https://blog.serverdensity.com/time-series-data-opentsdb-bigtable/

https://cloud.google.com/solutions/monitoring-time-series-data-opentsdb

https://cloud.google.com/solutions/running-janusgraph-with-bigtable

https://www.youtube.com/watch?v=HYvAPjukKic

https://www.youtube.com/watch?v=ChoXxlddGis

https://www.youtube.com/watch?v=CbOH8ixxjdM

https://github.com/GoogleCloudPlatform/professional-services/tree/master/examples/cryptorealtime

https://medium.com/google-cloud/bigtable-beam-dataflow-cryptocurrencies-gcp-terraform-java-maven-4e7873811e86

https://cloud.google.com/blog/products/databases/getting-started-with-time-series-trend-predictions-using-gcp

https://matematika.org/clouds/google/qwiklabs/data-science-on-google-cloud-platform-machine-learning/using-opentsdb-to-monitor-time-series-data-on-cloud-platform/

https://www.qwiklabs.com/focuses/629?parent=catalog

https://stackoverflow.com/questions/39044341/how-to-load-data-into-google-cloud-bigtable-from-google-bigquery

<https://github.com/GoogleCloudPlatform/cloud-bigtable-examples/blob/master/java/dataflow-connector-examples/src/main/java/com/google/cloud/bigtable/dataflow/example/BigQueryBigtableTransfer.java>

1. What is Bigtable?

Bigtable began as the storage system for the web search index at Google and has become one of the main technologies backing many of the other storage systems at Google, such as Megastore and Cloud Datastore. It was built to solve a specific but complex problem: How do you store and continuously update petabytes of data, with incredibly high throughput, low latency, and high availability?

The obvious question is why you can’t toss all of this into MySQL. MySQL falls over quickly in attacking this problem, so Google came up with an interesting way of using a globally sorted key-value map, which automatically rebalances data based on service use to reach the performance and scale requirements needed. Let’s look more closely at the design goals (and nongoals) that went into building Cloud Bigtable and how they affect whether you should use Bigtable in your own applications.

1.1 Design goals

Because the primary use case for Bigtable was the web search index, let’s look specifically

at those requirements. Google’s web search index is one of those things that

must be always on and always fast, so it should come as no surprise that many of the

requirements are related to both performance and scale—which come at the cost of

sacrificing many of the nice-to-have features common in modern databases.

LARGE AMOUNTS OF (REPLICATED) DATA

The search index will obviously be enormous, with overall sizes measured in petabytes,

which means that it’s far too large for a single server to manage. This is also a benefit,

however. One of the hidden requirements for the index would be that it’s distributed

across many different servers, each one being in some sense commodity hardware

(aka cheap). This problem is further exacerbated by the need to ensure that the data

itself is stored in more than one place—after all, hard drives and servers can fail and

you wouldn’t want a chunk of the data to disappear (even temporarily) due to occasional

hardware failure.

LOW LATENCY, HIGH THROUGHPUT

Regardless of the size of the data to be stored, the search index clearly sees a ton of

traffic, potentially millions of queries every second. If the search index starts failing

as more and more requests come in at the same time, folks will take their searches

elsewhere.

Each search request needs to return a result quickly, measured in milliseconds.

When you include all of the other things that need to happen to achieve that deadline,

this leaves relatively little time to query the database—likely only a few milliseconds.

Anything more than “get the data at this address” will exceed the time deadline.

RAPIDLY CHANGING DATA

New web pages are added all the time and the search index will be updated by a web

crawler frequently. Regardless of the number of queries asking to find web pages

(number of reads per second), the system must handle lots of updates at the same

time (number of writes per second).Though a single write

request can take longer to finish, the total number of write operations that can be

done in a given period of time needs to be a large number.

HISTORY OF DATA CHANGES

Because the data being stored will change rapidly over time, you want a way to easily

see the data as it was at a particular point in time. The client can do this manually by

constructing keys with timestamps to signify which version of the data you’re referring

to. Letting the storage system track change history, however, keeps your clients thin

