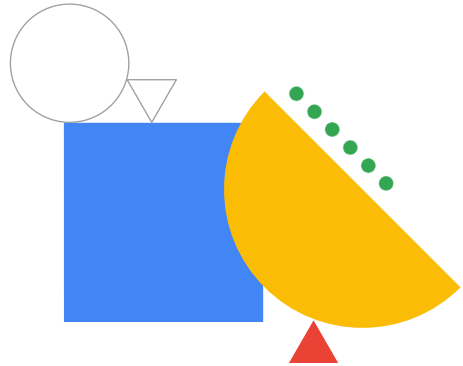




High-Throughput BigQuery and Bigtable Streaming Features



Welcome to the module: High-Throughput BigQuery and Bigtable Streaming Features.



Module agenda



- 01 Streaming into BigQuery and Visualizing Results
- 02 High-Throughput Streaming with Cloud Bigtable
- 03 Optimizing Cloud Bigtable Performance

In this module, you will learn about the analysis of streaming data and the throughput constraints associated with it. These analysis systems are primarily responsible for analyzing data in real time and helping make timely business decisions. We're going to talk about both BigQuery and Cloud Bigtable.

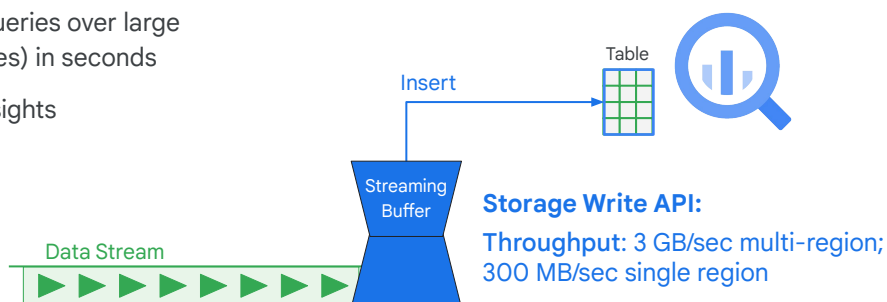


Streaming into BigQuery and Visualizing Results

In this first lesson, we'll discuss the streaming of data into BigQuery and using Looker Studio to visualize results.

BigQuery allows you to stream records into a table; query results incorporate latest data

- Interactive SQL Queries over large datasets (petabytes) in seconds
- Near-real-time insights



Note:

Unlike load jobs, there is a cost for streaming inserts (see [quota and limits](#))

Streaming data is not added to BigQuery via a load job. There is a separate BigQuery method called, "streaming inserts". Streaming inserts allows you to insert one item at a time into a table. New tables can be created from a temporary table that identifies the schema to be copied. Usually the data is available within seconds. The data enters a streaming buffer, where it is held briefly until it can be inserted into the table.

Data availability and consistency are considerations. Candidates for streaming are analysis or applications that are tolerant of late or missing data, or data arriving out of order, or data that is duplicated. The stream can pass through other services introducing additional latency and the possibility of errors.

Since streaming data is unbounded, you need to consider the streaming quotas. There is both a daily limit and a concurrent rate limit. You can find more information about these in the online documentation.

The following quota applies when using the preferred Storage Write API:

Throughput: 3 GB per second in multi-region; 300 MB per second in single regions

This leads to a pertinent question: When should you ingest a stream of data rather than use a batch approach to load data?

The answer is: When the immediate availability of the data is a solution requirement.

And the reason: Well, in most cases loading batch data is not charged. Loading streaming data is a charged activity. So use batch loading or repeated batch loading rather than streaming unless real-time is a requirement of the application.

Insert streaming data into a BigQuery table

```
export GOOGLE_APPLICATION_CREDENTIALS="/home/user/Downloads/[FILE_NAME].json"
```

Credentials

The service must have
Cloud IAM permissions
set in the Web UI

```
pip install google-cloud-bigquery
```

Install API

```
from google.cloud import bigquery
client = bigquery.Client(project='PROJECT_ID')
```

Python

Create a client

```
dataset_ref = bigquery_client.dataset('my_dataset_id')
table_ref = dataset_ref.table('my_table_id')
table = bigquery_client.get_table(table_ref) ← - - - Get table access from API
```

**Access dataset
and table**

```
# read data from Cloud Pub/Sub and place into row format
# static example customer orders in units:
```

```
rows_to_insert = [
    (u'customer 1', 5),
    (u'customer 2', 17),
]
```

Insert rows into table

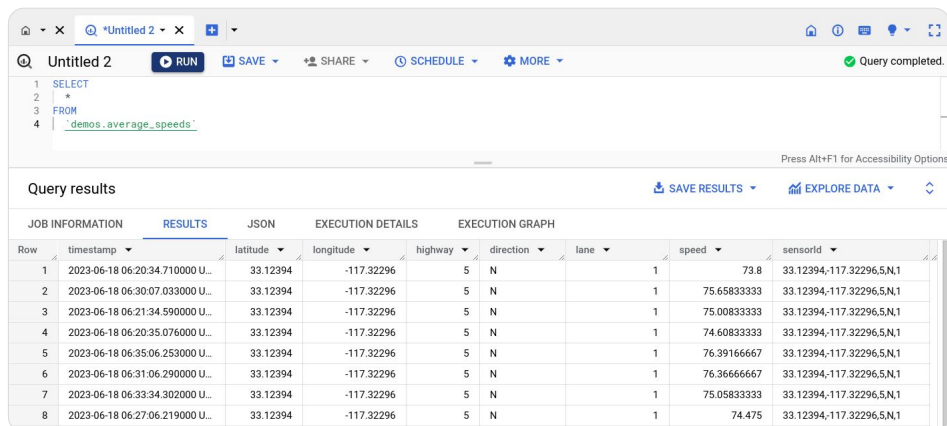
```
errors = bigquery_client.insert_rows(table, rows_to_insert)
```

