**Lab Manual- Google Cloud BigTable**

**Prepared for**:

**Date:** 18th June 2024

**Prepared by:** Bipin Sinhaa

Document Name: Lab Manual

**Document Number** GCPLab306

**Contributor:**

Table of Contents

[1 Objective 3](#_Toc169605017)

[2 The 4 Types of NoSql Databases 3](#_Toc169605018)

[3 What is Google Cloud Big Table ? 6](#_Toc169605019)

[3.1 **Bigtable Architecture** 7](#_Toc169605020)

[3.2 **Bigtable Storage Model** 9](#_Toc169605021)

[3.3 **Bigtable Best Practices** 9](#_Toc169605022)

[4 PRE-REQUISISTE 10](#_Toc169605023)

[5 Create Google Big Table Instance 10](#_Toc169605024)

[6 Access Google Big Table Instance with CLI 12](#_Toc169605025)

[7 Data structures & schema basics 14](#_Toc169605026)

[7.1 Tables 14](#_Toc169605027)

[7.2 Column Family 14](#_Toc169605028)

[7.3 **Rows, Columns & Cells** 16](#_Toc169605029)

[7.4 Querying and accessing Data 16](#_Toc169605030)

[7.5 Retrieve Single Entry 18](#_Toc169605031)

[7.6 Start & End 19](#_Toc169605032)

[7.7 Count 20](#_Toc169605033)

[8 Cleanup 21](#_Toc169605034)

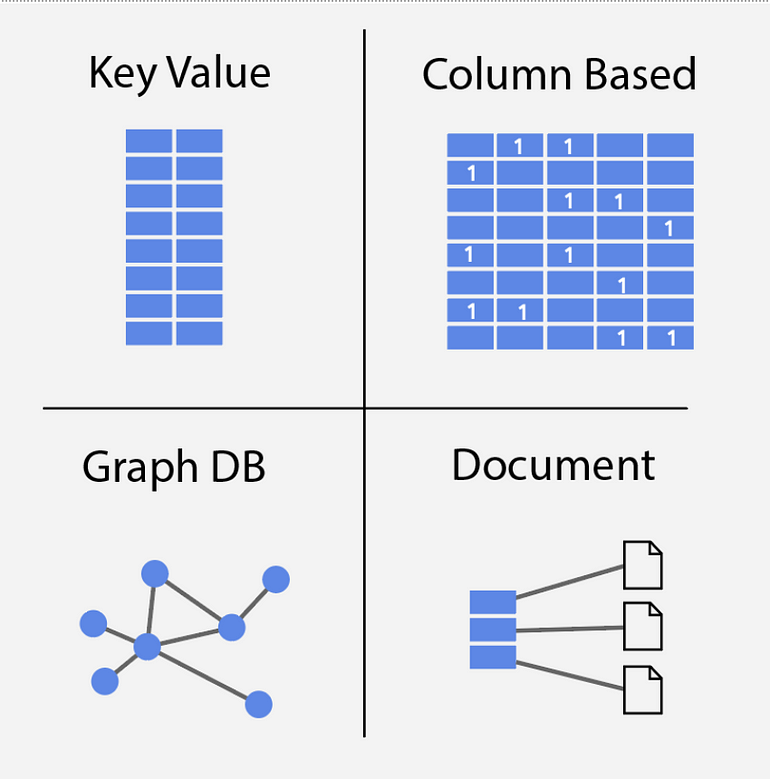
# Objective

In this tutorial we’ll walk you through your first steps with Bigtable, how to use it and what you really need to know to get started including:

* Understanding the different types of NoSQL databases
* How to setup and interact with Bigtable
* How big table structures and manages data
* Common ways to query and access data
* Best practices around schema design

# The 4 Types of NoSql Databases

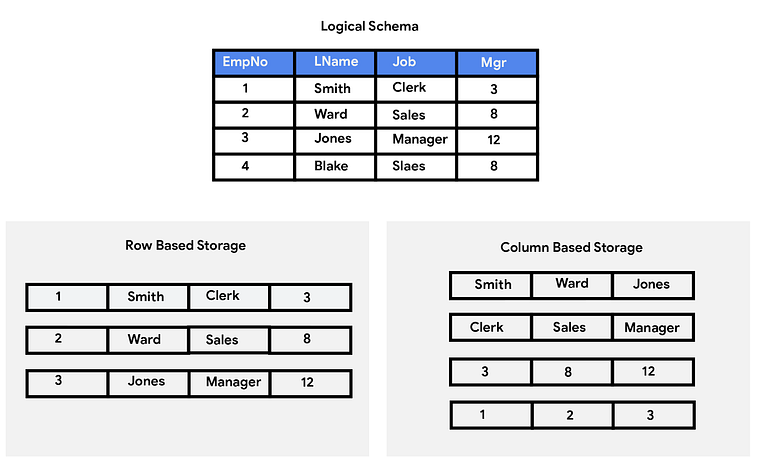
In the NoSql realm there are generally 4 types of databases. You’ve got your Column based, Document based, Key-Value based and Graph based databases.