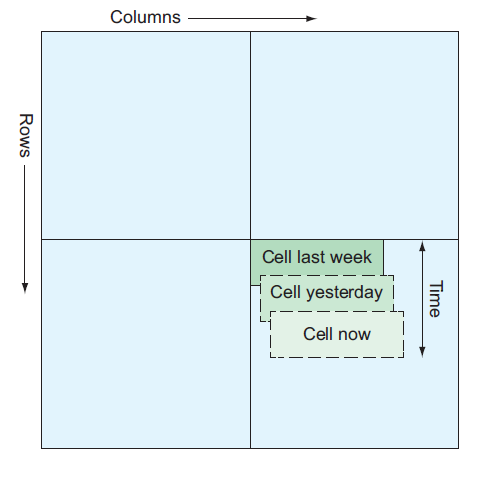
and simple. In some ways, you can think of this as a third dimension to data—typically

databases have a row and column position (two dimensions), but to see history of the

data in a row, you’ll need a third dimension: time. See figure

With this ability, you’ll be able to ask for the latest value in a row, as well as all the values

that this row has had over time.



STRONG CONSISTENCY

Next up is the need for strong consistency, which means anyone querying the index

will never see stale data. Updates either happen everywhere or don’t happen.

If the system didn’t have this property (and was eventually consistent instead), it’d

be possible for someone to search for the same thing in two browser windows and see

different results—definitely not good.

SUBSET SELECTION

Finally it’s important to remember that you don’t always want to request all of the data

stored for a given set of results, so it’d be nice if the system had a way of asking for only

a specific set of properties, such as a specific set of column families, columns, or timestamps,

which would allow you to ask for things like only the two most recent values.

Being able to limit the pieces the storage system should return, allows you to store

more data in one chunk and request only small bits of that large chunk.

1.2 Design nongoals

Quite a few things are not necessarily required—they’d be lovely to have, but you can

do without them if it makes the other aspects possible. In the case of Bigtable, to

achieve the enormous scale of the datasets combined with the throughput and latency

requirements, you would need to drop most of the nice-to-have features such as secondary

indexes (such as the ability to run queries like SELECT \* FROM users WHERE

name = "Jim"), multirow transactional semantics, and many of the other things you’ve

come to expect from databases.

1.3 Design overview

What came out of all of these requirements was a unique storage system that did

things quite differently from most of the nonrelational systems that existed at the

time (2006). As the name suggests, Bigtable is a large table of data with some important

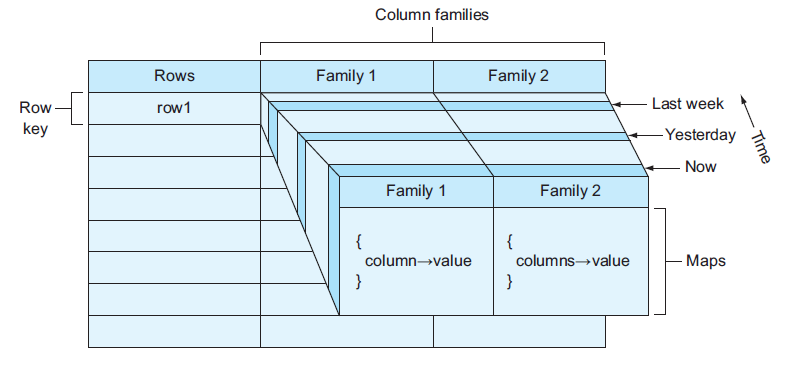
differences from the tables you’ve come to know. Though in many ways it can

act like a traditional table, the storage model of Bigtable is much more like a jagged

key-value map than a grid. In fact, the authors of the research paper describing Bigtable

called it “a sparse, distributed, persistent, multi-dimensional sorted map” (the

key word at this point being “map”). Put visually, this design looks something like



In short, Bigtable is less like a relational database and a bit more like a big key-value

store that distributes data across lots of servers while keeping all the keys in that map

sorted. Thanks to that global sorting, Bigtable allows you to do both key lookups (as

you would in any key-value store) as well as scans over key ranges and key prefixes.

Lastly, hidden in this list of features is the idea that the map is multidimensional.

In this case, the extra dimension attached to all data stored in Bigtable is a timestamp,

which effectively allows you to go back in time and view data as it was at a previous

point. This unique set of features is what makes Bigtable so powerful.

2 Infrastructure concepts

As discussed earlier, Cloud Bigtable acts as a managed service, which means that you

don’t have to manage individual virtual machines like you would if you were running

your own HBase cluster. Automated management features some new concepts that

you’ll need to understand. Unfortunately, Bigtable is one of the more confusing services,

particularly when it comes to how replication is handled. Another tricky area is

that Bigtable itself has a concept of a tablet, which isn’t directly exposed via the Cloud

Bigtable API. To keep things as simple as possible, let’s start by first looking at the hierarchy

of concepts that you can manage yourself: instances, clusters, and nodes.

