------------------------------------
ROW KEY
 COLUMN_FAMILY:COLUMN_QUALIFIER       @ TIMESTAMP
    VALUE
 COLUMN_FAMILY:COLUMN_QUALIFIER       @ TIMESTAMP
    VALUE
 ...
```

The output values should resemble the following:

```
----------------------------------------
blue0#1638940844350
 Interactions:blue_hat       @ 2022/06/08-19:47:33.864000
    "viewed details"
 Interactions:green_jacket   @ 2022/06/08-19:47:33.864000
    "seen"
 ...
 ----------------------------------------
 blue1#1638940844304
  Interactions:blue_dress    @ 2022/06/08-19:47:33.864000
    "purchased"
  Sales:sale                 @ 2022/06/08-19:47:33.864000
    "blue_dress"
```

Each row contains multiple product interactions for one user (such as blue_hat and green_jacket) including whether the user has *seen*, *viewed details*, or *purchased* the product. In addition, purchases are captured in the table in the sale column qualifier in the Sales column family.

Instead of creating one table for each interaction type, product, or sale, UserSessions follows best practices by containing all of the related user interactions and products in one table. In addition, all product interactions and purchases for an individual online shopping session are stored as one row in the table.

Query by row key

The most efficient queries in Bigtable retrieve data using one of the following:

- row key
- row key prefix
- range of rows defined by starting and ending row keys

In the next steps, you use each option in the `cbt` CLI to query the *UserSessions* table and retrieve desired records.

Information on using Bigtable client libraries to read single rows of data using row keys is available in the Bigtable documentation.

Query by row key prefix

- To see the first ten rows with a color preference of *yellow*, run the following command:

```
cbt read UserSessions \
    prefix=yellow \
    count=10
```

The output values should resemble the following:

```
----------------------------------------------
yellow991#1638940844645
  Interactions:green_skirt          @ 2022/06/08-19:47:33.864000
    "seen"
  Sales:sale                        @ 2022/06/08-19:47:33.864000
    "yellow_skirt"
```

Query by specific range of row keys

- To see all rows within a specific range of row keys, run the following command:

```
cbt read UserSessions \
    start=yellow941#1638940844381 \
    end=yellow991#1638940844645
```

The output values should resemble the following:

```
----------------------------------------------
yellow991#1638940844603
  Interactions:blue_blouse          @ 2022/06/08-19:47:33.864000
    "seen"
 Sales:sale                         @ 2022/06/08-19:47:33.864000
    "yellow_jacket#yellow_blouse"
```

The `read` command begins the range with the row key provided as the `start` value and ends the range *before* the row key provided as the `end` value. Therefore, the row key `yellow991#1638940844645` is not returned in the output.

Query by specific row key

- To see all data for a specific row key, run the following command:

```
cbt lookup UserSessions \
    yellow582#1638940844260
```

The output values should resemble the following:

```
----------------------------------------
yellow582#1638940844260
 Interactions:blue_jacket       @ 2022/06/08-19:47:33.864000
    "seen"
 Sales:sale                     @ 2022/06/08-19:47:33.864000
    "yellow_skirt"
```

Which product was purchased during the session identified as purple994#1638940844525?

- purple_jacket
- green_dress
- purple_hat
- orange_hat

Submit

Query by column qualifiers and column families

In these next steps, you retrieve data filtered by column qualifiers and column families to see how column best practices are implemented in the UserSessions table.

1. To query the first five rows that have data in the Interactions column family, run the following command:

```
cbt read UserSessions count=5 \
    columns="Interactions:.*"
```

The output values should resemble the following:

```
----------------------------------------
blue0#1638940844501
  Interactions:blue_blouse                @ 2022/06/08-19:47:33.864000
    "viewed details"
  Interactions:green_jacket               @ 2022/06/08-19:47:33.864000
    "seen"
```

2. To query the first five rows that have data in the green_jacket column qualifier in the Interactions column family, run the following command:

```
cbt read UserSessions count=5 \
    columns="Interactions:green_jacket"
```

The output values should resemble the following:

```
----------------------------------------
blue1009#1638940844380
  Interactions:green_jacket               @ 2022/06/08-21:30:08.683000
    "seen"
----------------------------------------
blue101#1638940844263
  Interactions:green_jacket               @ 2022/06/08-21:30:08.683000
```

```
  Interactions:green_jacket                    @ 2022/06/08-21:30:08.083000
    "seen"
```

3. To query the first five rows that have data in the sale column qualifier in the Sales column family, run the following command:

```
cbt read UserSessions count=5 \
    columns="Sales:sale"
```

The output values should resemble the following:

```
----------------------------------------
blue0#1638940844379
  Sales:sale                            @ 2022/06/08-19:47:33.864000
    "blue_shoes#blue_shoes"
----------------------------------------
blue1#1638940844409
  Sales:sale                            @ 2022/06/08-19:47:33.864000
    "blue_blouse"
```

What is the result if you run the previous query again by setting the columns parameter equal to Sales:.*, instead of Sales:sale?

- Same column qualifiers are returned
- An error. If this happens, check the syntax for a period after the colon (Sales:.*), and try again.
- Additional column qualifiers for the Sales column family are returned
- Nothing is returned
- Same column qualifiers but a different number of rows

Submit

Because the column family named *Sales* only has one column qualifier (*sale*), the values *"Sales:sale"* and *"Sales:.*"* for `columns` return the same columns.

Congratulations!

In this lab, you used the Google Cloud Console to explore a Bigtable instance and the Cloud Bigtable CLI (`cbt` CLI) to query data in a Bigtable table. You also designed a table schema and row key using best practices for Bigtable.

Next steps / Learn more

- Check out the lab titled Creating and Populating a Bigtable Instance.

Manual Last Updated April 25, 2024

Lab Last Tested February 16, 2023