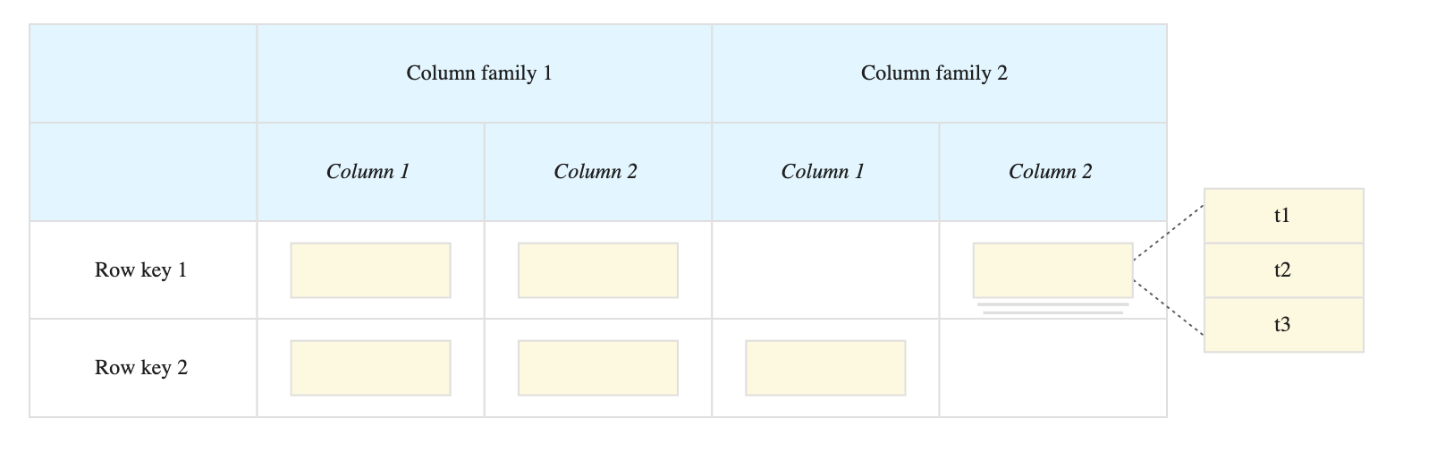
Bigtable falls into the Wide-Column based family along with others like Cassandra, and Hbase. To understand Wide-Column it helps to look first at traditional Column based. When we say Column based Nosql, what does that actually mean? Well traditional Relational databases are row based, meaning they’re optimized for returning rows of data.

**Consider a User database for example, a relational database would organize first name, last name, and address all near each other.**

If you wanted to access the **state** and **zip** of many users, the database would have to jump around to pull all the fields. A **column based database** on the other hand is optimized for accessing data by column instead of row. So in our Users database example it would store all the names together, all the states together, all the zip codes together and so on. This makes reads much more efficient. **To scan all the states, the database can stay within the same area on disk.**



A Wide-Column datastore looks similar however they often **group the columns into Column Families**, a set of columns that are typically used together. These Column Families are further optimized on disk to ensure fast access.

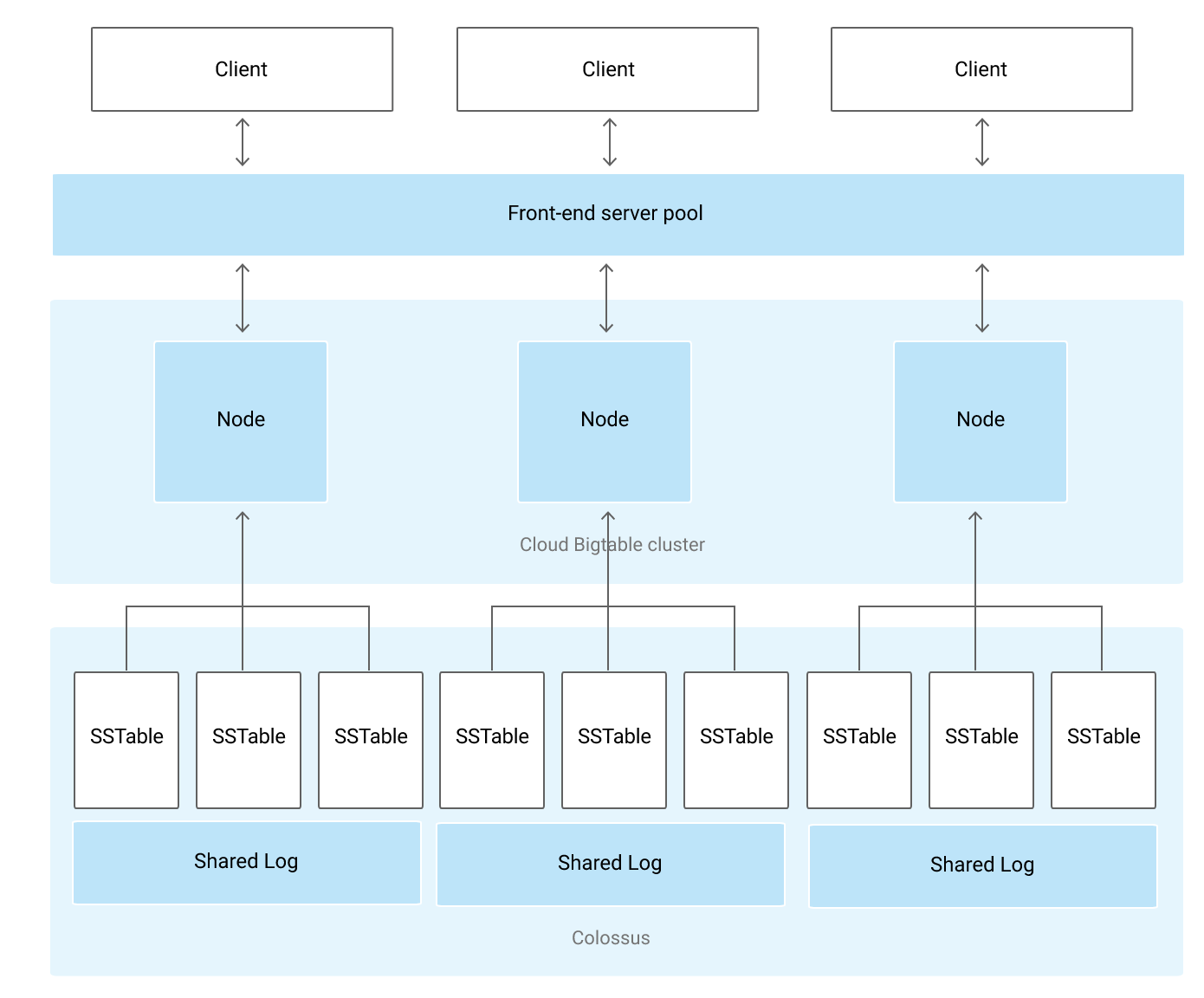


This will be good to keep in mind for later when we start designing our schemas. Alright, lets get on with it and get hands dirty

