Bigtable overview

Cloud Bigtable is a sparsely populated table that can scale to billions of rows and thousands of columns, enabling you to store terabytes or even petabytes of data. A single value in each row is indexed; this value is known as the row key. Bigtable is ideal for storing large amounts of single-keyed data with low latency. It supports high read and write throughput at low latency, and it's an ideal data source for MapReduce operations.

Bigtable is exposed to applications through multiple client libraries, including a supported extension to the Apache HBase library for Java (https://hbase.apache.org/). As a result, it integrates with the existing Apache ecosystem of open source big data software.

Bigtable's powerful backend servers offer several key advantages over a self-managed HBase installation:

- **Incredible scalability.** Bigtable scales in direct proportion to the number of machines in your cluster. A self-managed HBase installation has a design bottleneck that limits the performance after a certain threshold is reached. Bigtable does not have this bottleneck, so you can scale your cluster up to handle more reads and writes.
- **Simple administration.** Bigtable handles upgrades and restarts transparently, and it automatically maintains high data durability (#durability). To replicate your data, simply add a second cluster to your instance, and replication starts automatically. No more managing replicas or regions; just design your table schemas, and Bigtable will handle the rest for you.
- **Cluster resizing without downtime.** You can increase the size of a Bigtable cluster for a few hours to handle a large load, then reduce the size of the cluster again—all without any downtime. After you change a cluster's size, it typically takes just a few minutes under load for Bigtable to balance performance across all of the nodes in your cluster.

What it's good for

Bigtable is ideal for applications that need high throughput and scalability for key/value data, where each value is typically no larger than 10 MB. Bigtable also excels as a storage engine for batch MapReduce operations, stream processing/analytics, and machine-learning applications.

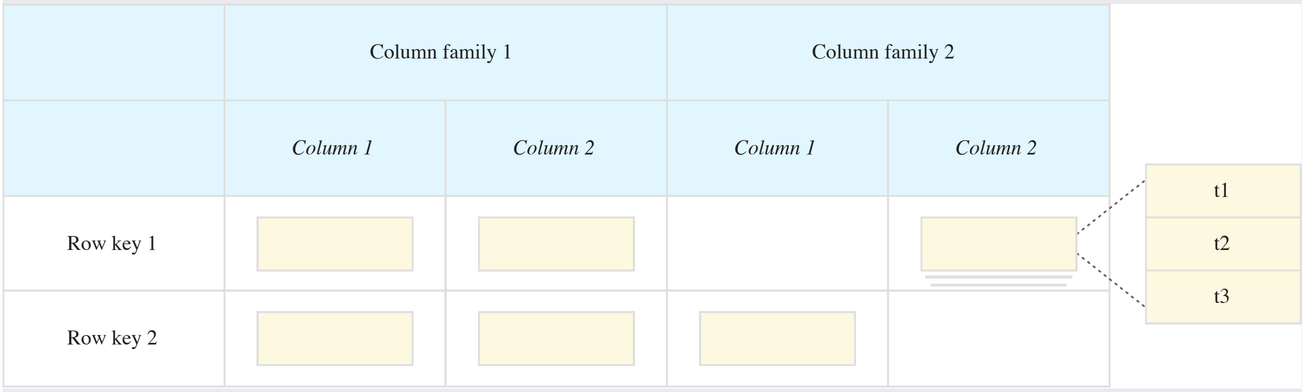
You can use Bigtable to store and query all of the following types of data:

- **Time-series data,** such as CPU and memory usage over time for multiple servers.
- **Marketing data,** such as purchase histories and customer preferences.
- **Financial data,** such as transaction histories, stock prices, and currency exchange rates.
- **Internet of Things data,** such as usage reports from energy meters and home appliances.
- **Graph data,** such as information about how users are connected to one another.

Bigtable storage model

Bigtable stores data in massively scalable tables, each of which is a sorted key/value map. The 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 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.