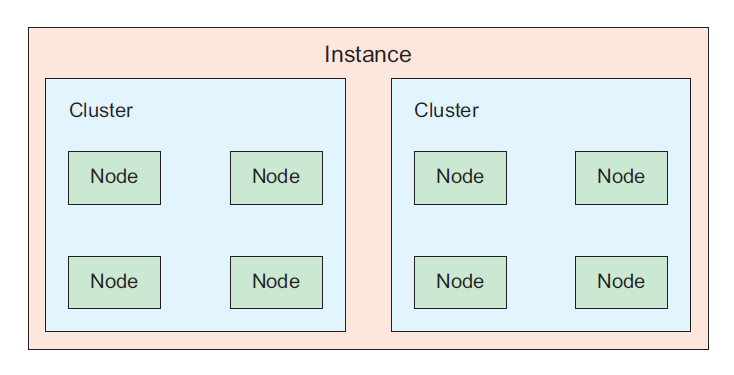


Figure: Hierarchy of instances, clusters, and nodes

As you can see, the basic structure here is that an instance is the top-most concept and

can contain many clusters, and each cluster contains several nodes (with a minimum

of three).

INSTANCES

Think of an instance as the primary resource you refer to when thinking about your

Bigtable deployment, similar to how you’d think of the database server when deploying

a MySQL cluster (with a primary and a read-slave). When you write data to Bigtable,

you’d refer to writing it to a specific Bigtable instance.

Unlike a MySQL cluster where you always write data to the primary, in Bigtable

you send your data to the instance, which ensures that those changes are propagated

to all the other clusters. Although you can address specific clusters directly if

needed, it shouldn’t be necessary because Bigtable should route your queries to the

closest cluster and, therefore, should be reliably fast. Instances are globally scoped,

meaning that they remain addressable regardless of whether a particular zone is experiencing

an outage.

CLUSTERS

Before we go into too much detail about clusters, let’s start with an important caveat:

though figure shows multiple clusters per instance, this is currently not yet possible—

you’re limited to a single cluster per instance. That said, Bigtable will almost certainly

support replication with multiple clusters per instance in the future. Given that

impending launch of the feature, let’s look at how clusters function with the assumption

that you’ll soon be able to maintain many of them inside a single instance.

Clusters, unfortunately (or fortunately, depending on who you ask), are boring.

They’re a grouping for a bunch of nodes, each of which is responsible for handling

some subset of queries sent to a Bigtable instance. Each cluster has a unique name, a

location (zone), and some performance settings such as the type of disk storage to use

as well as the number of nodes to run. Clusters themselves have an hourly computing

cost, as well as a monthly storage cost to reflect the amount of data stored in that particular

cluster. Each cluster holds a copy of your data, so more clusters would imply

higher availability of your data with the obvious trade-off of higher costs. As you’d

expect, your hourly computing cost goes up as you add more nodes, with the benefit

that you’ll never hit a bottleneck of “too many nodes,” as has been known to happen

with other systems such as HBase.

NODES

Nodes are even more boring than clusters for one important reason: from our perspective,

they’re invisible. Although we talk about nodes as discrete individual entities,

in reality you’ll never experience them that way except for seeing them on your bill.

Although you can think of a cluster as a grouping together of multiple nodes, the nodes

themselves are hidden from you in the API. You can communicate only with the particular

cluster that’s responsible for routing your request to a particular node.

This structure allows the cluster to ensure that requests are spread evenly across

the nodes and also allows the cluster to rebalance data to maintain this even distribution.

If nodes themselves were addressable, the cluster wouldn’t be able to move data

around as freely, which could lead to a case where a single node held all the hot data,

driving down performance during busy times. This leads us to the Bigtable concept of

a tablet, which we haven’t yet discussed, but it’s important to understand when you’re

concerned about performance.

TABLETS

Tablets are a way of referencing chunks of data that live on a particular node. The

cool thing about tablets is that they can be split, combined, and moved around to

other nodes to keep access to data spread evenly across the available capacity. As with

nodes, you’ll never address tablets directly, so you won’t see these in the API, but you

can influence how data is written to tablets through the choice of your keys. For example,

writing lots of data quickly over a long period of time to keys with two distinct prefixes

(such as machine\_ and sensor\_) will typically lead to the data being on two

distinct tablets (such as machine\_ prefixed data wouldn’t be on the same tablet as

sensor\_ prefixed data). Let’s take a quick look at the progression of data as you add

more (and query more) over time.

