



Handling throughput and latency requirements

Data Engineering on Google Cloud Platform



©Google Inc. or its affiliates. All rights reserved. Do not distribute.
May only be taught by Google Cloud Platform Authorized Trainers.

1 hour

Please take 5 minutes to give us feedback

g.co/CloudTrainEval

This URL is case-insensitive

The class code is always a tag in the QL class. Instructor can always find the class code (if they don't have it already) when they are in the class in QL, click Edit Class and the code is in the tags. Not ideal - but we'll figure out how to make it more prominent

Agenda

What is Cloud Spanner?

What is Bigtable?

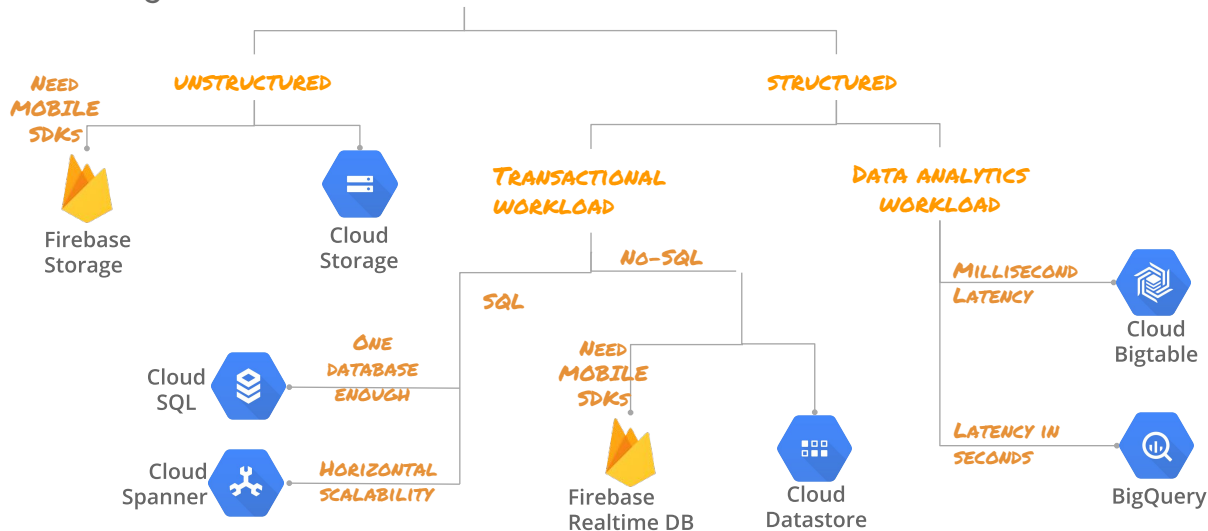
Designing for Bigtable

Ingesting into Bigtable

Lab: streaming into Bigtable

Performance considerations

Choosing where to store data on GCP



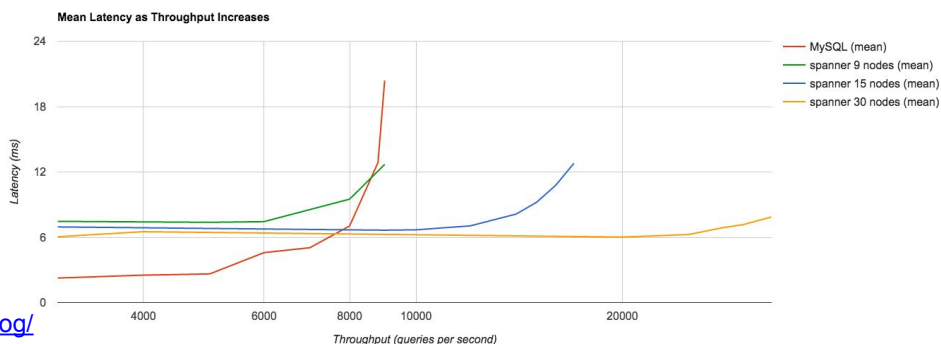
Positioning BigQuery in flow-chart of choices: structured data, primarily for analytics, latency of seconds okay

We'll look at Bigtable and Spanner later in the course, when we talk about streaming. Cloud SQL was covered in Fundamentals course.

Firebase and Datastore are covered in the App Dev course & App Engine content

Horizontal scalability == multiple databases, even globally distributed, with consistency

Use cloud spanner if you need globally consistent data or more than one Cloud SQL instance



Cloud SQL is fine if you can get by with a single database. But if your needs are such that you need multiple databases, Cloud Spanner is a great choice.

The graphs above (by Quizlet) illustrate this. MySQL hits a wall at around 8000 queries/second. If you look at the 99th percentile of latency, it is clear that performance degrades beyond 5000. Distributing MySQL is hard. However, Spanner distributes easily (even globally) and provides consistent performance. To support more throughput, just add more nodes.

How do you use Cloud Spanner?

It's a fully-managed SQL database
Create it from web console (or gcloud)
Interact with it via SQL (from Java, Python, etc.)



<https://cloud.google.com/spanner/docs/quickstart-console>
<https://cloud.google.com/spanner/docs/getting-started/python/>

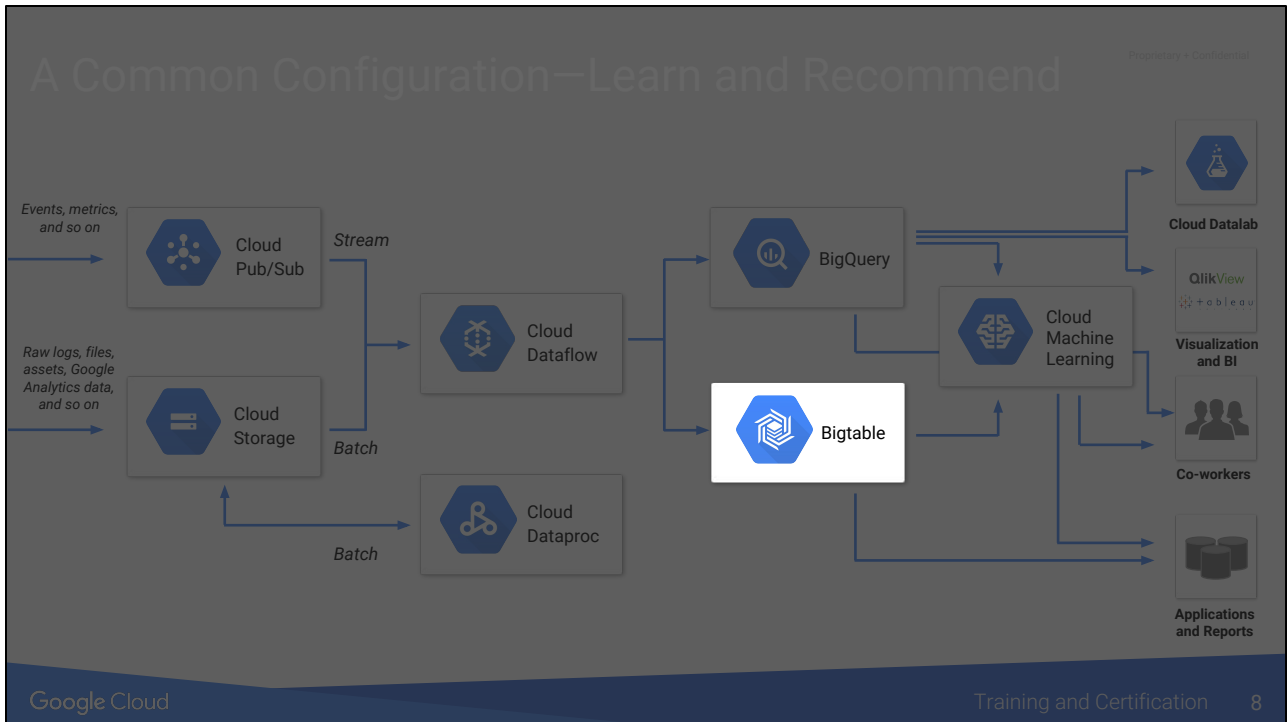
Feel free to make the quickstart-console a demo.