# What is Google Cloud Big Table ?

* Cloud Bigtable is a fully managed, scalable, wide-column NoSQL database service with up to 99.999% availability.
* Bigtable is ideal for applications that need very high throughput and scalability for **key/value data**, where each value is max. of 10 MB.
* Bigtable supports high read and write throughput at low latency and provides consistent sub-10ms latency – handle millions of requests/second
* Bigtable is a sparsely populated table that can scale to billions of rows and thousands of columns,
* Bigtable supports storage of **terabytes** or even **petabytes** of data
* Bigtable is not a relational database. It does not support SQL queries, joins, or multi-row transactions.
* **Fully Managed**
  + Bigtable handles upgrades and restarts transparently, and it automatically maintains high data durability.
  + Data replication can be performed by simply adding a second cluster to the instance, and replication starts automatically.
* **Scalability**
  + Bigtable scales linearly in direct proportion to the number of machines in the cluster
  + Bigtable throughput can be scaled dynamically by adding or removing cluster nodes without restarting
* Bigtable integrates easily with big data tools like **Hadoop, Dataflow, Dataproc** and supports **HBase** APIs.

## **Bigtable Architecture**

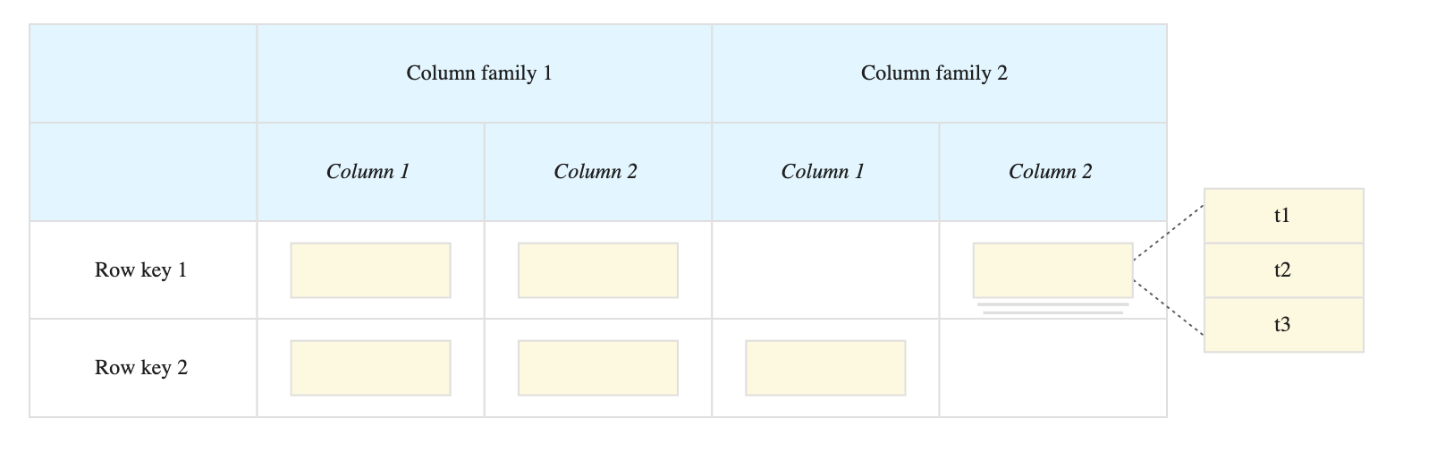


* Bigtable **Instance** is a container for **Cluster** where **Nodes** are organized.
* Bigtable stores data in Colossus, Google’s file system
* **Instance**
  + A Bigtable instance is a container for data.
  + Instances have one or more clusters, located in a different zone and different region (Different region adds to latency)
  + Each cluster has at least 1 node
  + A Table belongs to an instance and not to the cluster or node.
  + An instance also consists of the following properties
    - Storage Type – SSD or HDD
    - Application Profiles – primarily for instances using replication
* **Instance Type**
  + **Development – Single node cluster with no replication or SLA**
  + **Production – 1+ clusters which 3+ nodes per cluster**
* **Storage Type**
  + Storage Type dictates where the data is stored i.e. SSD or HDD
  + Choice of SSD or HDD storage for the instance is permanent
  + SSD storage is the most efficient and cost-effective choice for most use cases.
  + HDD storage is sometimes appropriate for very large data sets (>10 TB) that are not latency-sensitive or are infrequently accessed.
* **Application Profile**
  + An application profile, or app profile, stores settings indicate Bigtable on how to handle incoming requests from an application
  + Application profile helps define custom application-specific settings for handling incoming connections
* **Cluster**
  + Clusters handle the requests sent to a single Bigtable instance
  + Each cluster belongs to a single Bigtable instance, and an instance can have up to 4 clusters
  + Each cluster is located in a single-zone
  + Bigtable instances with only 1 cluster do not use replication
  + An Instances with multiple clusters replicate the data, which
    - improves data availability and durability
    - improves scalability by routing different types of traffic to different clusters
  + If multiple clusters within an instance, Bigtable automatically starts replicating the data by keeping separate copies of the data in each of the clusters’ zones and synchronizing updates between the copies
* **Nodes**
  + Each cluster in an instance has 1 or more nodes, which are the compute resources that Bigtable uses to manage the data.
  + Each node in the cluster handles a subset of the requests to the cluster
  + All client requests go through a front-end server before they are sent to a Bigtable node.
  + Bigtable separates the Compute from the Storage. Data is never stored in nodes themselves; each node has pointers to a set of tablets that are stored on Colossus. This helps as
    - Rebalancing tablets from one node to another is very fast, as the actual data is not copied. Only pointers for each node are updated
    - Recovery from the failure of a Bigtable node is very fast as only the metadata needs to be migrated to the replacement node.
    - When a Bigtable node fails, no data is lost.
  + A Bigtable cluster can be scaled by adding nodes which would increase
    - the number of simultaneous requests that the cluster can handle
    - the maximum throughput of the cluster.
  + Each node is responsible for:
    - Keeping track of specific tablets on disk.
    - Handling incoming reads and writes for its tablets.
    - Performing maintenance tasks on its tablets, such as periodic compactions
  + Bigtable nodes are also referred to as tablet servers