Perform insert

Here is an example of the code used to insert streaming data into a BigQuery table.

In this example, the message body has already been decoded. In a full example, a step would be required to extract the appropriate message elements to be inserted.

Review streaming data in BigQuery



The screenshot displays the Google Cloud BigQuery console interface. At the top, a query editor shows a SQL query: `SELECT * FROM `demos.average_speeds``. Below the editor, a status bar indicates "Query completed." and provides options to "RUN", "SAVE", "SHARE", "SCHEDULE", and "MORE".

The "Query results" section is active, showing a table with 8 rows of data. The table has columns for timestamp, latitude, longitude, highway, direction, lane, speed, and sensorid. The data represents streaming data from a table named `demos.average_speeds``.

Row	timestamp	latitude	longitude	highway	direction	lane	speed	sensorid
1	2023-06-18 06:20:34.710000 U...	33.12394	-117.32296	5	N	1	73.8	33.12394,-117.32296,5,N,1
2	2023-06-18 06:30:07.033000 U...	33.12394	-117.32296	5	N	1	75.65833333	33.12394,-117.32296,5,N,1
3	2023-06-18 06:21:34.590000 U...	33.12394	-117.32296	5	N	1	75.00833333	33.12394,-117.32296,5,N,1
4	2023-06-18 06:20:35.076000 U...	33.12394	-117.32296	5	N	1	74.60833333	33.12394,-117.32296,5,N,1
5	2023-06-18 06:35:06.253000 U...	33.12394	-117.32296	5	N	1	76.39166667	33.12394,-117.32296,5,N,1
6	2023-06-18 06:31:06.290000 U...	33.12394	-117.32296	5	N	1	76.36666667	33.12394,-117.32296,5,N,1
7	2023-06-18 06:33:34.302000 U...	33.12394	-117.32296	5	N	1	75.05833333	33.12394,-117.32296,5,N,1
8	2023-06-18 06:27:06.219000 U...	33.12394	-117.32296	5	N	1	74.475	33.12394,-117.32296,5,N,1

After streaming into a BigQuery table has been initiated, you can review the data in BigQuery by querying the table receiving the streaming data.

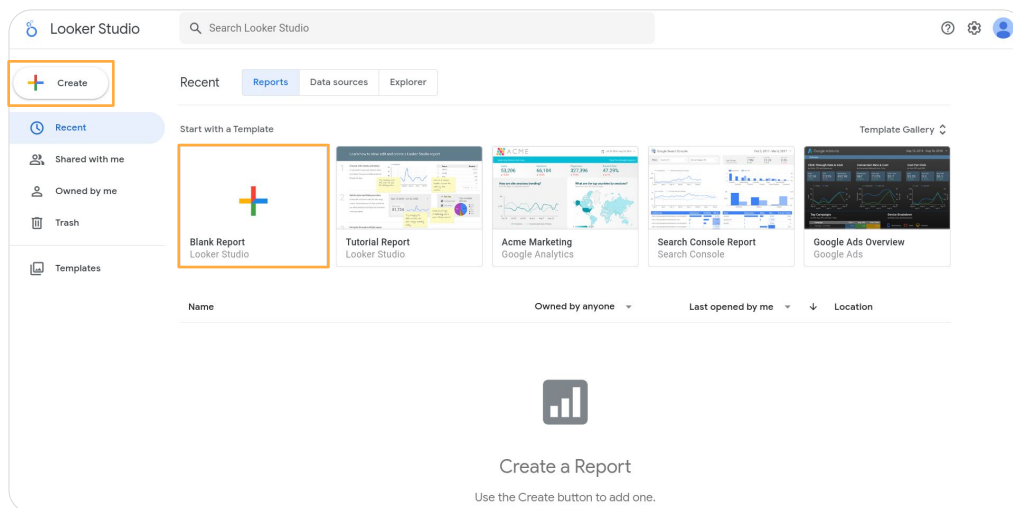
Want to visualize insights? Explore Looker Studio insights right from within BigQuery

The screenshot shows the Google Cloud BigQuery interface. At the top, there's a query editor with a SQL query: `SELECT * FROM `demos.average_speeds``. Below the editor, the 'Query results' section is visible. It includes tabs for 'JOB INFORMATION', 'RESULTS' (selected), 'JSON', 'EXECUTION DETAILS', and 'EXECUTION GRAPH'. The 'RESULTS' tab displays a table with 8 rows of data. The columns are: Row, timestamp, latitude, longitude, highway, direction, lane, speed, and sensorid. The 'EXPLORE DATA' button is highlighted with an orange box.

