

# Google Cloud Bigtable

COSC 516 – Cloud Databases





# Google Cloud Bigtable

---

**Google Cloud Bigtable** is a managed, scalable NoSQL database service for large analytical/operational workloads needing high availability.

- Data model is key-value not relational. Similar to HBase and Cassandra.

## Key benefits:

- Consistent sub-10ms latency while supporting millions of requests per second
- CPU and storage scale with no downtime
- Easy connections to other Google Cloud services

# Use Cases

## Financial analysis

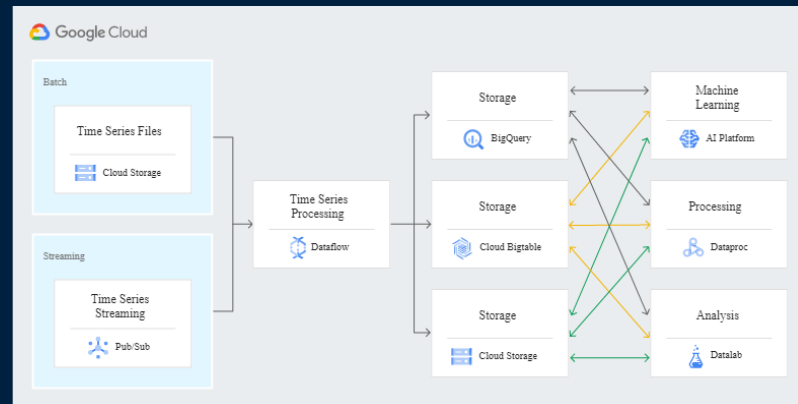
- Analyze historical transaction patterns to detect fraud in real-time or analyze market/trade data.

## IoT

- Ingest and analyze sensor time series data.
- Detect abnormal events or values.
- Analyze and produce dashboards for analytics.

## AdTech

- Integrate data from multiple sources to determine customer behavior across channels.
- Use collected data to drive user-specific campaigns and marketing.



Source: Google <https://cloud.google.com/bigtable>

Bigtable supports applications that need high throughput and scalability for **key/value data**.

May be used as storage engine for batch MapReduce operations, stream processing/analytics, and machine-learning applications.

# Overview

---

**Cloud Bigtable** is a *sparsely* populated table scaling to billions of rows and thousands of columns.

Each row is indexed by a unique **row key**.

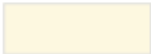
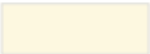
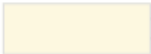
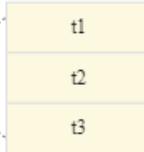



Supports multiple client libraries including an extension to the Apache HBase library for Java.

# Storage Model

Data is stored in a **table**, which is a sorted key/value map.

- Table contains **rows**, uniquely identified by **row key**, and **columns** containing values for each row.
- Columns are grouped into a **column family** and are identified by combining column family and **column qualifier** (unique name within column family).

Each row/column pair can contain multiple cells. Each **cell** contains a unique **timestamped** version of the data for that row and column.

	Column family 1		Column family 2		
	Column 1	Column 2	Column 1	Column 2	
Row key 1					
Row key 2					