**Tables**

* Bigtable stores data in massively scalable tables, each of which is a sorted key/value map.
* A Table belongs to an instance and not to the cluster or node.
* A Bigtable table is sharded into blocks of contiguous rows, called **tablets**, to help balance the workload of queries.
* Bigtable splits all of the data in a table into separate tablets.
* Tablets are stored on the disk, separate from the nodes but in the same zone as the nodes.
* Each tablet is associated with a specific Bigtable node.
* Tablets are stored in SSTable format which provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings.
* In addition to the SSTable files, all writes are stored in Colossus’s shared log as soon as they are acknowledged by Bigtable, providing increased durability.

## **Bigtable Storage Model**



* Bigtable stores data in tables, each of which is a sorted key/value map.
* A Table is composed of rows, each of which typically describes a single entity, and columns, which contain individual values for each row.
* Each row is indexed by a single row key, and columns that are related to one another are typically grouped together into a column family.
* Each column is identified by a combination of the column family and a column qualifier, which is a unique name within the column family.
* Each row/column intersection can contain multiple cells.
* Each cell contains a unique timestamped version of the data for that row and column.
* Storing multiple cells in a column provides a record of how the stored data for that row and column has changed over time.
* Bigtable tables are sparse; if a column is not used in a particular row, it does not take up any space.

## **Bigtable Best Practices**

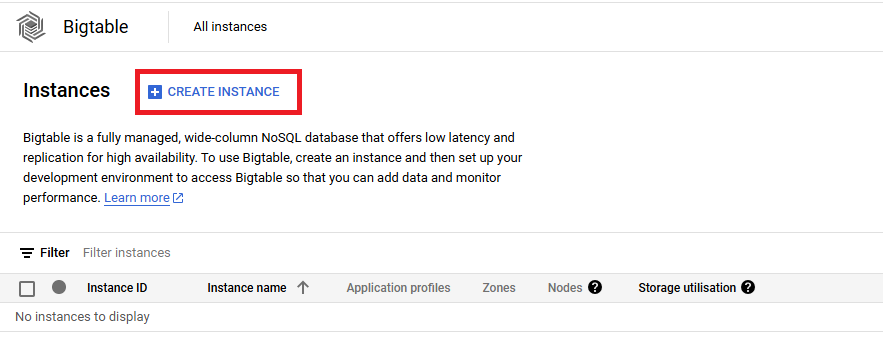
* Store datasets with similar schemas in the same table, rather than in separate tables as in SQL.
* Bigtable has a limit of 1,000 tables per instance
* Creating many small tables is a Bigtable anti-pattern
* Put related columns in the same column family
* Create up to about 100 column families per table. A higher number would lead to performance degradation.
* Choose short but meaningful names for your column families
* Put columns that have different data retention needs in different column families to limit storage cost.
* Create as many columns as you need in the table. Bigtable tables are sparse, and there is no space penalty for a column that is not used in a row
* Don’t store more than 100 MB of data in a single row as a higher number would impact performance
  + Don’t store more than 10 MB of data in a single cell.
* Design the row key based on the queries used to retrieve the data
* Following queries provide the most efficient performance
  + Row key
  + Row key prefix
  + Range of rows defined by starting and ending row keys
* Other types of queries trigger a full table scan, which is much less efficient.
* Store multiple delimited values in each row key. Multiple identifiers can be included in the row key.
* Use human-readable string values in your row keys whenever possible. Makes it easier to use the Key Visualizer tool.

# PRE-REQUISISTE

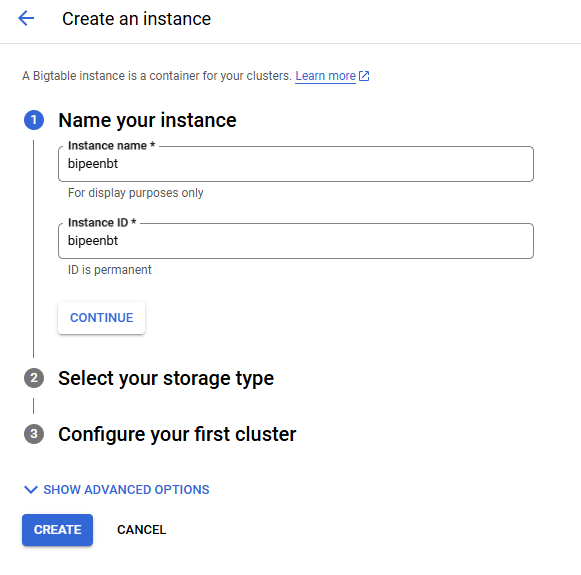
* Prior knowledge of Linux
* Accounts in GCP