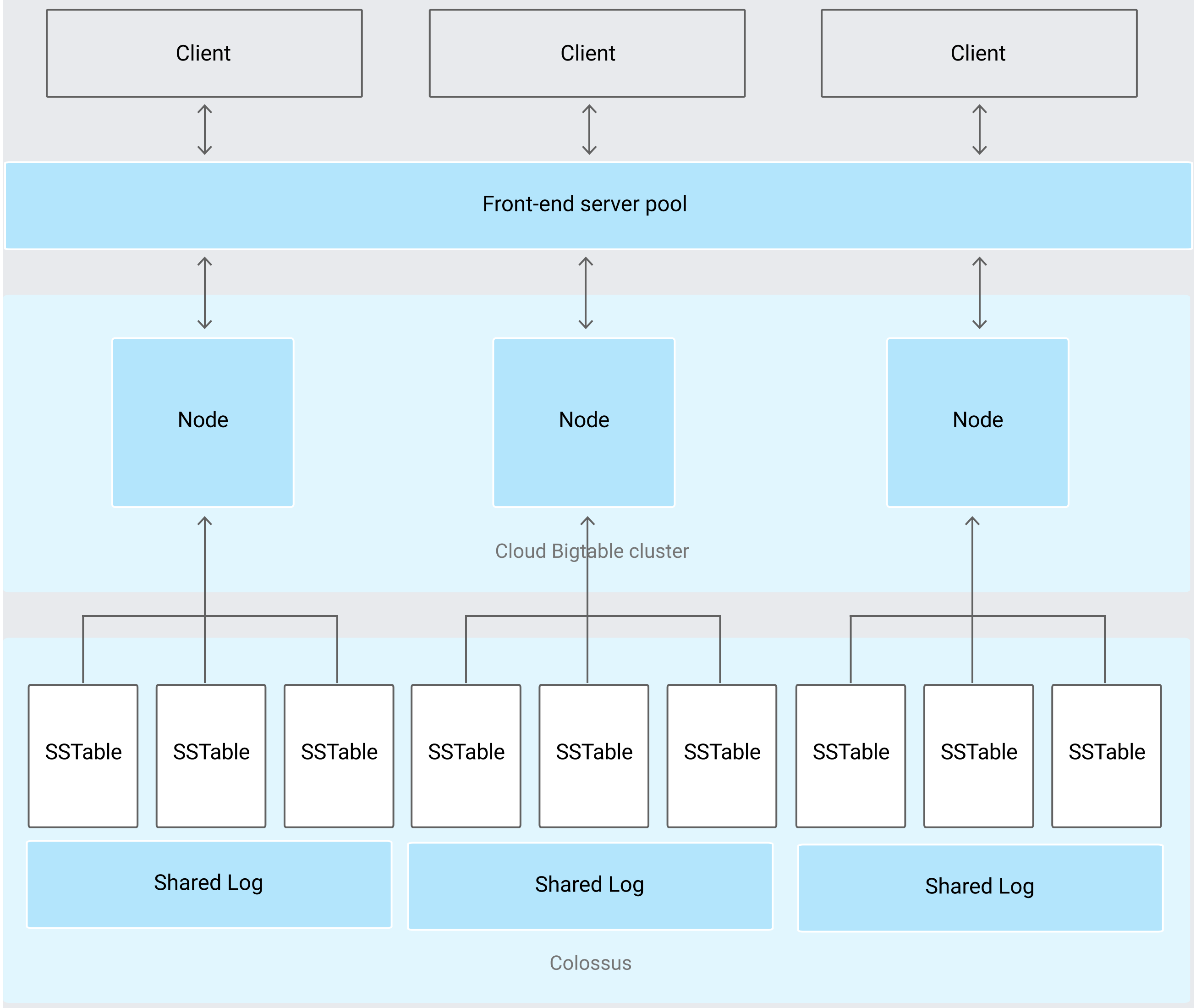


A few things to notice in this illustration:

- Columns can be unused in a row.
- Each cell in a given row and column has a unique timestamp (t).

Bigtable architecture

The following diagram shows a simplified version of Bigtable's overall architecture:



As the diagram illustrates, all client requests go through a frontend server before they are sent to a Bigtable node. (In the original Bigtable paper (https://research.google.com/archive/bigtable-osdi06.pdf), these nodes are called "tablet servers.") The nodes are organized into a Bigtable cluster, which belongs to a Bigtable instance, a container for the cluster.

Note: The diagram shows an instance with a single cluster. You can also add clusters to replicate your data (/bigtable/docs/replication-overview), which improves data availability and durability.