# Architecture

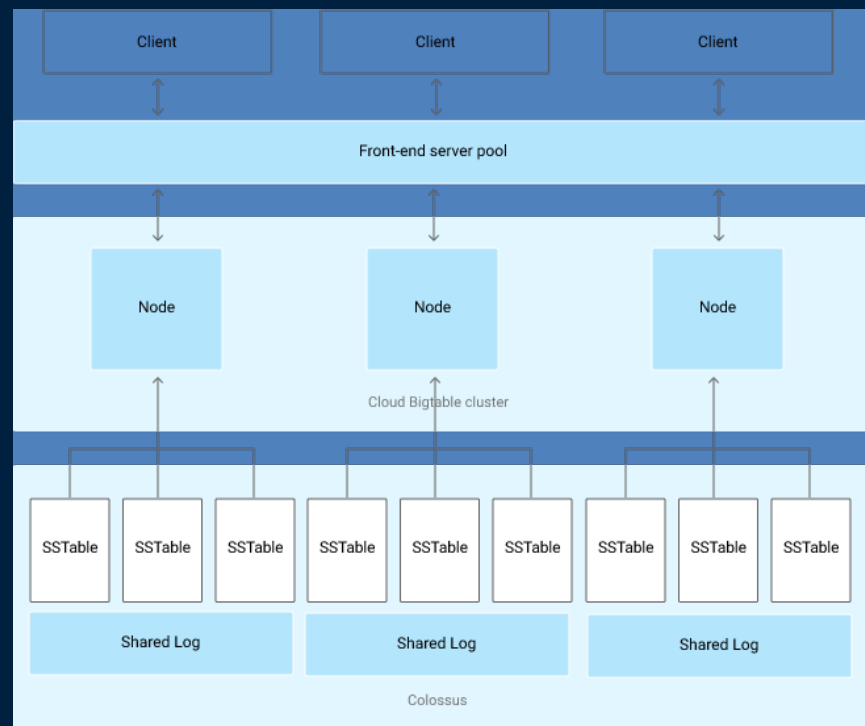
Client requests go to a frontend server then are sent to a **node**.

Nodes are organized into a **cluster**, which belongs to a **instance**, a container for the cluster. Each node handles a subset of requests.

Adding nodes increases throughput. Using replication allows different traffic to be sent to different clusters. Also allows for fail over.

A table is sharded into blocks of contiguous rows, called **tablets**. Tablets are stored in **SSTable** format (ordered immutable map).

Keys and values are arbitrary byte strings. Each tablet is associated with a specific node. Nodes contain no data themselves.



Source: Google <https://cloud.google.com/bigtable>

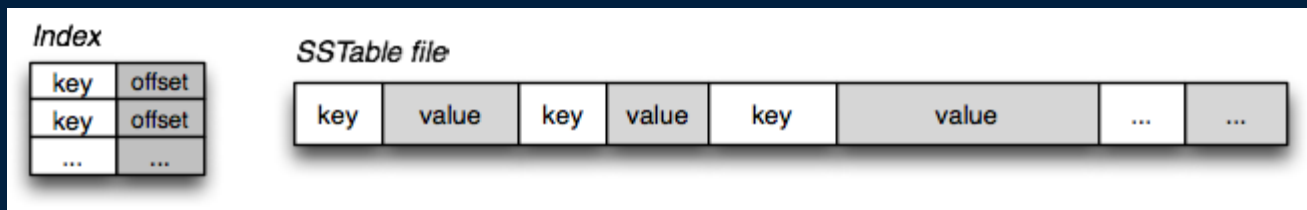


# Sorted String Table (SSTable)

A Sorted String Table (SSTable) is a file containing a set of sorted key value pairs.

- Keys and values are arbitrary byte strings.
- Index provides key/offset pairs for rapid lookup.

SSTable on disk is immutable because insert/update/delete is expensive operation in order to preserve sorted order.



# Clusters

---

A **cluster** provides the Bigtable service in a specific location and belongs to a single instance.

Each cluster is located in a single zone.

An instance can have clusters in up to 8 regions.

Instances with only 1 cluster do not use replication.

Multi-cluster instances will automatically perform replication.



# Load Balancing

---

A Bigtable zone is managed by a primary process balancing workload and data volume within clusters.

System will split large or busy tablets in half and merge smaller tablets together. This redistribution helps maintain performance and response time.

Best performance is achieved by evenly distributing workload. Schema and query design is important to achieve good performance.

- Recommend ensuring row keys do not have predictable order.
- Developing a row key that groups related rows next to another improves read performance.

# Data Compression

---

Bigtable compresses data automatically and the settings are not configurable.

High compression when:

- Patterned data (such as text) versus random data
- Identical values are in same or adjacent rows
- Perform user-level compression of large values greater than 1 MB

# Durability

---

Data is stored on Colossus, Google's durable file system.