# Create Google Big Table Instance

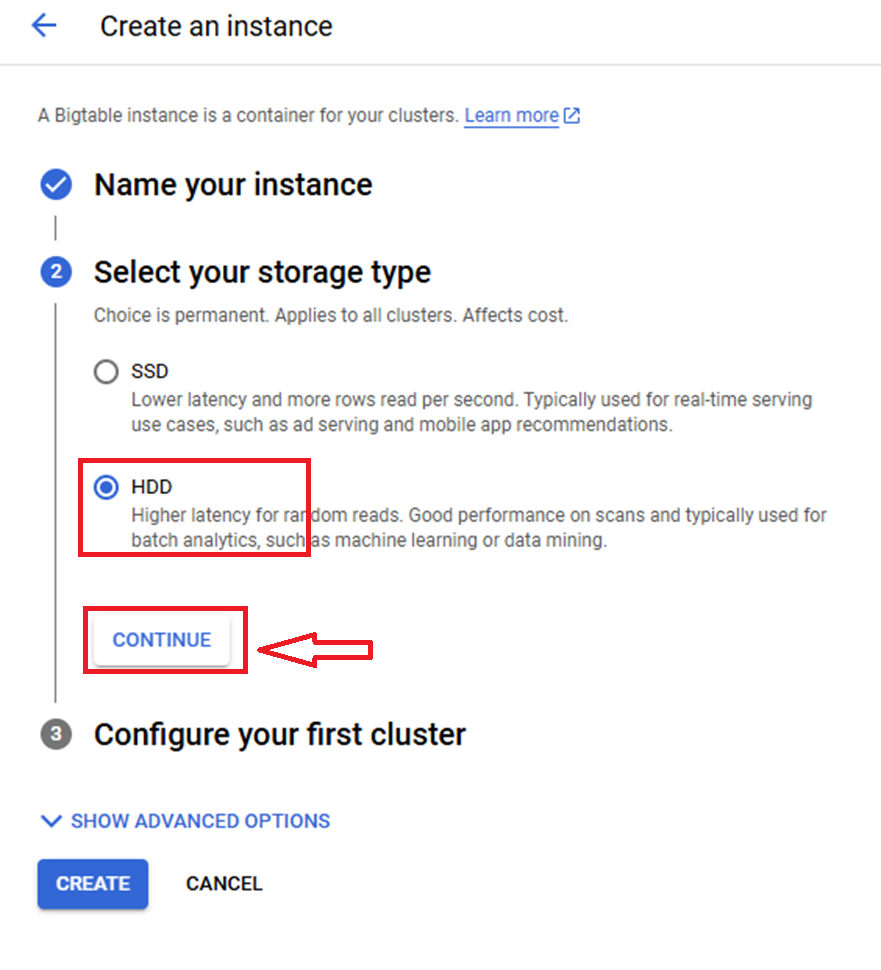
* Click Create instance .



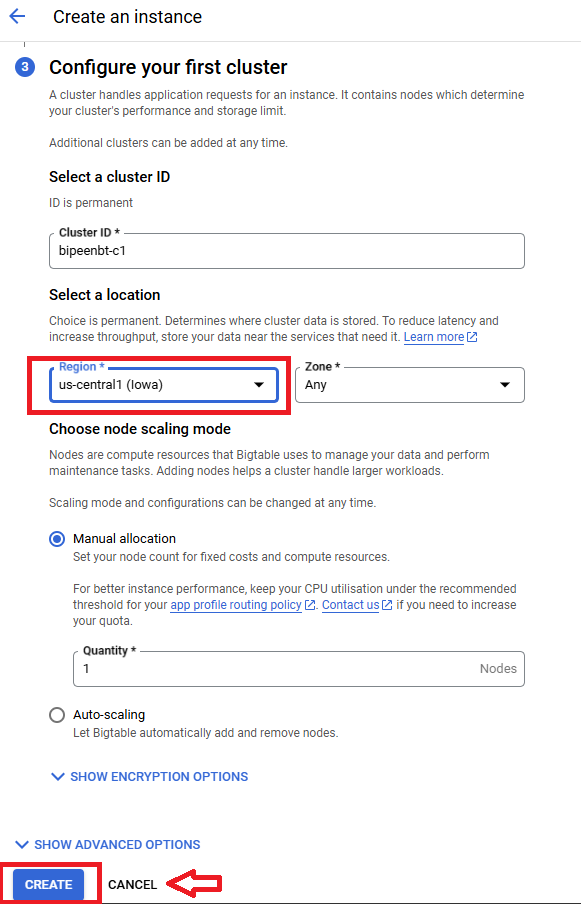
* Type the name of Instance and Click **Continue** .



* In Storage select HDD and Click **Continue** .



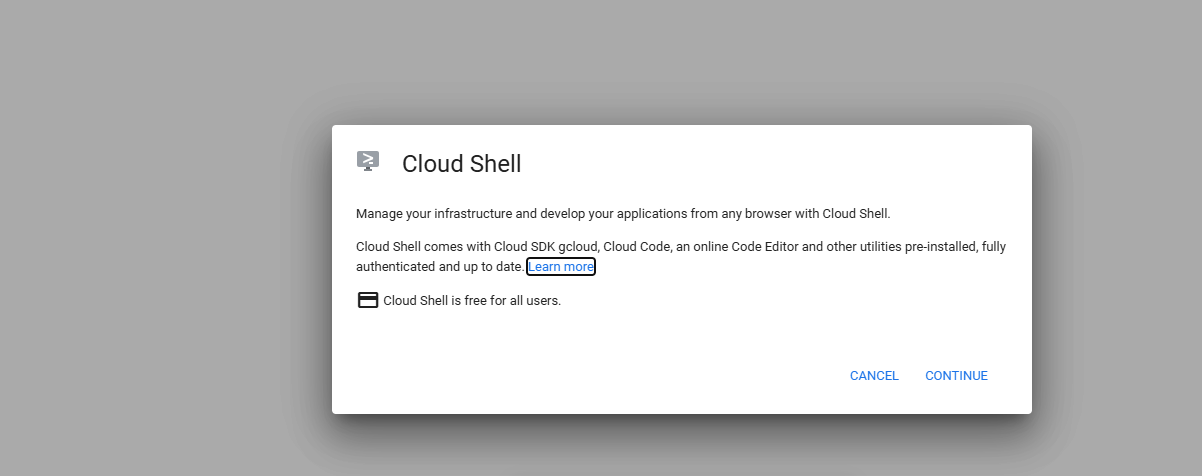
* Select region **US-Central-1** and Click **Create** .