Each node in the cluster handles a subset of the requests to the cluster. By adding nodes to a cluster, you can increase the number of simultaneous requests that the cluster can handle. Adding nodes also increases the maximum throughput for the cluster. If you enable replication by adding additional clusters, you can also send different types of traffic to different clusters. Then if one cluster becomes unavailable, you can fail over to another cluster.

A Bigtable table is sharded into blocks of contiguous rows, called *tablets*, to help balance the workload of queries. (Tablets are similar to HBase regions.) Tablets are stored on Colossus, Google's file system, in SSTable (https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/) format. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Each tablet is associated with a specific Bigtable node. 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.

Importantly, data is never stored in Bigtable nodes themselves; each node has pointers to a set of tablets that are stored on Colossus. As a result:

- Rebalancing tablets from one node to another happens quickly, because the actual data is not copied. Bigtable simply updates the pointers for each node.
- Recovery from the failure of a Bigtable node is fast, because only metadata must be migrated to the replacement node.
- When a Bigtable node fails, no data is lost.

See Instances, Clusters, and Nodes (/bigtable/docs/instances-clusters-nodes) for more information about how to work with these fundamental building blocks.

Load balancing

Each Bigtable zone is managed by a primary process, which balances workload and data volume within clusters. This process splits busier/larger tablets in half and merges less-accessed/smaller tablets together, redistributing them between nodes as needed. If a certain tablet gets a spike of traffic, Bigtable splits the tablet in two, then moves one of the new tablets to another node. Bigtable manages the splitting, merging, and rebalancing automatically, saving you the effort of manually administering your tablets. Understand performance (/bigtable/docs/performance) provides more details about this process.

To get the best write performance from Bigtable, it's important to distribute writes as evenly as possible across nodes. One way to achieve this goal is by using row keys that do not follow a predictable order. For example, usernames tend to be distributed more or less evenly throughout the alphabet, so including a username at the start of the row key will tend to distribute writes evenly.

At the same time, it's useful to group related rows so they are next to one another, which makes it much more efficient to read several rows at the same time. For example, if you're storing different types of weather data over time, your row key might be the location where the data was collected, followed by a timestamp (for example, `WashingtonDC#201803061617`). This type of row key would group all of the data from one location into a contiguous range of rows. For other locations, the row would start with a different identifier; with many locations collecting data at the same rate, writes would still be spread evenly across tablets.

See Choosing a row key (/bigtable/docs/schema-design#row-keys) for more details about choosing an appropriate row key for your data.

Supported data types

Bigtable treats all data as raw byte strings for most purposes. The only time Bigtable tries to determine the type is for increment operations, where the target must be a 64-bit integer encoded as an 8-byte big-endian value.

Memory and disk usage

The following sections describe how several components of Bigtable affect memory and disk usage for your instance.

Unused columns

Columns that are not used in a Bigtable row do not take up any space in that row. Each row is essentially a collection of key/value entries, where the key is a combination of the column family, column qualifier and timestamp. If a row does not include a value for a specific column, the key/value entry is simply not present.

Column qualifiers

Column qualifiers take up space in a row, since each column qualifier used in a row is stored in that row. As a result, it's often efficient to use column qualifiers as data.

Compactions

Bigtable periodically rewrites your tables to remove deleted entries, and to reorganize your data so that reads and writes are more efficient. This process is known as a *compaction*. There are no configuration settings for compactions—Bigtable compacts your data automatically.

Mutations and deletions

Mutations, or changes, to a row take up extra storage space, because Bigtable stores mutations sequentially and compacts them only periodically. When Bigtable compacts a table, it removes values that are no longer needed. If you update the value in a cell, both the original value and the new value will be stored on disk for some amount of time until the data is compacted.

Deletions also take up extra storage space, at least in the short term, because deletions are actually a specialized type of mutation. Until the table is compacted, a deletion uses extra storage rather than freeing up space.

Data compression

Bigtable compresses your data automatically using an intelligent algorithm. You cannot configure compression settings for your table. However, it is useful to know how to store data so that it can be compressed efficiently:

- **Random data cannot be compressed as efficiently as patterned data.** Patterned data includes text, such as the page you're reading right now.
- **Compression works best if identical values are near each other,** either in the same row or in adjoining rows. If you arrange your row keys so that rows with identical chunks of data are next to each other, the data can be compressed efficiently.
- **Compress values larger than 1 MiB before storing them in Bigtable**. This compression saves CPU cycles, server memory, and network bandwidth. Bigtable automatically turns off compression for values larger than 1 MiB.