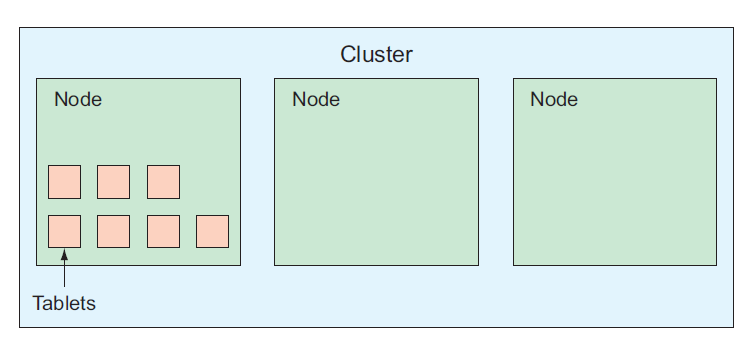


Figure : When starting, Bigtable might put data on a single node.

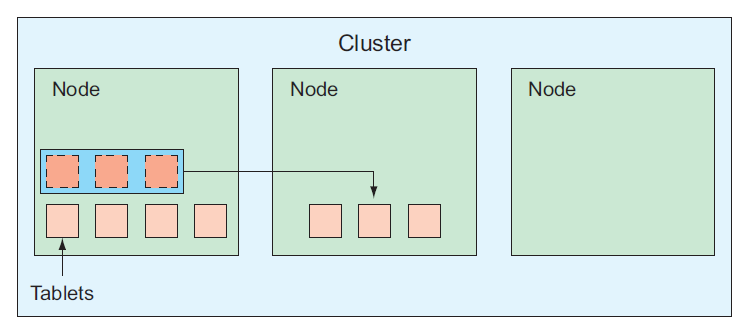


Figure : Bigtable redistributes tablets to spread data more evenly

across nodes.

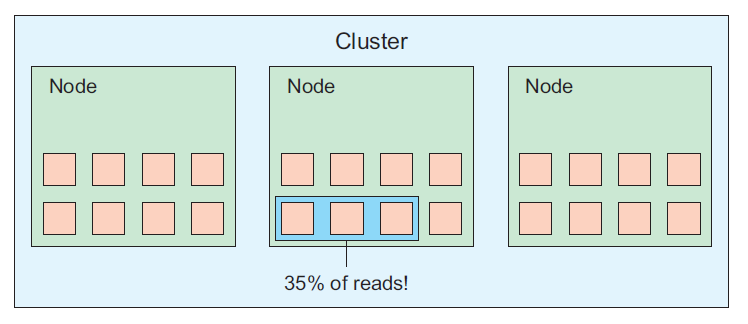


Figure : Sometimes a few tablets are responsible for a high

percentage of traffic

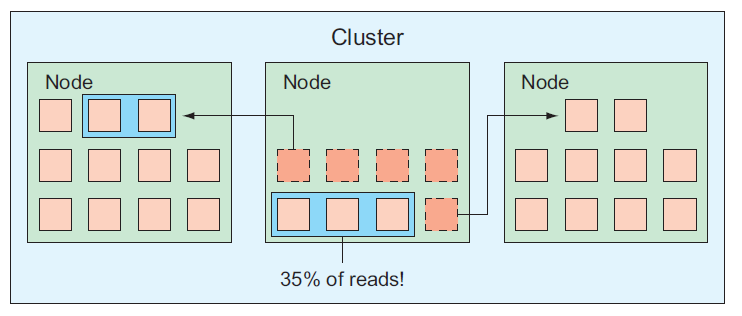
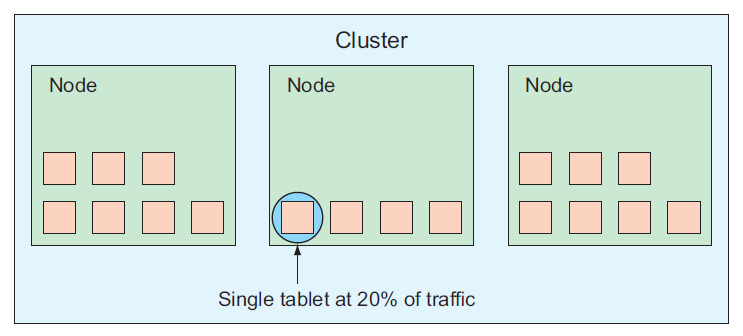
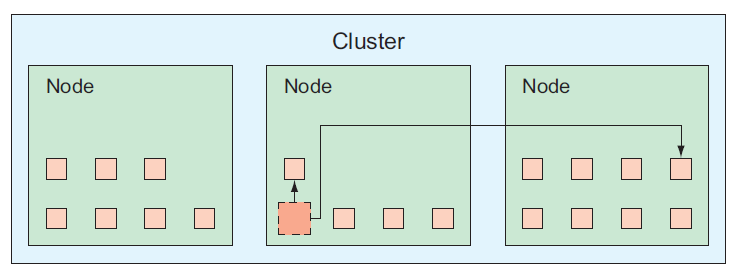