* Wait till time it finished.

# Access Google Big Table Instance with CLI

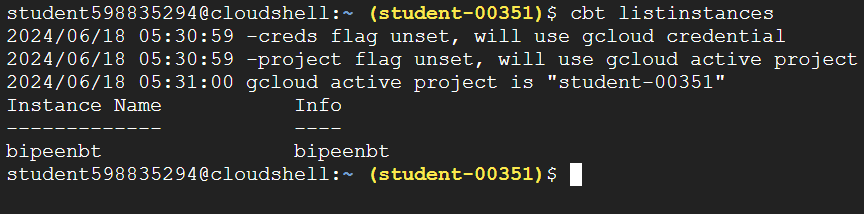
Launch cloud Shell



* list our instances

cbt listinstances

You should see the instance you just created in the output



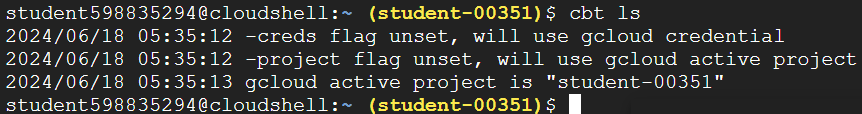
* write Instance Name to an rc file as a default

**echo instance = bipeenbt >> ~/.cbtrc**

****

* List the table

**cbt ls**

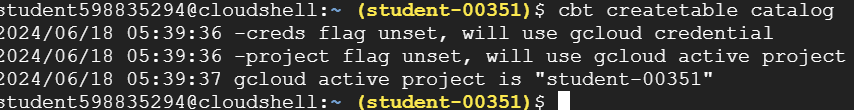
****

# Data structures & schema basics

## **Tables**

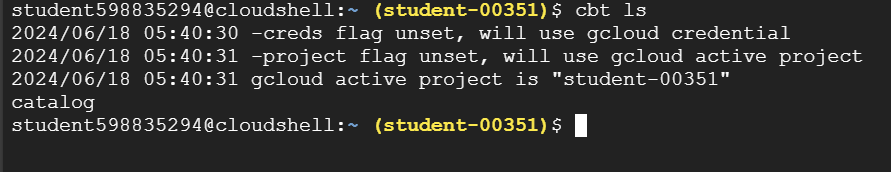
* For this example we’ll be creating a product catalog that might be used by a typical retailer. So in this step we’ll create a table called `catalog`

**cbt createtable catalog**



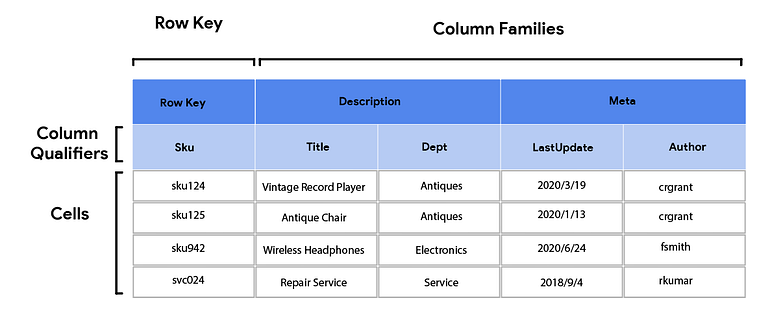
* Calling `ls` one more time and we see our table **catalogue**

**cbt ls**



## Column Family

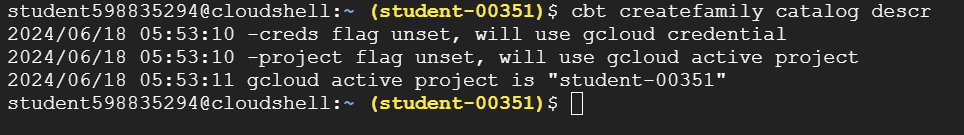
Earlier I mentioned that Bigtable stores data related to columns. To help organize the data and limit what you’re pulling back, columns are grouped into what’s called column families. These column families group the fields that are typically accessed in the same request to ensure more efficient access.



In our catalog example we may have product description fields and pricing or inventory fields. A listing of products may use data from the descriptors but not need all the store level inventory.

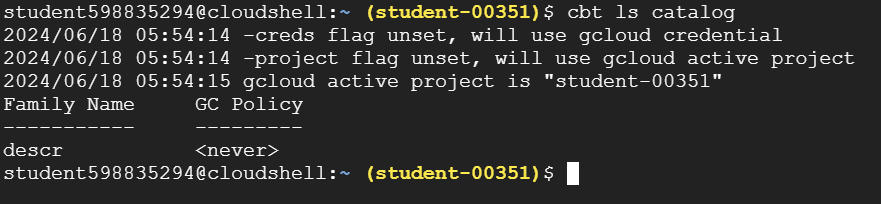
Lets go ahead and create a column family for those product descriptors

**cbt createfamily catalog descr**



And now we’ve got our column family in the table. Running the ls command again display comlomuns family **descry**

**cbt ls catalog**



## **Rows, Columns & Cells**

Just like with relational databases we have a concept of rows columns and cells. Each row is identified by a unique key you provide. Cells are at the intersection of a row id and column id To access a specific cell you need to identify the location including Row Key, Column Family, and Column Qualifier