Row	timestamp	latitude	longitude	highway	direction	lane	speed	sensorid
1	2023-06-18 06:20:34.710000 U...	33.12394	-117.32296	5	N	1	73.8	33.12394,-117.32296,5,N,1
2	2023-06-18 06:30:07.033000 U...	33.12394	-117.32296	5	N	1	75.65833333	33.12394,-117.32296,5,N,1
3	2023-06-18 06:21:34.590000 U...	33.12394	-117.32296	5	N	1	75.00833333	33.12394,-117.32296,5,N,1
4	2023-06-18 06:20:35.076000 U...	33.12394	-117.32296	5	N	1	74.60833333	33.12394,-117.32296,5,N,1
5	2023-06-18 06:35:06.253000 U...	33.12394	-117.32296	5	N	1	76.39166667	33.12394,-117.32296,5,N,1
6	2023-06-18 06:31:06.290000 U...	33.12394	-117.32296	5	N	1	76.36666667	33.12394,-117.32296,5,N,1
7	2023-06-18 06:33:34.302000 U...	33.12394	-117.32296	5	N	1	75.05833333	33.12394,-117.32296,5,N,1
8	2023-06-18 06:27:06.219000 U...	33.12394	-117.32296	5	N	1	74.475	33.12394,-117.32296,5,N,1

When working with data in BigQuery, including streaming data, you can use Looker Studio to explore the data further. After you execute a query, you can choose Looker Studio from the Explore Data options to immediately start creating visualizations as part of a dashboard.

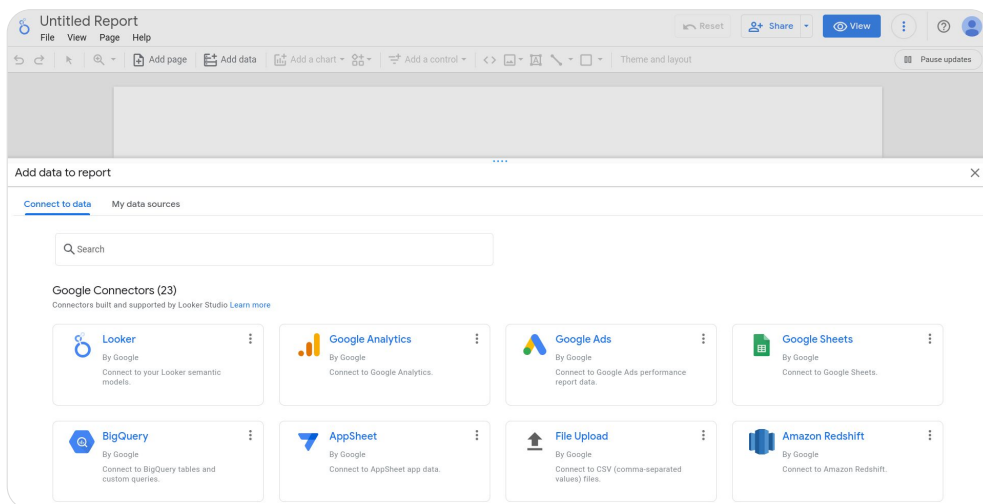
Create new reports in the Looker Studio UI



This is the Looker Studio Home page. There are two ways to create a new report from scratch:

- Select **Blank Report** in the templates panel in the middle of the screen.
- Or click the **Create** button in the navigation pane on the left of the screen.

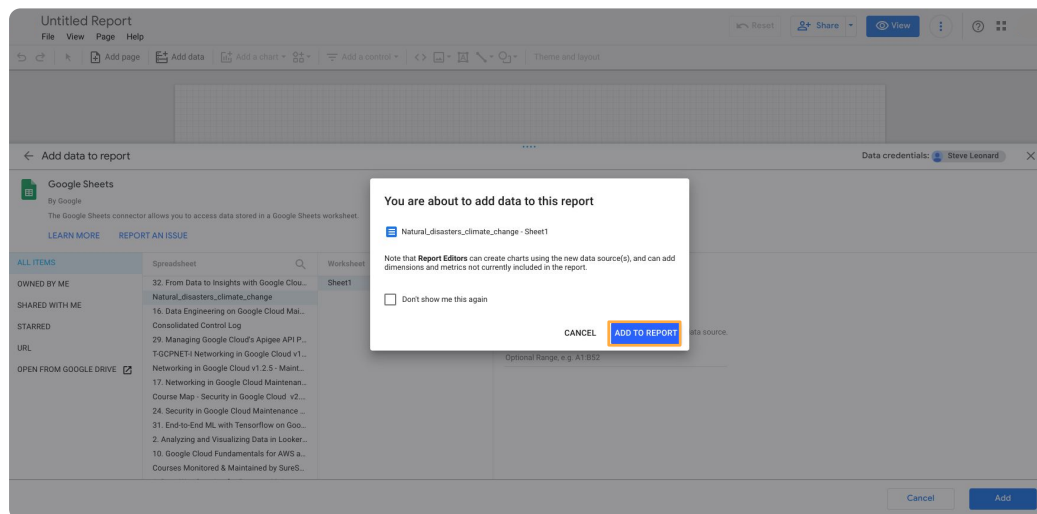
Connect to multiple different types of data sources



Google Cloud

Note that you can have any or all of these data sources in a single Looker Studio report. In addition to the Google Connectors, there is an increasing list of Partner Connectors to choose from as well.