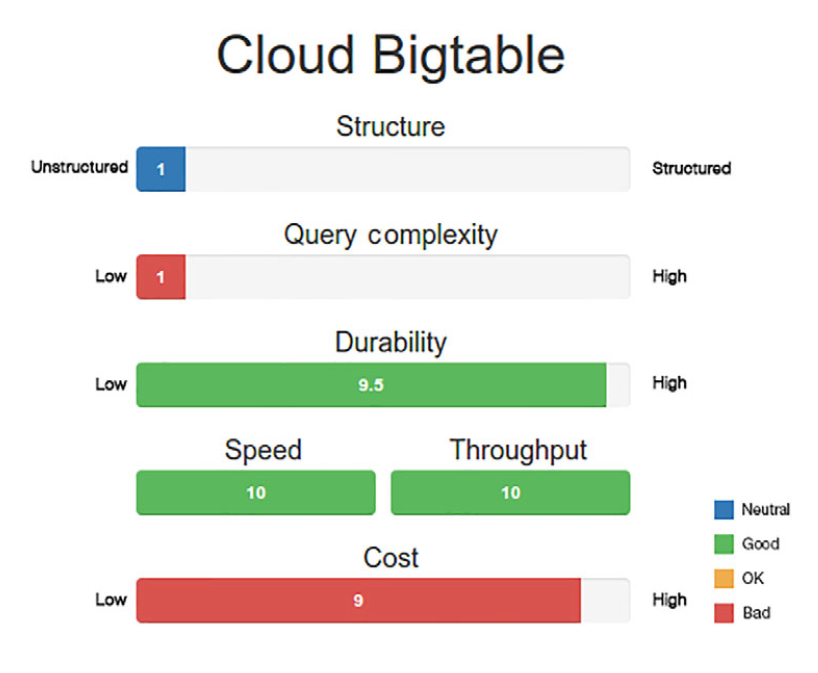
Figure : Bigtable shifts data away from hot tablets.

Figure : Sometimes a single tablet is responsible for a high

percentage of traffic.

 Figure : Bigtable splits tablets and shifts them to other nodes.

3 When should I use Cloud Bigtable?



3.1 Structure

As you’ve learned throughout the chapter, Bigtable is loosely structured when compared

to the other storage systems we’ve seen. Although it does require specific column

family names, the column qualifiers can be dynamic and created on the fly,

meaning the column qualifiers can themselves store data.

In many ways, the structured aspect of Bigtable applies more to the concepts than

it does to the data. Inside that conceptual framework, the column qualifiers and the

values can be anything you want them to be. This freedom, however, means that you

lose out on many of the more advanced features that you might be used to in other

storage systems.

3.2 Query complexity

If a strict key-value storage system (such as Memcache) is an example of a system that

offers the minimal query complexity possible, Bigtable should be considered a hair

above that. As you saw earlier, Bigtable can mimic the key-value querying by constructing

a row key and asking for the data with that row key, but it allows you to do something

critical that services like Memcache don’t: scan the key space.

In most key-value systems, you can request a given key but have no way of asking

for all keys matching a specific prefix (or even “all keys”). In Bigtable you’re able to

specify a range of keys to return, making it important to choose row keys that serve

this purpose. In some ways, this is a bit like being able to choose one and only one

index for your data. Therefore, many things you’re used to with relational databases

are not possible:

 Querying based on data inside a row (SELECT \* FROM employees WHERE name =

'Jimmy' AND age > 20)

 Computing new values based on data (SELECT AVERAGE(age) FROM employees)

 Joining sets of data together in a query (SELECT \* FROM employees, employers

WHERE employees.employer\_id = employer.id)

3.3 Durability

Because all Bigtable data is stored on persistent disk, the chances of losing any stored

data are extraordinarily low. But like any storage system, in addition to worrying about

the underlying storage system (the physical disks), you have to consider the software

system’s persistence model.