When using replication, a copy of data is stored on file system for each cluster.

Automatically creates copies of data to protect against failures and allow for disaster recovery.

# Consistency

---

Single-cluster Bigtable instances provide strong consistency.

Multi-cluster instances provide *eventual consistency*.

- Configurable options for read-your-writes consistency or strong consistency.

Replication delay is typically a few seconds or minutes.

- Varies depending on amount of data written and distance between clusters.

# Backups

---

Backups save a copy of a table's schema and data for use in restore later.

Backups help recover from application-level data corruption or from operator errors.

## Features:

- Fully integrated: Backups are automatically handled with no need for import/export.
- Automatic expiration: user-defined expiry up to 30 days
- Flexible restore options: Can restore a table backup in a different instance.

# Encryption

---

All data stored is encrypted at rest using Google keys.

Customers have option of using their own keys at an additional cost.

- customer-managed encryption keys (CMEK)

# Other Database Options

---

Bigtable is not a relational database. It does not support SQL queries, joins, or multi-row transactions.

## Other Google data services:

- Cloud Spanner or Cloud SQL - SQL support for online transaction processing (OLTP)
- BigQuery - interactive querying for online analytical processing (OLAP)
- Firestore - document database, with ACID transactions and SQL-like queries
- Memorystore – low latency, in-memory data storage
- Firebase Realtime Database – sync data in real time between users



# Schema Design

---

Performance is highly dependent on schema design and matching design to the query frequency.

Data is sorted lexicographically by row key. Selecting the row key and columns is even more critical than relational schema design.

Efficient queries using point lookups by row key or row-range scans for contiguous set of rows. Full table scans are expensive and should be avoided.

# Schema Design: Best Practices

---

## Table and column family design:

- Datasets with similar schema should be in one table rather than separate tables.
- Put related columns in the same column family.
- Choose short but meaningful names for column families.
  - Names are included in the data that is transferred for each request.
  - Put columns that have different data retention needs in different column families.
- In some cases column qualifiers can be used as the data rather than storing a column value.
- Bigtable tables are sparse and support millions of columns. Create as many columns as needed. Unlike relational, all rows do not have the same columns.
  - No space penalty for a column that is not used in a row.
  - Row size limit is 256 MB.

# Schema Design: Best Practices (2)

---

## Row keys:

- **Select row key based on the queries required.**
- Keep row keys short. Max size is 4 KB.
- Row key often consists of multiple delimited values. E.g. field1#field2#field3
  - If possible, use human-readable string values in row keys.

## Avoid:

- Row keys that start with a timestamp. May cause hotspots at a single node.
- Sequential numeric IDs. May also cause unbalanced traffic and hot spots.

# Time Series Design Exercise

---

How would you design a schema in Bigtable where the data consists of time series environmental data:

- Stations: Each with a unique id and location.
- Sensors: Each sensor is connected to a station and measures a single value at each timestamp (e.g. pressure, temperature, humidity).

Discuss your approach in groups.

# Write Requests

---

**Simple write** – atomic write to single row. Provide table name, row key, and changes to row.

**Increment/append** – increment existing numeric value or append data to existing value

**Conditional write** – only write row if condition is true

**Batch writes** – write more than one row. Provide set of row keys and changes for each row.

# Read Requests

---

**Single-row read** – using row key

**Scan range of rows** – read contiguous range of rows by specifying row key prefix or beginning/ending row keys

**Filtered reads** – returns rows that contain specific values

- May scan entire table but only return to client after filter is applied. Filter may remove rows/columns that save data transfer.

Try to avoid queries that require scanning entire table.