Add the data source to your report

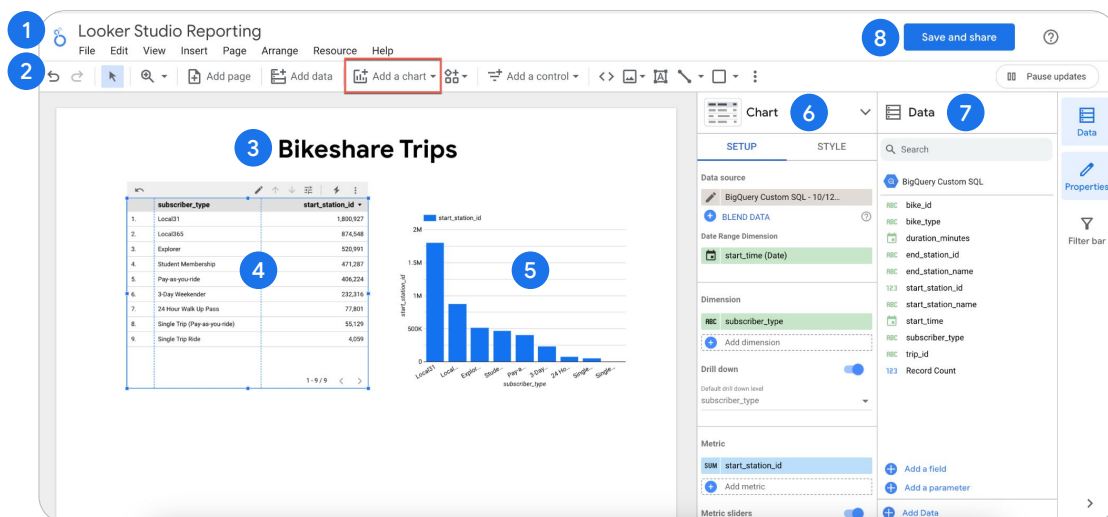


Since Looker Studio reports can be shared, you should be aware of the ramifications of adding a data source.

When you add a data source to a report, other people who can view the report can potentially see all the data in that data source. And anyone who can edit the report can use all the fields from any added data sources to create new charts with them.

Click ADD TO REPORT

Looker Studio layout and features



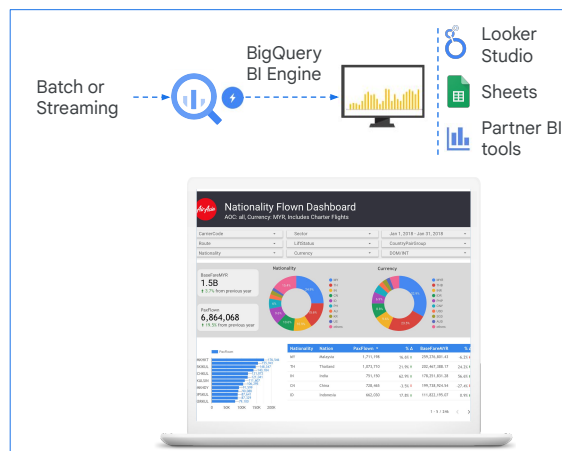
Google Cloud

The Looker Studio main window has a lot of features:

1. **Looker Studio logo and report name.**
 - To go to the Looker Studio page, click the logo.
 - To edit the report name, click the name.
2. **Looker Studio toolbar.** The Add a chart tool is highlighted.
3. **Report title.** To edit the text, click the text box.
4. **Table** (selected in the image). You can interact with a selected chart by using the options in the chart header. Tables are types of Charts.
5. **Bar Chart** (not selected in the image).
6. **Chart properties pane.** For a selected table, you can configure its data properties and appearance on the Setup and Style tabs.
7. **Data pane.** In this pane, you can access the fields and data sources to use in your report.
 - To add data to a chart, drag fields from the Data pane onto the chart.
 - To create a chart, drag a field from the Data pane onto the canvas.
8. **Save and share.** Save this report so you can view, edit, and share it with others later. Before you save the report, review the data source settings and the credentials that the data sources use.

Add value: BI Engine for dashboard performance

- No need to manage OLAP cubes or separate BI servers for dashboard performance.
- Natively integrates with BigQuery streaming for real-time data refresh.
- Column oriented in-memory BI execution engine.



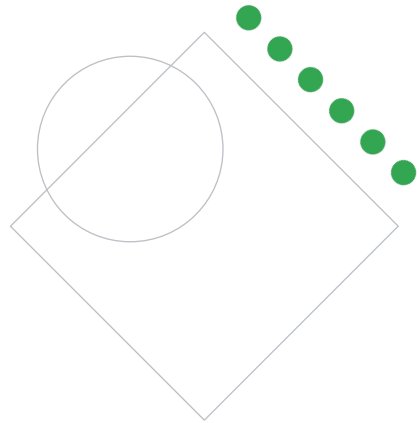
Google Cloud

One of the Google Cloud products that helps manage the performance of dashboards is BigQuery BI Engine. [BI Engine](#) is a fast, in-memory analysis service that is built directly into BigQuery and available to speed up your business intelligence applications.