In Bigtable’s case, the system is built to shard data across multiple machines (and

multiple tablets) so that the load is spread evenly across the system. Also, Bigtable’s

row-level atomicity means that when writing a row, the write either persists or fails, so

losing data isn’t something to worry about.

3.4 Speed (latency)

One of the main reasons to use Bigtable is its performance. The whole reason you’re

not able to run fancy, complex queries or operate atomically on more than a single

row means that things like reading a single row are incredibly fast (typically below 10 ms,

even with thousands of writes per second). Though some in-memory storage systems

are capable of this, few can maintain this level of speed without sacrificing durability

or concurrency (for example, throughput). The system is able to keep this latency low

because it automatically moves your data around, so choosing a row key is important

and may have adverse effects on performance if done poorly.

3.5 Throughput

As we hinted previously, throughput on Bigtable is best in class for storage systems.

The same aspects of data redistribution that help to keep latency low also help keep

throughput high. Because Bigtable uses SSD disks, random reads and writes are

extremely fast, and many of them happen concurrently. By combining the high performance

of the low-level storage with the even load balancing across tablets, Bigtable

clusters as a whole can handle extraordinarily large levels of throughput, with measurements

starting in the tens of thousands of requests per second.

Further, adding more capacity to the cluster is as simple as adding more nodes.

Because Bigtable will shift data to nodes that are underused, adding more nodes is the

same as having empty nodes with no traffic to them. As you’d expect, Bigtable notices

these empty and idle nodes, shift tablets to them based on the traffic to those tablets.

At the end you have a larger cluster with traffic evenly balanced across each node,

improving your overall throughput.

3.6 Cost

Bigtable’s primary benefit above all else is its performance. Unlike some of the other

storage systems discussed so far, Cloud Bigtable has no free tier and has a minimum

cluster size of three nodes, which translates to about $1,400 per month as a minimum.

This is quite a change from the $30 per month minimum for Cloud SQL.

In short, because of this high initial and on-going cost for Cloud Bigtable, you

should use it only when you absolutely need it due to the scale you expect to see. If

you can make do with something else (for example, MySQL), it’s probably going to be

a better fit.

3.7 Overall

As you might notice, most of the value from Bigtable comes from performance with

both speed and throughput topping the charts. Aside from the performance, Bigtable

acts much like any other key-value store, with almost no structure (you have a row key

that points to mostly unstructured data) and little supported query complexity (you

ask for a row key, or sequence of row keys, and get back subsets of data). If you’re still

wondering why you’d want to use Cloud Bigtable, don’t worry, because you’re not

alone. Bigtable is incredibly powerful, but the lack of common features (such as secondary

indexes) tends to be a big drawback for most projects. Why might you want to

use Bigtable?

First and foremost, Bigtable should always be on the list of options whenever you

have a large dataset. In this case, large typically means terabytes or more. If your data

is only in the gigabyte range (which is typical for a database storing user information),

you’re probably better off with something else.

Second, Bigtable is great for usage sustained over a long period of time. In this

case, a long period of time is measured in hours or days rather than seconds or minutes.

If you use Bigtable to store and query data only infrequently, you’re probably better

off with some other analytical storage system.

Third, Bigtable is likely to be a good fit if you need extraordinarily high levels of

throughput. In this case, extraordinarily high means tens to hundreds of thousands

of queries every second. If you need only a few queries per second, you have many

options and may want to start with another system.

Finally, if you need basic access to your data in the form of lookups and simple

scans across keys, then Bigtable may be a good fit. If you need more than this (like secondary

indexes), you’re probably better off using a relational database. To make this

more concrete, let’s look briefly at our example applications and see whether Cloud

Bigtable might be a good fit.

Summary

* Bigtable is a large-scale data storage system, originally built for Google’s web

search index.

* It was designed to handle large amounts of replicated, rapidly changing data

and can be queried quickly (low latency) with high concurrency (high throughput),

while maintaining strong consistency throughout.

* Cloud Bigtable is a fully managed version of Google’s Bigtable, exposing almost

all of the features available in Google’s original version.

* Bigtable is likely a good fit if you have a large amount of data and primarily

access it using key lookups or key scans but not a great fit if you need secondary

indexes or relational queries.