* In our case the rowID will be a unique product sku and we’ll add a title for it in the descriptors column family. The format will be

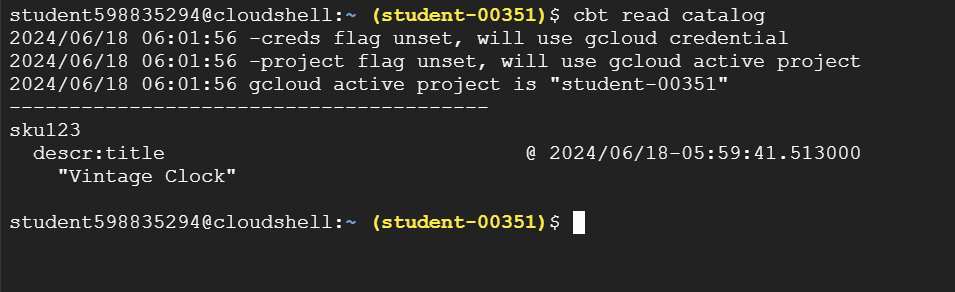
*cbt set <table> <rowID> <colFamily>:<colQualifier>=<value>*

**cbt set catalog sku123 descr:title="Vintage Clock"**



* Now if we read our catalog table , we will see the value

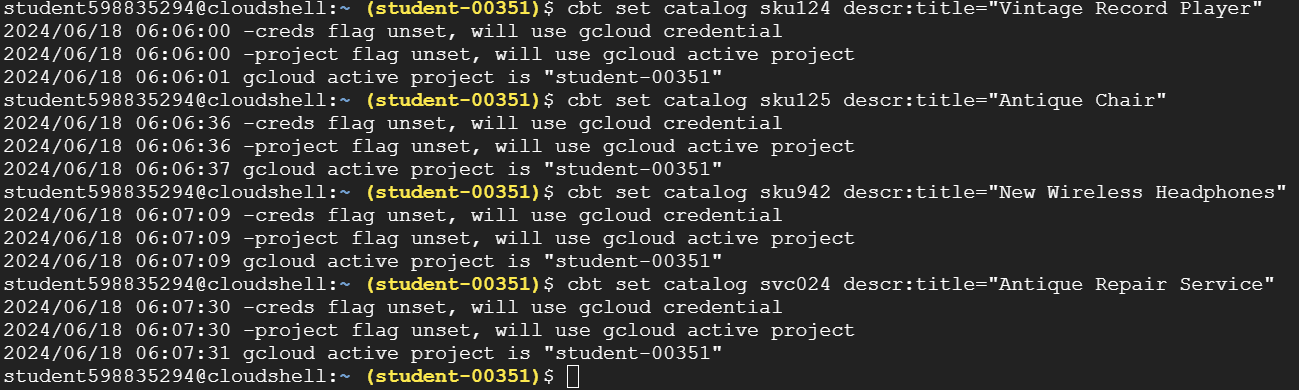
**cbt read catalog**



## Querying and accessing Data

Add some data

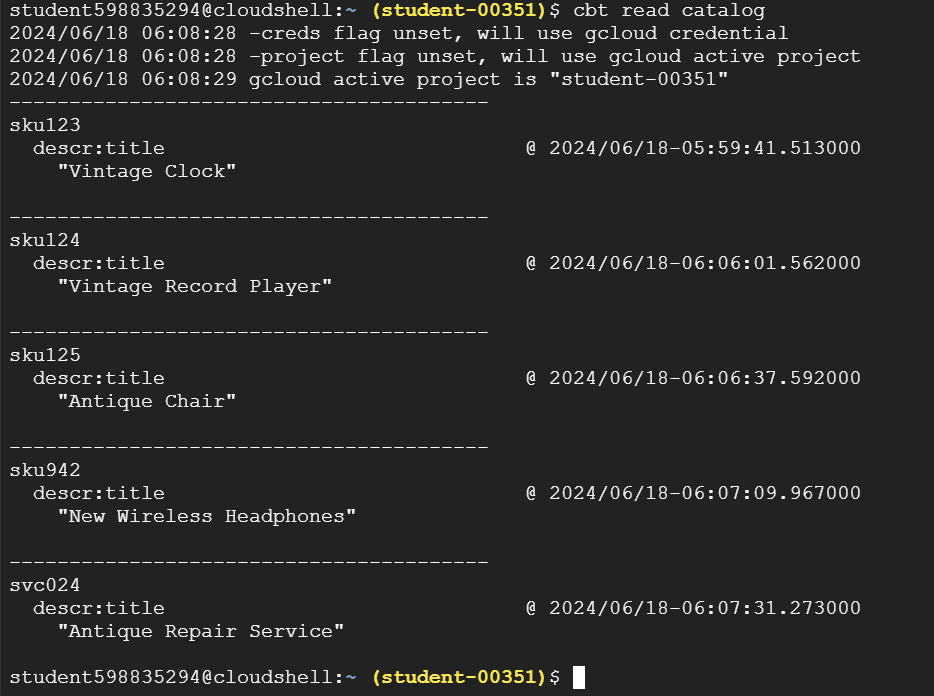
**cbt set catalog sku124 descr:title="Vintage Record Player"  
cbt set catalog sku125 descr:title="Antique Chair"  
cbt set catalog sku942 descr:title="New Wireless Headphones"  
cbt set catalog svc024 descr:title="Antique Repair Service"**

****

We’ve added 3 more skus some sequential and one in the 900s. We’ve also added the **last entry as a service** rather than a product.