Historically, BI teams would have to build, manage, and optimize their own BI servers and OLAP cubes to support reporting applications. With BI Engine, you can get sub-second query response time on your BigQuery datasets without having to create your own cubes. BI Engine is built on top of the same BigQuery storage and compute architecture and servers as a fast in-memory intelligent caching service that maintains state.

Lab Intro

Streaming Data Processing:
Streaming Analytics and Dashboards



In this Streaming Analytics and Dashboards lab, you will connect to a BigQuery data source from Looker Studio, and create reports and charts to visualize the BigQuery data.



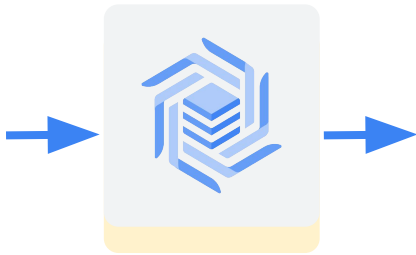
High-Throughput Streaming with Cloud Bigtable

So far, we looked at how to do queries on data even if it's streaming in using BigQuery, and displaying the data using Data Studio. BigQuery is a very good general purpose solution, something that would work in most cases. But every once in a while, you will come across a situation where the latency of BigQuery is going to be problematic.

In BigQuery, the data that's streaming in is available in a matter of seconds, but sometimes you will want lower latency than that. You will want your information to be available in milliseconds or microseconds. You may also have run into issues where the throughput of BigQuery may not be enough and you may want to deal with a higher throughput.

So what we will be looking at next is how to handle such throughput or latency requirements when BigQuery is not enough. Where do you go? We will talk about Cloud Bigtable, which is suited to high-performance applications. We will look at how to design for Bigtable, specifically how to design schemas, how to design the row key from Bigtable. We will look at how to ingest data into Bigtable.

Cloud Bigtable

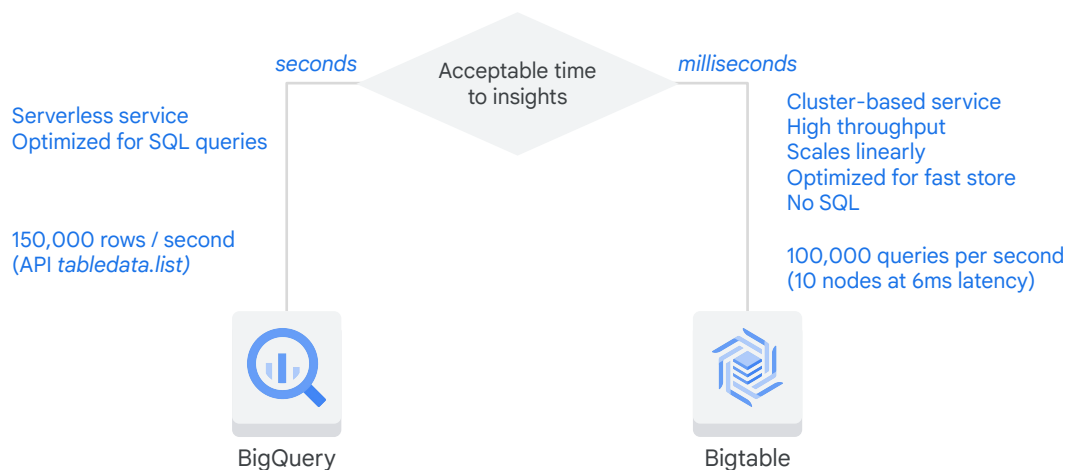


Qualities that Bigtable contributes to data engineering solutions:

- ✓ NoSQL Queries over large datasets (petabytes) in milliseconds
- ✓ Very fast for specific cases

To use Bigtable effectively you have to know a lot about your data and how it will be queried up-front. A lot of the optimizations happen before you load data into Bigtable.

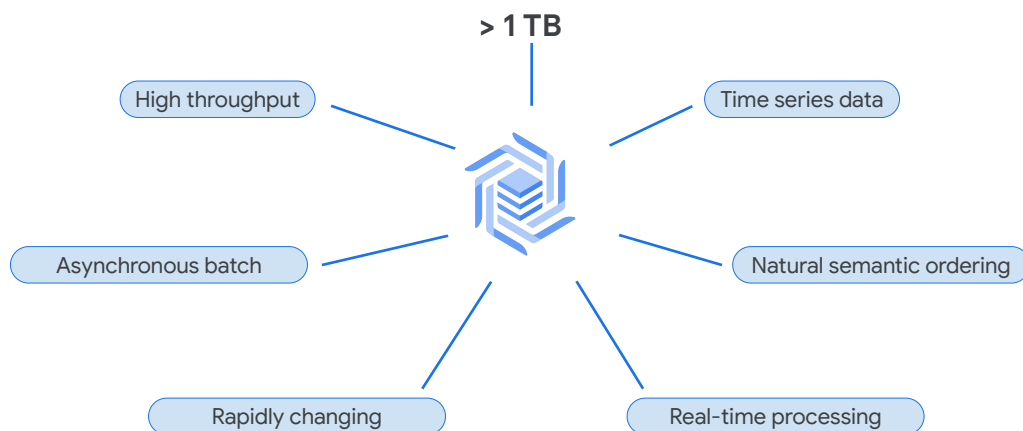
How to choose between Bigtable and BigQuery



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 well suited for highly structured data, transactional data, small data volumes less than 1 TB, and anything requiring SQL Queries and SQL-like joins.