<https://pixabay.com/en/easy-office-business-work-success-1030467/> (cc0)

In many ways, it's just like using Cloud SQL ... that's why we are not going to talk about it any more.

Agenda

What is Bigtable?



Notes:

How our engagement model has changed:

And now, in the new model, we publish the code in tandem with the paper. Case in point, for Dataflow we put the SDK out first (12/2014) and then published the paper (8/2015).

Graphically, the most obvious way to do this would be to use the timeline slide as starting point and do a build-up in 4 steps:

- Show our papers (GFS, MapReduce, Dremel, BigTable, FlumeJava, Millwheel)
- Show how they were followed by open source implementations (see list above)
- Show how for Dataflow we published the SDK ourselves, and it was followed by the paper

Scaling streaming beyond BigQuery

BigQuery:

easy, inexpensive

- latency in order of seconds
- 100k rows/second streaming

Bigtable:

low latency/high-throughput

- 100,000 QPS @ 6ms latency for a 10-node cluster

Notes:

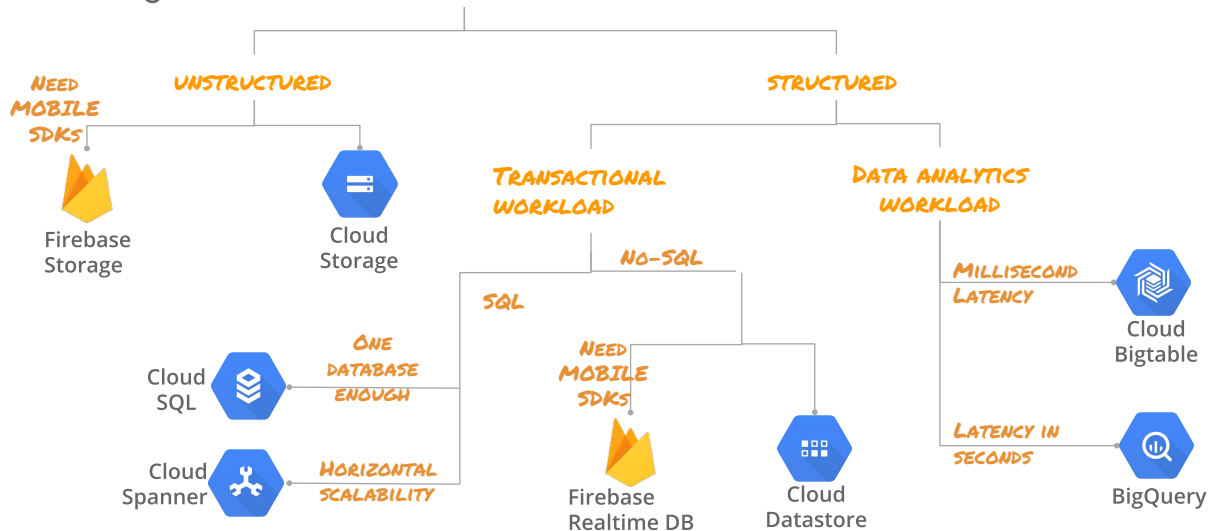
Why Bigtable and not Cloud Spanner? Cost! Note that we can support 100,000 qps with 10 nodes in Bigtable, but would need ~150 nodes in Cloud Spanner.

Blog post to show read/write performance and write throughput:

<https://cloudplatform.googleblog.com/2015/05/introducing-Google-Cloud-Bigtable.html>

In general, a cluster's performance increases linearly as you add nodes to the cluster. For example, if you create an SSD cluster with 10 nodes, the cluster can support up to 100,000 QPS for a typical workload, with 6 ms latency for each read and write operation.

Choosing where to store data on GCP



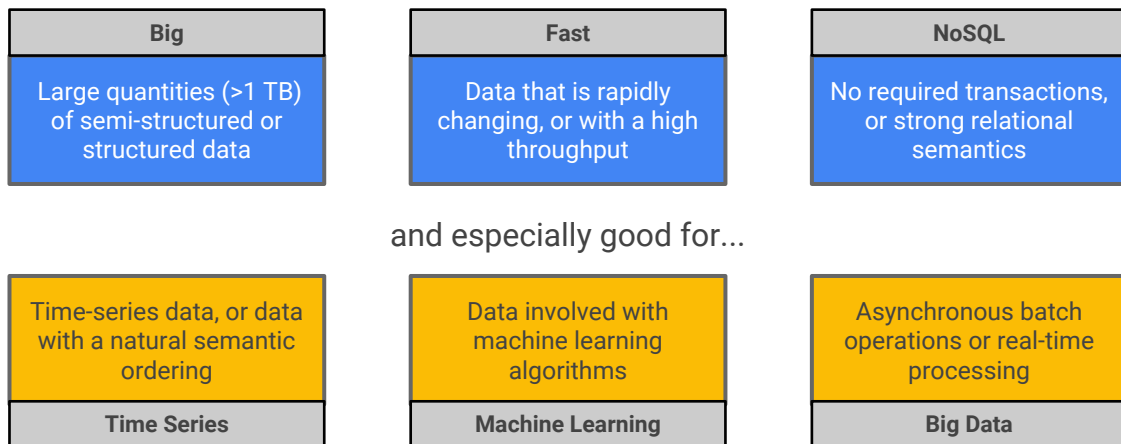
Positioning BigQuery in flow-chart of choices: structured data, primarily for analytics, latency of seconds okay

We'll look at Bigtable and Spanner later in the course, when we talk about streaming. Cloud SQL was covered in Fundamentals course.