# Pricing

---

## 1) Total number of nodes in instance's clusters

- Tracked hourly. Charged each hour for the maximum number of nodes that exist during that hour. Minimum of 1 hour charged.
- Node charges are for provisioned resources, regardless of node usage. Charges apply even if your cluster is inactive.
- Toronto (northamerica-northeast2): \$0.715/hour per node

## 2) Amount of storage

- Measures average amount of data over time and bills as average over the month period.
- With multiple clusters, each keeping its own copy of the data, charged for every copy.
- When delete data it becomes inaccessible immediately, but storage is charged until compaction. This process typically takes up to a week.
- SSD storage: \$0.187 GB/month, HDD storage: \$0.029 GB/month
- Backups: \$0.029 GB/month

## 3) Amount of network bandwidth

- Ingress: Free, Egress with region: Free, Egress with continent: \$0.01/GB



# Pricing example: Single cluster with one node

1 instance in us-central1 (Iowa) with a single cluster with 1 node. 50 GB stored on SSD drives.

Nodes:

- 1 cluster \* 1 node \* 30 days \* 24 hours/day \* \$0.65 per node per hour in us-central1: \$468.00 USD

Storage

- 1 cluster \* 50 GB \* \$0.17 per GB in us-central1: \$8.50

Network

- No network ingress or egress

Monthly total: **\$476.50**

# Pricing Example: 2 clusters in different regions

---

## Setup:

- 1 instance with 1 cluster in us-central1 (Iowa) and 1 cluster in asia-south1 (Mumbai). Each cluster has 18 nodes during days 1-10 and 25 nodes during days 11-30.
- Average of 30 TB of data stored on SSD drives for each cluster
- 10 TB of network ingress to us-central1
- 2 TB of network ingress to asia-south1
- 50 GB of network egress to us-central1
- 25 GB of network egress to europe-north1
- 10 TB of writes replicated from us-central1 to asia-south1
- 2 TB of writes replicated from asia-south1 to us-central1

# Pricing Example: 2 clusters (cont.)

## Nodes (Days 1-10):

- 1 cluster \* 18 nodes \* 10 days \* 24 hours/day \* \$0.65 per node per hour in us-central1: \$2,808.00
- 1 cluster \* 18 nodes \* 10 days \* 24 hours/day \* \$0.748 per node per hour in asia-south1: \$3,231.36

## Nodes (Days 11-30):

- 1 clusters \* 25 nodes \* 20 days \* 24 hours/day \* \$0.65 per node per hour in us-central1: \$7,800.00
- 1 clusters \* 25 nodes \* 20 days \* 24 hours/day \* \$0.748 per node per hour in asia-south1: \$8,976.00

## Storage

- 1 cluster \* 30 TB \* 1,024 GB/TB \* \$0.17 per GB in us-central1: \$5,222.40
- 1 cluster \* 30 TB \* 1,024 GB/TB \* \$0.196 per GB in asia-south1: \$6,021.12

## Network

- 10 TB of ingress to us-central1: No charge
- 2 TB of ingress to asia-south1: No charge
- 50 GB of egress to us-central1: No charge
- 25 GB of egress to europe-north1 \* \$0.12 per GB: \$3.00

## Replication

- 10 TB of writes replicated from us-central1 to asia-south1 \* 1,024 GB/TB \* \$0.12 per GB: \$1,228.80
- 2 TB of writes replicated from asia-south1 to us-central1 \* 1,024 GB/TB \* \$0.12 per GB: \$245.76

Monthly total: **\$35,536.44**

# Conclusion

---

**Google Bigtable** is a managed, scalable NoSQL database service for large analytical/operational workloads needing high availability.

- Data model is key-value not relational. Similar to HBase and Cassandra.

## Key benefits:

- Consistent sub-10ms latency while supporting millions of requests per second
- CPU and storage scale with no downtime
- Easy connections to other Google Cloud services

## Use cases:

- Timeseries data and analysis
- Financial analysis
- AdTech

# Objectives

---

- List the use cases for Google Bigtable.
- Describe the Bigtable architecture.
- Explain why the architecture makes Bigtable especially suited for its use cases.
- Define and explain: table, row, row key, column family, column qualifier, instance, cluster
- Given a design problem, develop a high performing schema design for use with Bigtable. Argue why design and row key selection will demonstrate good performance.
- Explain the pricing model and how that impacts deployment.



THE UNIVERSITY OF BRITISH COLUMBIA