Consider Bigtable for these requirements



Here are a few examples of data engineering requirements that have been solved using Bigtable.

Machine learning algorithms frequently have many or all of these requirements. Applications that use marketing data, such as purchase histories or customer preferences. Applications that use financial data such as transaction histories, stock prices, or currency exchange rates. Internet of Things - IoT data, such as usage reports from meters, sensors, or devices.

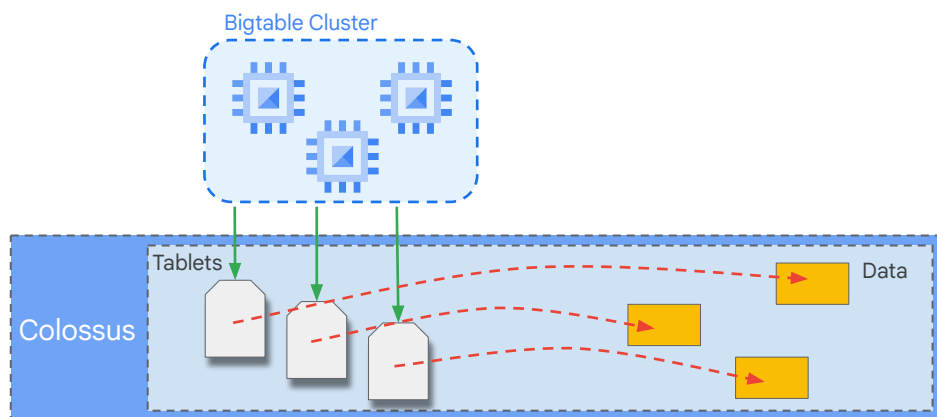
Time-series data, such as resource consumption like CPU and memory usage over time for multiple servers.

The most common use of Bigtable

Productionize a real-time lookup as part of an application, where speed and efficiency are desired beyond that of other databases.

Bigtable is most often used in real-time lookup capacity for an application where high-throughput is a necessity.

How does Bigtable work?



Bigtable stores data in a file system called Colossus. Colossus also contains data structures called Tablets that are used to identify and manage the data. And metadata about the Tablets is what is stored on the VMs in the Bigtable cluster itself.

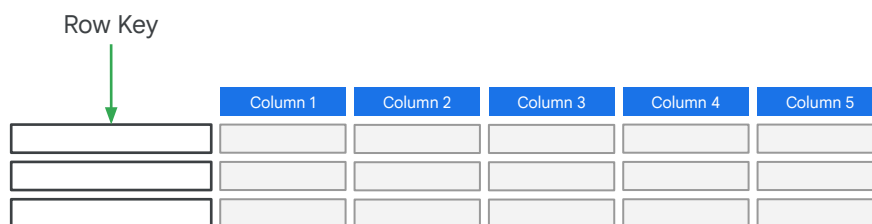
This design provides amazing qualities to Bigtable. It has three levels of operation. It can manipulate the actual data. It can manipulate the Tablets that point to and describe the data. Or it can manipulate the metadata that points to the Tablets. Rebalancing tablets from one node to another is very fast, because only the pointers are updated.

Bigtable is a learning system. It detects "hot spots" where a lot of activity is going through a single Tablet and splits the Tablet in two. It can also rebalance the processing by moving the pointer to a Tablet to a different VM in the cluster. So its best use case is with big data -- above 300 GB -- and very fast access but constant use over a longer period of time. This gives Bigtable a chance to learn about the traffic pattern and rebalance the Tablets and the processing.

When a node is lost in the cluster, no data is lost. And recovery is fast because only the metadata needs to be copied to the replacement node. Colossus provides better durability than the default 3 replicas provided by HDFS.

Bigtable design idea is "simplify for speed"

Row Key



The diagram illustrates a Bigtable structure. A green arrow labeled 'Row Key' points to the first column of a table. The table has five columns, labeled 'Column 1' through 'Column 5' in blue headers. There are three rows of data, each represented by a white box with a black border. The first row is highlighted with a green border, indicating it is the row selected by the Row Key.

	Column 1	Column 2	Column 3	Column 4	Column 5
Row Key					

The Row Key is the index.
And you get only one.

Bigtable stores the actual data elements in tables. And to begin with, it is just a table with rows and columns. However, unlike other table-based data systems like spreadsheets and SQL databases, Bigtable has only one index.