Firebase and Datastore are covered in the App Dev course & App Engine content

Horizontal scalability == multiple databases, even globally distributed, with consistency

Bigtable: Scalable, fast NoSQL with auto-balancing



Notes:

Time series: stock tickers...

What is Bigtable good for?

- Cloud Bigtable is ideal for applications that need very high throughput and scalability for non-structured key/value data, where each value is typically no larger than 10 MB.

Bigtable is NOT:

- A good solution for small amounts of data (< 1 TB). Using Bigtable is much like flying by airplane; it doesn't perform as well for traveling small distances (processing small data amounts).
- A good storage choice for highly structured objects, or for use cases requiring ACID transactions and SQL-like queries. For these cases, look to Cloud Datastore.
- A relational database. It does not support SQL queries or joins, nor multi-row transactions. If users need full SQL support for an online transaction processing (OLTP) system, look to Google Cloud SQL.

Additional Use Cases:

- 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.

Cloud Bigtable features

- **Global Availability**

Place your service and data where you want it, with available regions located around the world.

- **Security & Permissions**

Data is encrypted both in-flight and at rest. Enjoy full control over access to data stored in Cloud Bigtable.

- **Redundant Autoscaling Storage**

Bigtable is built leveraging a redundant internal storage strategy for high durability—all managed for you.



Notes:

Scaling / Seamless Cluster

- Increase the size of your Cloud Bigtable cluster for a few hours to handle a large load, then reduce the cluster's size again - all without any downtime.

Global Availability

- Place your service and data where you want it, with available regions located around the world.

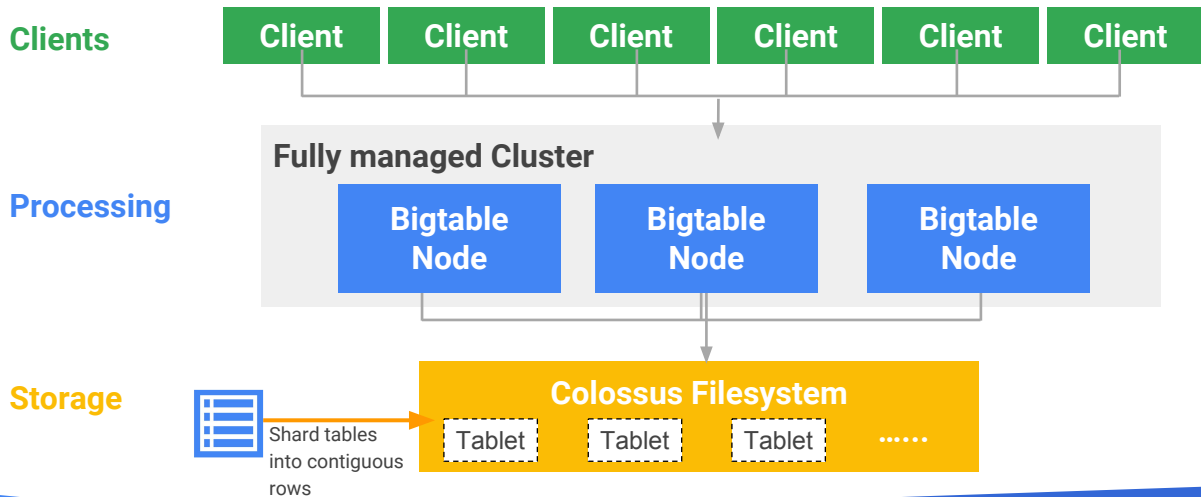
Fully Managed

- Cloud Bigtable is a fully managed service. Focus on developing your applications, instead of configuring and fine-tuning your database.

Redundant Autoscaling Storage

- Bigtable is built leveraging a redundant internal storage strategy for high durability - all managed for you.

Bigtable separates processing and storage



Notes:

A Cloud 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 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 Cloud Bigtable node.

Importantly, data is never stored in Cloud 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 is very fast, because the actual data is not copied. Cloud Bigtable simply updates the pointers for each node.
- Recovery from the failure of a Cloud Bigtable node is very fast, because only metadata needs to be migrated to the replacement node.
- When a Cloud Bigtable node fails, no data is lost.

Bigtable has efficiency when it comes to fast streaming ingestion..just storing data....whereas bigquery is efficient for queries with SQL support

Agenda

Designing for Bigtable

A table can have only one index (the row key)

	user_information				
Row Key	followers	birthdate	age	gender	messageCount
andrew	34	06_04_1986	34	F	1
brianna	31	07_24_1993	23	F	12
caitlyn	55	03_22_1952	51	F	15



**ROWS ARE STORED
IN ASCENDING
ORDER OF THE
ROW KEY**



**ENTRIES IN THE
BIRTHDATE COLUMN
CANNOT BE INDEXED**

Notes:

Each table has only one index, the row key. There are no secondary indices.


Rows are sorted lexicographically by row key, from the lowest to the highest byte string. Row keys are sorted in big-endian, or network, byte order, the binary equivalent of alphabetical order. This is important to keep in mind given how tables are automatically sharded.

All operations are atomic at the row level. For example, if you update two rows in a table, it's possible that one row will be updated successfully and the other update will fail. Avoid schema designs that require atomicity across rows.

Group related columns into column families

This may be a better fit in the *profile_statistics* column family.

These may be useful to group together under *user_information*



	user_information				profile_statistics
Row Key	followers	birthdate	age	gender	messageCount
andrew	34	06_04_1986	34	F	1
brianna	31	07_24_1993	23	F	12
caitlyn	55	03_22_1952	51	F	15

Notes:

Queries can request data by column family, so well-grouped families allows for easier query construction.

Unlike HBase, Bigtable remains performant with up to ~100 column families.

Two types of designs

	user_information				
Row Key	followers	birthdate	age	gender	messageCount
andrew	34	06_04_1986	34	F	1
brianna	31	07_24_1993	23	F	12
caitlyn	55	03_22_1952	51	F	15

follower username hash follows username hash
 2ed5e6cfd887e44

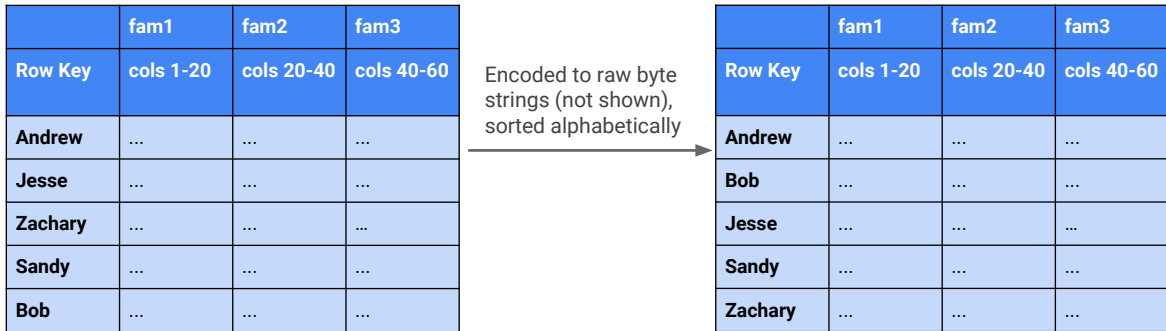
	Follows
Row Key	Username
0d0bab51b0452e5c	tjefferson
2ed5e6cfd887e44	jadams
b0452e5c2ed5e6cf	gwashtington
b0452e5cfd887e44	jadams
b0452e5c0d0bab51	wmckinley
df887e442ed5e6cf	gwashtington
df887e44b0452e5c	tjefferson

WIDE TABLES WHEN EVERY COLUMN VALUE EXISTS FOR EVERY ROW

NARROW TABLES FOR SPARSE DATA

Imagine using the wide-table format for the second case. You'd have a very sparse table since you have millions of users and every user only follows ~100 other users.