Data durability

When you use Bigtable, your data is stored on Colossus, Google's internal, highly durable file system, using storage devices in Google's data centers. You do not need to run an HDFS cluster or any other file system to use Bigtable. If your instance uses replication (/bigtable/docs/replication-overview), Bigtable maintains one copy of your data in Colossus for each cluster in the instance. Each copy is located in a different zone or region, further improving durability.

Behind the scenes, Google uses proprietary storage methods to achieve data durability above and beyond what's provided by standard HDFS three-way replication. In addition, we create copies of your data to protect against catastrophic events and provide for disaster recovery.

Consistency model

Single-cluster Bigtable instances provide strong consistency. By default, instances that have more than one cluster provide eventual consistency, but for some use cases (/bigtable/docs/replication-overview#consistency-model) they can be configured to provide read-your-writes consistency or strong consistency, depending on the workload and app profile settings.

Security

Access to your Bigtable tables is controlled by your Google Cloud project and the Identity and Access Management (IAM) roles (/bigtable/docs/access-control#roles) that you assign to users. For example, you can assign IAM roles that prevent individual users from reading from tables, writing to tables, or creating new instances. If someone does not have access to your project or does not have an IAM role with appropriate permissions for Bigtable, they cannot access any of your tables.

You can manage security at the project, instance, and table levels. Bigtable does not support row-level, column-level, or cell-level security restrictions.

Encryption

By default, all data stored within Google Cloud, including the data in Bigtable tables, is encrypted at rest (/security/encryption/default-encryption) using the same hardened key management systems that we use for our own encrypted data.

If you want more control over the keys used to encrypt your Bigtable data at rest, you can use customer-managed encryption keys (CMEK) (/bigtable/docs/cmek).

Backups

Bigtable backups (/bigtable/docs/backups) let you save a copy of a table's schema and data, then restore from the backup to a new table at a later time. Backups can help you recover from application-level data corruption or from operator errors such as accidentally deleting a table.

Request routing with app profiles

App profile routing policies (/bigtable/docs/replication-overview#routing-policies) let you control which clusters handle incoming requests from your applications. Options for routing policies include the following:

- **Single-cluster routing:** Sends all requests to a single cluster.
- **Multi-cluster routing to any cluster:** Sends requests to the nearest available cluster in an instance.
- **Cluster group routing:** Sends requests to the nearest available cluster within a selected group of clusters in an instance.

Other storage and database options

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

- If you need full SQL support for an online transaction processing (OLTP) system, consider Cloud Spanner (/spanner) or Cloud SQL (/sql).
- If you need interactive querying in an online analytical processing (OLAP) system, consider BigQuery (/bigquery).
- If you must store highly structured objects in a document database, with support for ACID transactions and SQL-like queries, consider Firestore (/firestore).
- For in-memory data storage with low latency, consider Memorystore (/memorystore).
- To sync data between users in real time, consider the Firebase Realtime Database (https://firebase.google.com/products/realtime-database/).

For more information about other database options, see the overview of database services (/products/databases). Google Cloud also has various storage options (/products/storage).

What's next

- Try a Bigtable quickstart (/bigtable/docs/create-instance-write-data-cbt-cli) using `cbt`, the command-line tool for Bigtable. If you're familiar with HBase, try the HBase shell quickstart (/bigtable/docs/create-instance-write-data-hbase-shell).
- Work through a Bigtable codelab (https://codelabs.developers.google.com/codelabs/cloud-bigtable-intro-java/index.html).
- Learn more about Bigtable instances, clusters, and nodes (/bigtable/docs/instances-clusters-nodes).
- Learn how to create a Bigtable instance (/bigtable/docs/creating-instance).
- Learn about the client libraries for Bigtable (/bigtable/docs/reference/libraries).
- Read the original OSDI paper (https://research.google.com/archive/bigtable-osdi06.pdf) about Bigtable.

Last updated 2022-09-29 UTC.