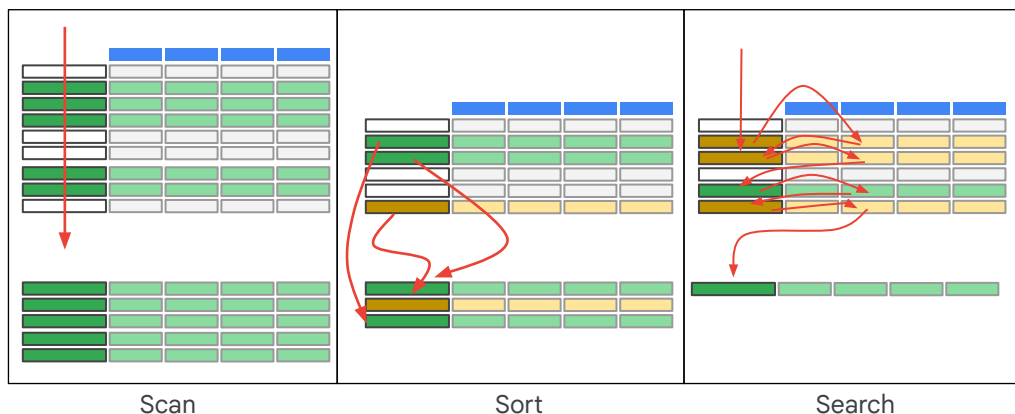
That index is called the Row Key. There are no alternate indexes or secondary indexes. And when data is entered, it is organized lexicographically by the Row Key.

The design principle of Bigtable is speed through simplification. If you take a traditional table, and simplify the controls and operations you allow yourself to perform on it, then you can optimize for those specific tasks.

It is the same idea behind RISC - Reduced Instruction Set Computing - Simplify the operations. And when you don't have to account for variations, you can make those that remain very fast.

In Bigtable, the first thing we must abandon in our design is SQL. This is a standard of all the operations a database can perform. And to speed things up we will drop most of them and build up from a minimal set of operations. That is why Bigtable is called a NoSQL database.

But speed depends on your data and Row Key



Scan

The green items are the results you want to produce from the query. In the best case you are going to scan the Row Key one time, from the top-down. And you will find all the data you want to retrieve in adjacent and contiguous rows. You might have to skip some rows. But the query takes a single scan through the index from top-down to collect the result set.

Sort

The second instance is sorting. You are still only looking at the Row Key. In this case the yellow line contains data that you want, but it is out of order. You can collect the data in a single scan, but the solution set will be disordered. So you have to take the extra step of sorting the intermediate results to get the final results. Now think about this. What does the additional sorting operation do to timing? It introduces a couple of variables. If the solution set is only a few rows, then the sorting operation will be quick. But if the solution set is huge, the sorting will take more time. The size of the solution set becomes a factor in timing. The orderliness of the original data is another factor. If most of the rows are already in order, there will be less manipulation required than if there are many rows out of order. The orderliness of the original data becomes a factor in timing. So introducing sorting means that the time it takes to produce the result is much more variable than scanning.

Search

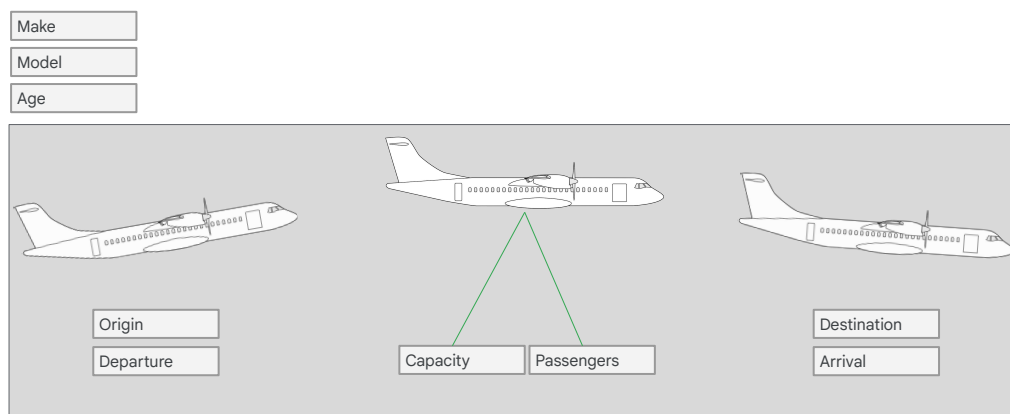
The third instance is searching. In this case, one of the columns contains critical data. You can't tell whether a row is a member of the solution set or not without examining

the data contained in the critical column. The Row Key is no longer sufficient. So now you are bouncing back and forth between Row Key and column contents. There are many approaches to searching. You could divide it up into multiple steps, one scan through the Row Keys and subsequent scans through the columns, and then perhaps a final sort to get the data in the order you want. And it gets much more complicated if there are multiple columns containing critical information. And it gets more complicated if the conditions of solution set membership involve logic such as a value in one column AND a value in another column, or a value in one column OR a value in another column. However, any algorithm or strategy you use to produce the result is going to be slower and more variable than scanning or sorting.

What is the lesson from this exploration? To get the best performance with the design of the Bigtable service, you need to get your data in order first, if possible, and you need to select or construct a Row Key that minimizes sorting and searching and turns your most common queries into scans.

Not all data and not all queries are good use cases for the efficiency that the Bigtable service offers. But when it is a good match, Bigtable is so consistently fast that it is magical.