Rows are sorted lexicographically by row key,
from lowest to highest byte string



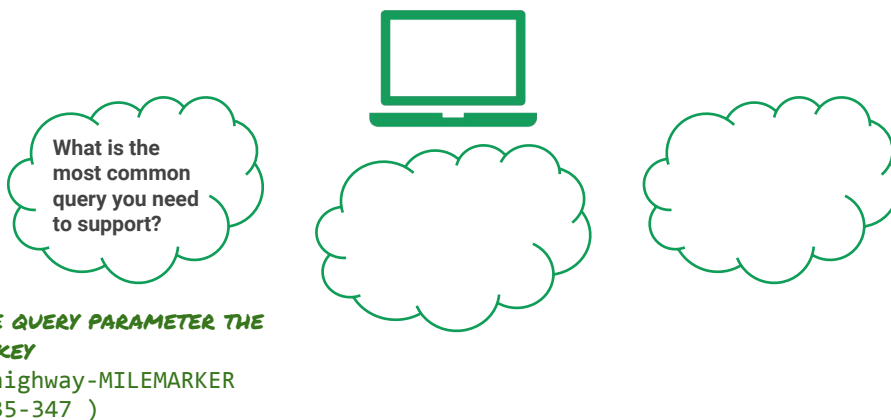
Notes:

Lexicographically is equivalent to alphabetical order. This is ordered based on byte string encoding!

Keep this in mind when designing a schema; you want to avoid certain rows or chunks of rows being overloaded.

You don't typically need to keep the byte string component in mind; for example, when using timestamps along the lines of HH:MM:SS, the byte strings for a given hour and minute will also be very close to each other in order.

Queries that use the row key, a row prefix, or a row range are the most efficient (1/2)



Notes:

Other types of queries trigger a full table scan, which is much less efficient. By choosing the correct row key now, you can avoid a painful data-migration process later.

Start by asking how you'll use the data that you plan to store. For example:

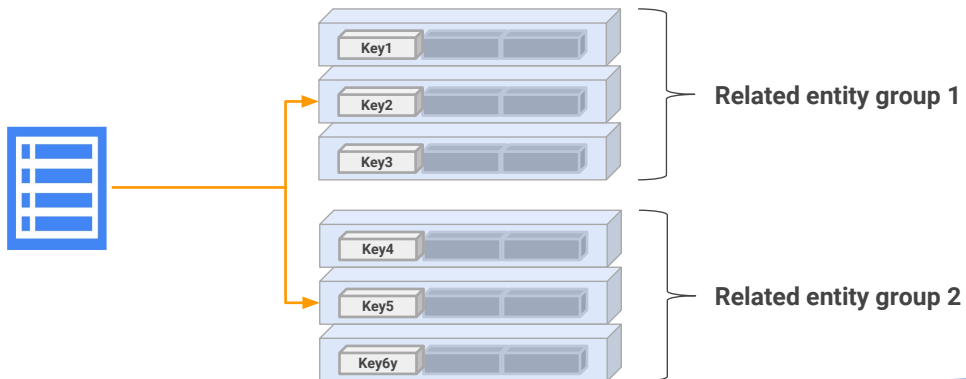
- **User information:** Do you need quick access to information about connections between users (for example, whether user A follows user B)?
- **User-generated content:** If you show users a sample of a large amount of user-generated content, such as status updates, how will you decide which status updates to display to a given user?
- **Time series data:** Will you often need to retrieve the most recent N records, or records that fall within a certain time range? If you're storing data for several kinds of events, will you need to filter based on the type of event?

By understanding your needs up front, you can ensure that your row key, and your overall schema design, provide enough flexibility to query your data efficiently.

The example is for the traffic sensor case. We will often want to know the traffic speed on I-5 between specific mile markers. For example, between mile markers 340 and 350, we can query for keys that start with the prefix I35-34 (i.e. based on row prefix)

Store related entities in adjacent rows

- Entities are considered related if users are likely to pull both records in a single query. This makes reads more efficient.



Notes:

Entities are considered **related** if users are likely to pull both records in a single query.

Suppose u store stock market data...key can be timestamp OR stock symbol and timestamp...latter is better

If its only timestamp, then u will be pulling all records and the filter..and u access many tablets to do the scan

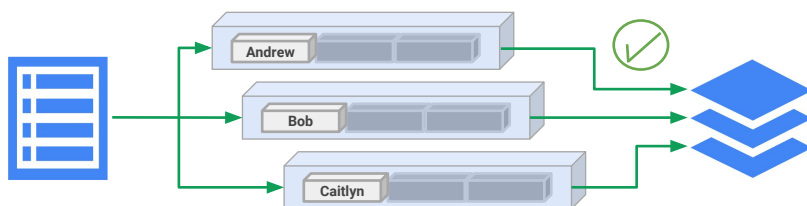
Queries that use the row key, a row prefix, or a row range are the most efficient (2/2)



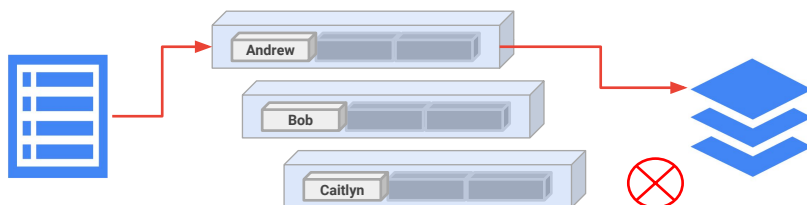
Notes:

Why reverse timestamp? So that the ascending order of row keys puts the latest records at the top of the table. One approach to getting a reverse timestamp is to compute `LONG_MAX - timestamp.millisecondsSinceEpoch()`