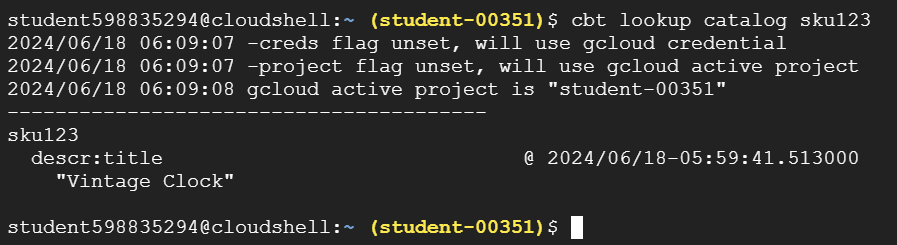
**cbt set catalog svc024 descr:title="Antique Repair Service"**

**cbt read**

****

## Retrieve Single Entry

**cbt lookup catalog sku123**

****

Additionally you can get even more specific indicating the exact columns you want

**cbt lookup catalog sku123 columns=descr:title**

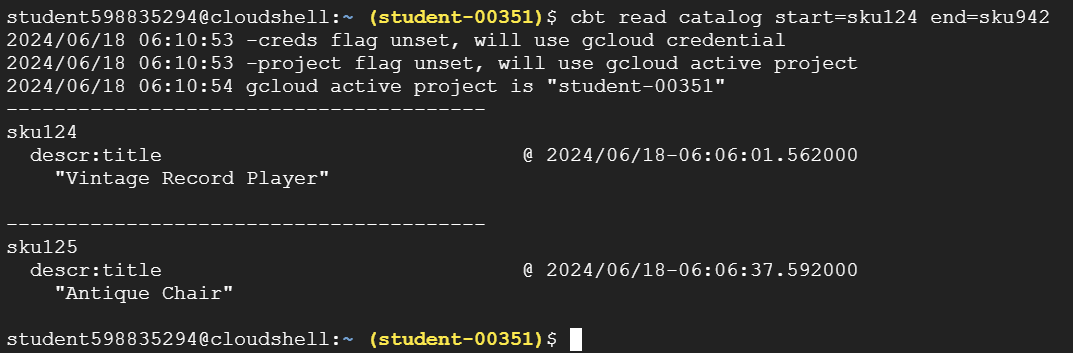
****

## Start & End

First it’s important to understand that Bigtable stores all its rows in ascending order based on the row id. Many of the features and patterns in bigtable revolve around this core concept. To see it in practice, the simplest way is to use `start` and `end` on the read command.

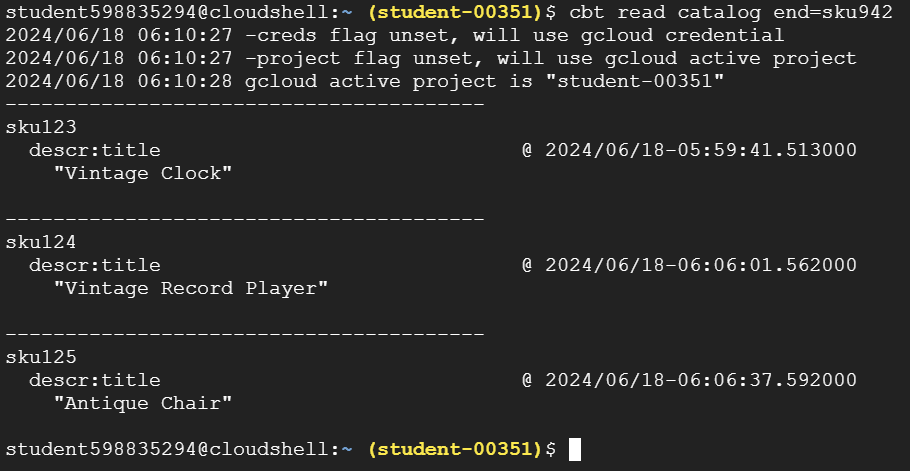
Here we’re saying we want to start reading at sku124 and return all the rest of the rows.

**cbt read catalog start=sku124**

****

Or, read all the rows up to but excluding sku942

**cbt read catalog end=sku942**

****

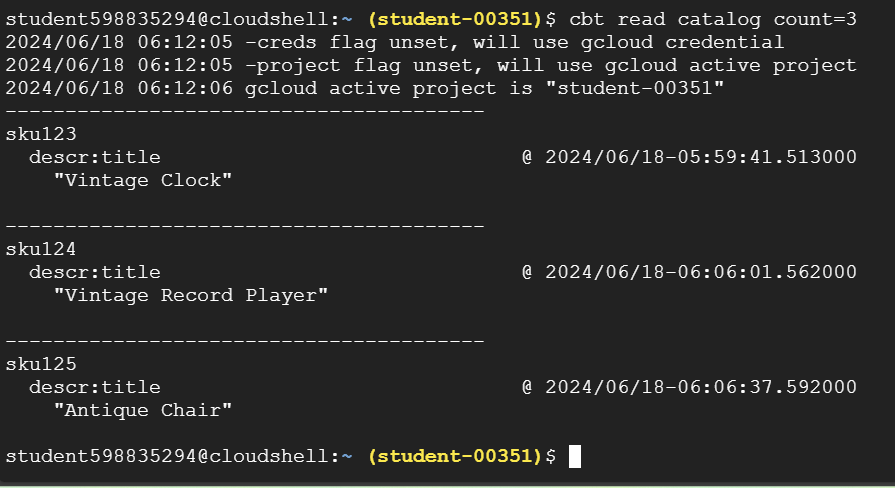
You can combine them of course to get more targeted

**cbt read catalog start=sku124 end=sku942**

## Count

Finally we have count. It’s pretty self explanatory, cont returns only X number of rows that you indicate. This comes in handy when dealing with time series data and other scenarios.

**cbt read catalog count=3**

****

# Cleanup

OK that’s it for this session. Let’s delete our instance and clean things up.

Delete the table instance & .cbtrc file

cbt deletetable catalog  
cbt deleteinstance my-instance  
rm ~/.cbtrc