Distribute your writes and reads across rows



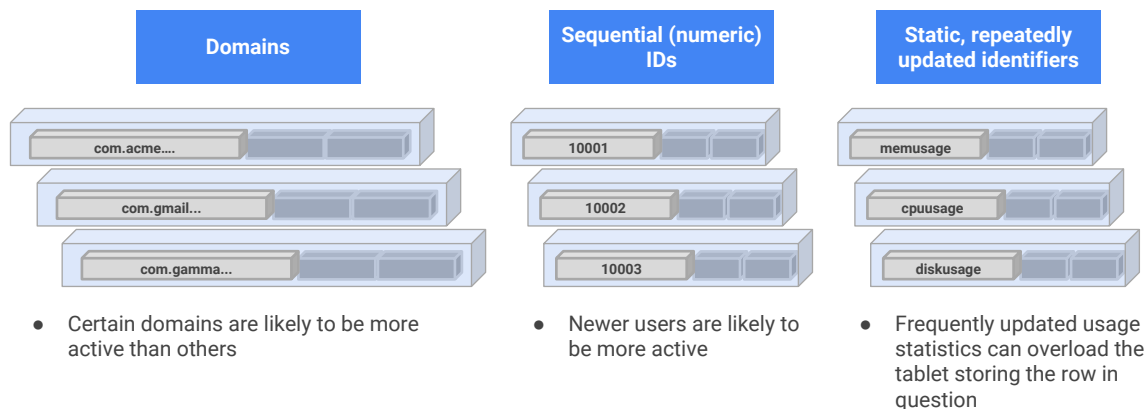
Reads and writes should ideally be **distributed evenly across rows**, so that node workloads are easily balanced.



When read/writes are concentrated in one part of a table, **individual tablets (and their nodes) can be overloaded.**

Design row keys to avoid hotspotting

These are row keys to avoid:



Notes:

Domain names

Avoid using standard, non-reversed domain names as row keys. Using standard domain names makes it inefficient to retrieve all of the rows within a portion of the domain (for example, all rows that relate to `company.com` will be in separate row ranges like `services.company.com`, `product.company.com` and so on). In addition, using standard domain names causes rows to be sorted in such a way that related data is not grouped together in one place, which can result in less efficient compression.

But even if using reversed domain names, some domains (e.g. gmail) will be hot-spotted

Sequential numeric IDs

Suppose your system assigns a numeric ID to each of your application's users. You might be tempted to use the user's numeric ID as the row key for your table. However, since new users are more likely to be active users, this approach is likely to push most of your traffic to a small number of nodes.

A safer approach is to use a reversed version of the user's numeric ID, which spreads traffic more evenly across all of the nodes for your Cloud Bigtable table.

Static, repeatedly updated identifiers

Avoid using a single row key to identify a value that must be updated very frequently. For example, if you store memory-usage data once per second, do not use a single row key named `memusage` and update the row repeatedly. This type of operation overloads the tablet that stores the frequently used row. It can also cause a row to exceed its size limit, since a cell's previous values take up space for a while.

Instead, store one value per row, using a row key that contains the type of metric, a delimiter, and a timestamp. For example, to track memory usage over time, you could use row keys similar to `memusage#1423523569918`. This strategy is efficient because in Cloud Bigtable, creating a new row takes no more time than creating a new cell. In addition, this strategy allows you to quickly read data from a specific date range by calculating the appropriate start and end keys.

For values that change very frequently, such as a counter that is updated hundreds of times each minute, it's best to simply keep the data in memory, at the application layer, and write new rows to Cloud Bigtable periodically.

Distribute the writing load between tablets while allowing common queries to return consecutive rows



What is the most common query you need to support?

MAKE QUERY PARAMETER THE ROW KEY

E.G. highway-MILEMARKER
(I35-347)

Will you often need to retrieve records within a certain time range?

ADD REVERSE TIMESTAMP TO THE ROWKEY

E.G. highway-MILEMARKER-RTS
(I35-347-123456789)

Can you have both distributed writes and block reads?

GOOD:

highway-MILEMARKER-RTS

BAD: (I35-347-123456789)

TS-highway-MILEMARKER
(20170531T102354-I35-347)

MILEMARKER-highway-RTS
(347-I35-1234567)

Notes:

Q1: assume that we want to show data on a traffic map. The most common query will involve highway and range of mile markers (e.g. I35 between exits 332 and 363).

Q2: If you store in original timestamp order, then new data will at the *bottom* of the table. We want table scans within a tablet to return the first few rows if possible. Hence, reverse time stamp.

Q3: The problem with the first bad example is that writes won't be distributed. The problem with the second bad example is that the reads won't be of consecutive blocks. Remember that rows are stored consecutively. So, if all your keys at a certain time start with the same string, they will all hit the same tablet. That's the problem with starting with the timestamp (same problem occurs for reverse timestamp since the number of seconds left till 2028 is pretty much the same now as it is 10 minutes from now; it's just easier to see with real timestamp). Starting the key with the milemarker makes block reads hard, since queries will be for specific highways (think about an application that displays traffic on a map).

Agenda

Ingesting into Bigtable

Create a Bigtable cluster using gcloud (or web UI)

```
gcloud beta bigtable instances create INSTANCE --cluster=CLUSTER \  
  --cluster-zone=CLUSTER_ZONE \  
  [--instance-type=INSTANCE_TYPE; default="PRODUCTION"] \  
  [--cluster-num-nodes=CLUSTER_NUM_NODES] \  
  --description=DESCRIPTION [--async] \  
  [--cluster-storage-type=CLUSTER_STORAGE_TYPE; default="SSD"] \  
  --
```

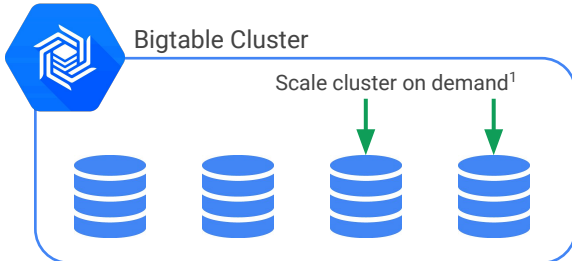
Notes:

Production instances have 3 or more nodes while development is single node cluster.

Modifying Bigtable clusters



You can add or remove nodes, or change a cluster's name, without any downtime



¹ It may take a few minutes under load for Bigtable to balance performance across nodes after resizing your cluster.

Note, you cannot modify the Cluster ID, Zone, or Storage Type after a cluster is created.

Notes:

After you create a Cloud Bigtable cluster, you can update the cluster's name and the number of Cloud Bigtable nodes for the cluster. For example, you may want to add nodes to your cluster for a few hours to handle a large load, then reduce the number of nodes again.

There is no downtime when you make changes to your cluster. If you change the number of nodes in your cluster, it typically takes a few minutes under load for Cloud Bigtable to balance performance across all of the nodes.

You cannot update the cluster ID or the zone in which the cluster is located. To change these values, you must create a new cluster with new cluster ID or zone, copy your data from the old cluster into the new one, and then delete the old cluster.

Top Ref.: <https://cloud.google.com/solutions/bigdata/>

Global Network: <https://cloud.google.com/networking/>

Innovation: (see Top Ref.)

Scale: (see Top Ref.)

Connect to Bigtable instance and create table

```
from google.cloud import bigtable

client = bigtable.Client(project=project_id, admin=True)
instance = client.instance(instance_id)
```

```
print('Creating the {} table.'.format(table_id))
table = instance.table(table_id)
table.create()
column_family_id = 'cf1'
cf1 = table.column_family(column_family_id)
cf1.create()
```

Notes:

Python code

The client must be created with admin=True because it will create table.

Write rows into table

```
column_id = 'greeting'.encode('utf-8')
greetings = ['Hello World!', 'Hello Cloud Bigtable!', 'Hello Python!']

for i, value in enumerate(greetings):
    row_key = 'greeting{}'.format(i)
    row = table.row(row_key)
    row.set_cell(
        column_family_id,
        column_id,
        value.encode('utf-8'))
    row.commit()
```

Notes:

The typical notion of a transaction does not exist

The row key is based on sequential numeric ids...not ideal for production use cases. Since rows are stored in sorted order by key, sequential keys can result in poor distribution of operations across nodes.

Steps to stream into Bigtable using Dataflow

1. Get/create table
2. Convert object to write into Mutation(s)
3. Write mutations to Bigtable

Notes:

1. Create table

```
BigtableOptions.Builder optionsBuilder = new BigtableOptions.Builder()
    .setProjectId(options.getProject())
    .setInstanceId(INSTANCE_ID).setUserAgent("my-program-name");
BigtableSession session = new BigtableSession(
    optionsBuilder.setCredentialOptions(                GET AUTHENTICATED SESSION
    CredentialOptions.credential(
    options.as(GcpOptions.class).getGcpCredential()).build());
BigtableTableAdminClient tableAdminClient = session.getTableAdminClient();

CreateTableRequest.Builder createTableRequestBuilder = // CREATE TABLE
    CreateTableRequest.newBuilder().setParent(getInstanceName(options)) //
    .setTableId(TABLE_ID).setTable(tableBuilder.build());
tableAdminClient.createTable(createTableRequestBuilder.build());
```

The Bigtable connector is a bit code-heavy. This code is to authenticate to Google Cloud, and then create a table.

2. Convert object to Mutation(s) inside a ParDo

```
LaneInfo info = c.element();
DateTime ts = fmt.parseDateTime(info.getTimestamp().replace('T', ' '));
// key is HIGHWAY#DIR#LANE#SENSORID#REVTS
String key = info.getHighway() + "#" + info.getDirection() //
            + "#" + info.getLane() + "#" + info.getSensorKey() //
            + "#" + (Long.MAX_VALUE - ts.getMillis()); // reverse time stamp
// all the data is in a wide column table with only one column family
List<Mutation> mutations = new ArrayList<>();
mutations.add(Mutation.newBuilder().setSetCell(Mutation.SetCell.newBuilder()
        .setValue(Double.toString(info.getLatitude()))//
        .setFamilyName(CF_FAMILY).setColumnQualifier("latitude"))//
        .setTimestampMicros(ts)).build())
// other columns
c.output(KV.of(ByteString.copyFromUtf8(key), mutations));
```

This is akin to converting an Object to TableRow when writing out BigQuery. A mutation is a change to a single cell (different applications could manage different column families for the same row key).

3. Write mutations to Bigtable

```
mutations.apply("write:cbt",  
    BigtableIO.write().withBigtableOptions(  
        optionsBuilder.build()).withTableId(TABLE_ID))
```

Reading from Bigtable

Typically programmatic (using HBase API)

HBase command-line client

Or even BigQuery

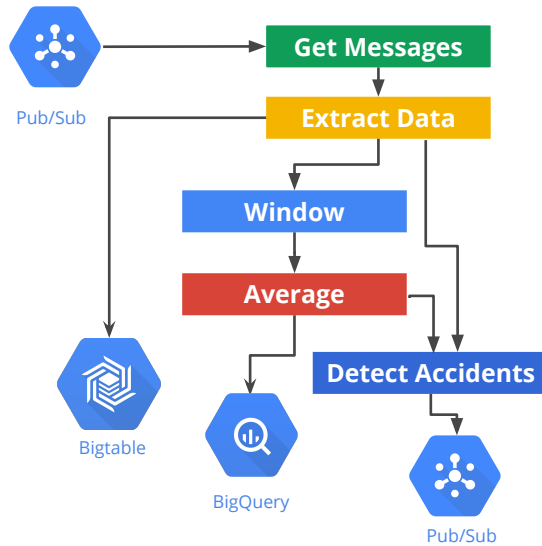
<https://cloud.google.com/bigquery/external-data-bigtable>

#3 requires creating a federated table in BigQuery. This can be useful if you want to do ETL -- extract data from Bigtable and write it into BigQuery without having to need to write a Dataflow job. You can stream into Bigtable, and then once every hour, do a batch ingest into BigQuery ... this keeps costs low.

Agenda

Streaming into Bigtable

Lab: Stream into Bigtable at low-latency



Notes:

The lab has them doing CurrentConditions so that they can compare resource usage between BigQuery and Bigtable

The code for AccidentAlert has the code like above

Agenda

Performance considerations

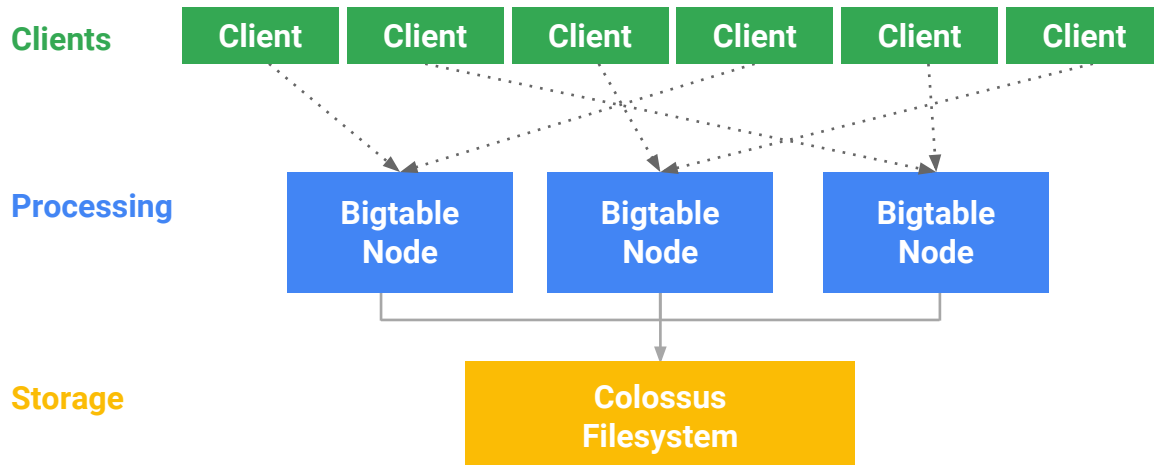
Years of engineering to...

- Teach Bigtable to configure itself
- Isolate performance from “noisy neighbors”
- React automatically to new patterns, splitting, and balancing

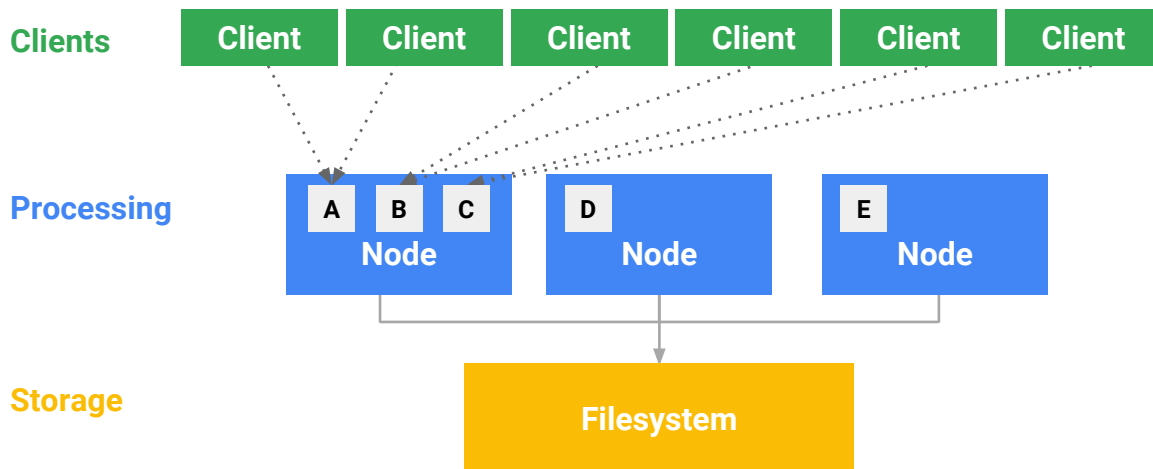


Cloud Bigtable

Bigtable looks at access patterns and improves itself



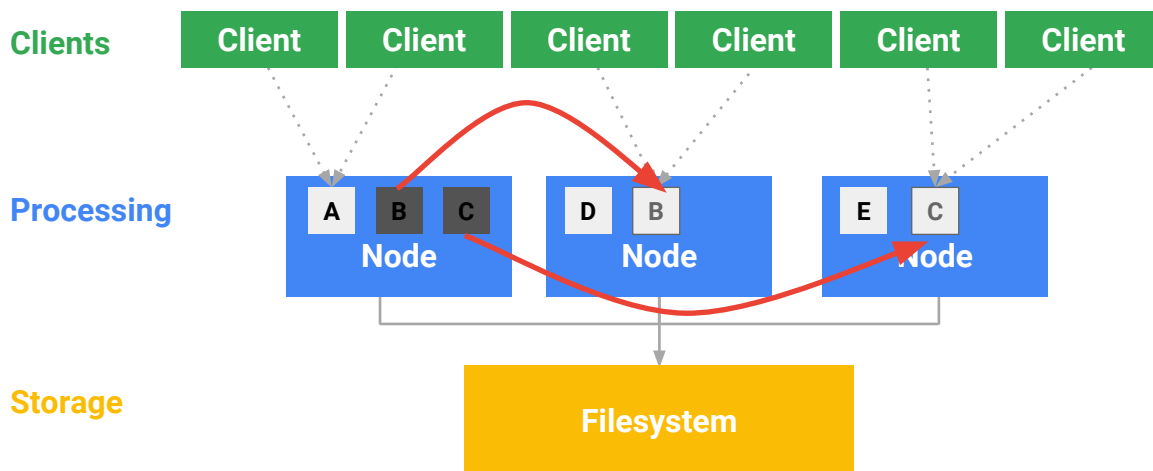
Cloud Bigtable learns access patterns...



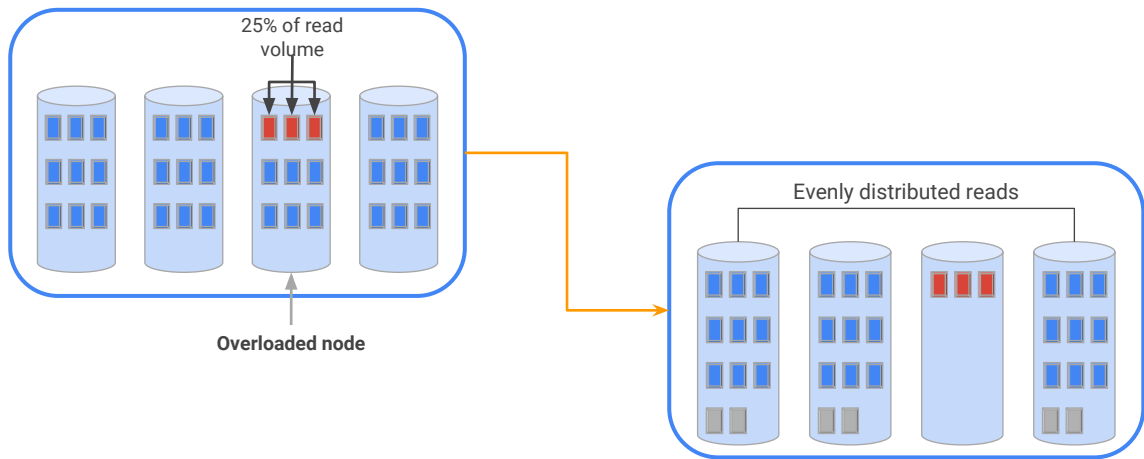
Notes:

A,B,C,D,E are not data but rather pointers/references and cache....which is why rebalancing is not time consuming...we're just moving pointers. Actual data is in tablets in Colossus FS.

...and rebalances data accordingly



Rebalance strategy: distribute reads



Notes:

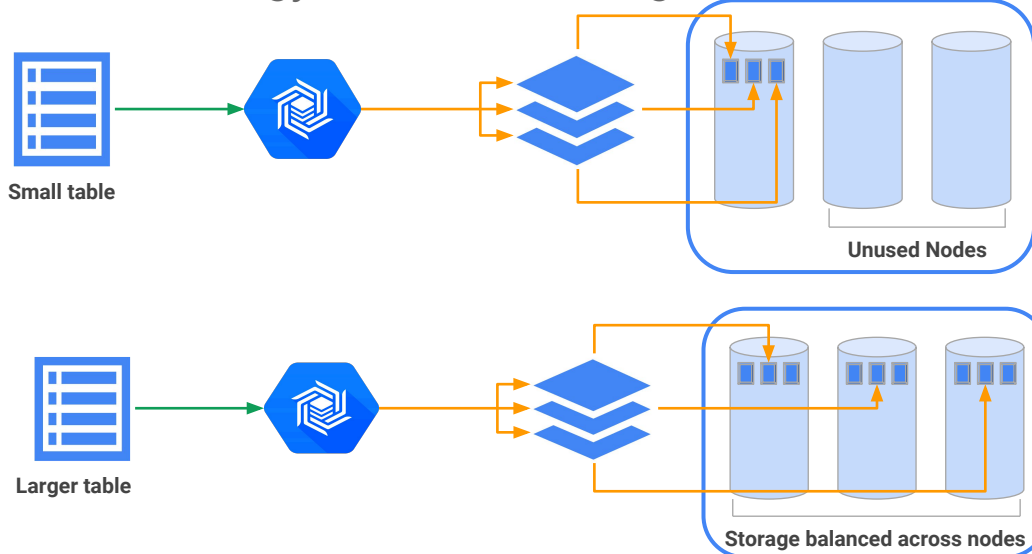
Cloud Bigtable tries to distribute reads and writes equally across all Cloud Bigtable nodes.

With a well-designed schema, reads and writes should be distributed fairly evenly across an entire table and cluster. However, in some cases, it is inevitable that certain rows will be accessed more frequently than others.

In these cases, Cloud Bigtable will redistribute tablets so that reads are spread evenly across nodes in the cluster.

Note that ensuring an even distribution of reads has taken priority over evenly distributing storage across the cluster.

Rebalance strategy: distribute storage

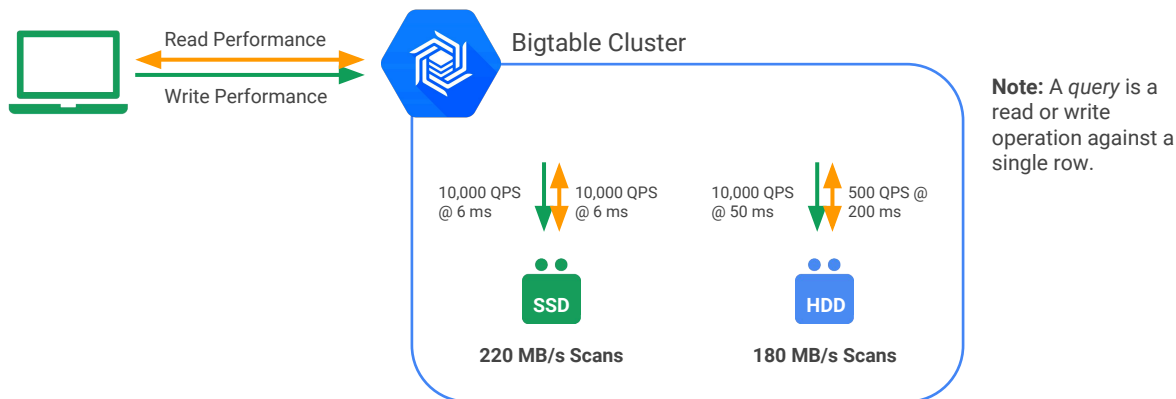


Notes:

Cloud Bigtable tries to store roughly the same amount of data on each Cloud Bigtable node.

Cluster performance

- Under typical workloads, Cloud Bigtable delivers highly predictable performance. When everything is running smoothly, you can expect to achieve the following performance for each node in your Cloud Bigtable cluster, depending on which type of storage your cluster uses.



Notes:

These performance numbers are guidelines, not hard and fast rules. Per-node performance may vary based on your workload and the typical size of each row in your table. In general, a cluster's performance increases linearly as you add nodes to the cluster.

Tips for improving performance

Change schema to minimize data skew

Test with > 300 GB and for minutes-to-hours

Performance increases linearly with number of nodes

Takes a while after scaling up nodes for performance improvement to be seen

Disk speed: SSD faster than HDD

Make sure clients & Bigtable are in same zone

Notes:

There are several factors that can result in slower performance:

- **The table's schema is not designed correctly**
 - It's essential to design a schema that allows reads and writes to be evenly distributed across the Cloud Bigtable cluster. Otherwise, individual nodes can get overloaded, slowing performance.
- **The workload isn't appropriate for Cloud Bigtable**
 - Testing with a small amount (< 300 GB) of data, or for a very short period of time (seconds rather than minutes or hours), Cloud Bigtable won't be able to properly optimize your data. It needs time to learn your access patterns, and it needs large enough shards of data to make use of all of the nodes in your cluster.
- **The Cloud Bigtable cluster doesn't have enough nodes**
 - Typically, performance increases linearly with the number of nodes in a cluster. Adding more nodes can therefore improve performance. Use the monitoring tools to check whether a

- cluster is overloaded.
- **The Cloud Bigtable cluster was scaled up very recently**
 - While nodes are available in your cluster almost immediately, Cloud Bigtable can take up to 20 minutes under load to optimally distribute cluster workload across the new nodes.
- **The Cloud Bigtable cluster uses HDD disks**
 - Using HDD disks instead of SSD disks means slower response times and a significantly lower cap on the number of read requests handled per second (500 QPS for HDD disks vs. 10,000 QPS for SSD disks).
- **There are issues with the network connection**
 - Network issues can reduce throughput and cause reads and writes to take longer than usual. In particular, you'll see issues if your clients are not running in the same zone as your Cloud Bigtable cluster.

Because different workloads can cause performance to vary, you should perform tests with your own workloads to obtain the most accurate benchmarks.

[https://pixabay.com/en/skewness-tower-pisa-building-1026740/\(cc0\)](https://pixabay.com/en/skewness-tower-pisa-building-1026740/(cc0))

[https://pixabay.com/en/black-business-computer-computing-18320/\(cc0\)](https://pixabay.com/en/black-business-computer-computing-18320/(cc0))

[https://pixabay.com/en/grid-wire-mesh-stainless-rods-826831/\(cc0\)](https://pixabay.com/en/grid-wire-mesh-stainless-rods-826831/(cc0))

[https://pixabay.com/en/hourglass-sand-watch-time-glass-1046841/\(cc0\)](https://pixabay.com/en/hourglass-sand-watch-time-glass-1046841/(cc0))

[https://pixabay.com/en/pedestrian-zone-rotterdam-street-1851577/\(cc0\)](https://pixabay.com/en/pedestrian-zone-rotterdam-street-1851577/(cc0))

[https://pixabay.com/en/disc-reader-reading-arm-hard-drive-1085276/\(cc0\)](https://pixabay.com/en/disc-reader-reading-arm-hard-drive-1085276/(cc0))

Bigtable Nodes	QPS
0	0
25	~300,000
300	3,000,000



cloud.google.com