Flights of the world: Reviewing the data



Google Cloud

Here is an example using airline flight data.

Each entry records the occurrence of one flight.

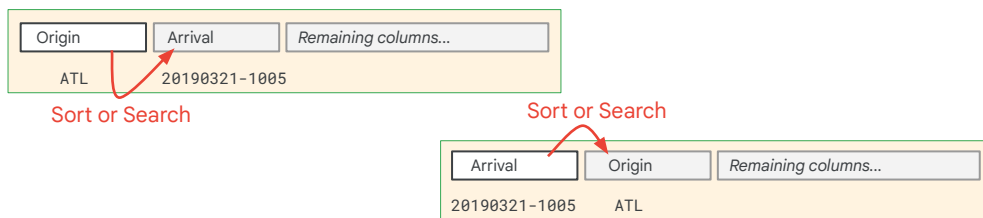
The data include city of origin and the date and time of departure, and destination city and date and time of arrival.

Each airplane has a maximum capacity, and related to this is the number of passengers that were actually aboard each flight.

Finally, there is information about the aircraft itself, including the manufacturer (called the make), the model number, and the current age of the aircraft at the time of the flight.

What is the best Row Key?

Query: All flights originating in Atlanta and arriving between March 21st and 29th



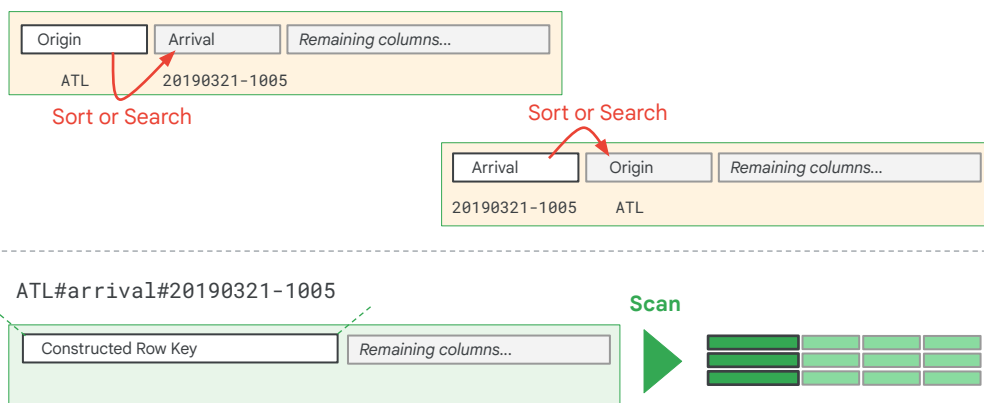
In this example, the Row Key will be defined for the most common use case. The Query is to find all flights originating from the Atlanta airport and arriving between March 21st and 29th. The airport where the flight originates is in the Origin field. And the date when the aircraft landed is listed in the Arrival field.

If you use Origin as the Row Key, you will be able to pull out all flights from Atlanta -- but the Arrival field will not necessarily be in order. So that means searching through the column to produce the solution set.

If you use the Arrival field as the Row Key, it will be easy to pull out all flights between March 21st and 29th, but the airport of origin won't be organized. So you will be searching through the arrival column to produce the solution set.

What is the best Row Key?

Query: All flights originating in Atlanta and arriving between March 21st and 29th



In the third example, a Row Key has been constructed from information extracted from the Origin field and the Arrival field -- creating a constructed Row Key. Because the data is organized lexicographically by the Row Key, all the Atlanta flights will appear in a group, and sorted by date of arrival. Also, with the word "arrival" present in the key there is no need to verify that the timestamp is for the Arrival rather than the Departure. Using this Row Key you can generate the solution set with only a scan.

In this example, the data was transformed when it arrived. So constructing a Row Key during the transformation process is straightforward.

Bigtable schema organization



Row Key	Column Families								
	Flight_Information					Aircraft_Information			
	Origin	Destination	Departure	Arrival	Passengers	Capacity	Make	Model	Age
ATL#arrival#20190321-1121	ATL	LON	20190321-0311	20190321-1121	158	162	B	737	18
ATL#arrival#20190321-1201	ATL	MEX	20190321-0821	20190321-1201	187	189	B	737	8
ATL#arrival#20190321-1716	ATL	YVR	20190321-1014	20190321-1716	201	259	B	757	23

Bigtable also provides Column Families. By accessing the Column Family, you can pull some of the data you need without pulling all of the data from the row or having to search for it and assemble it. This makes access more efficient.

Queries that use the row key, a row prefix, or a row range are the most efficient

Query: Current arrival delay for flights from Atlanta

1

row key based on atlanta arrivals
e.g. ORIGIN#arrival
(ATL#arrival#20190321-1005)

Puts latest flights at bottom
of table

2

reverse timestamp to the rowkey
e.g. ORIGIN#arrival#RTS
(ATL#arrival#560549313)

Puts latest flights at top
of table

The most common query is for the current arrival delay from Atlanta. That will involve averaging flight delays over the last 30 minutes. Hence, origin arrival. We want this at the top of the table, hence the reverse timestamp or RTS.

Use reverse timestamps when your most common query is for the latest values

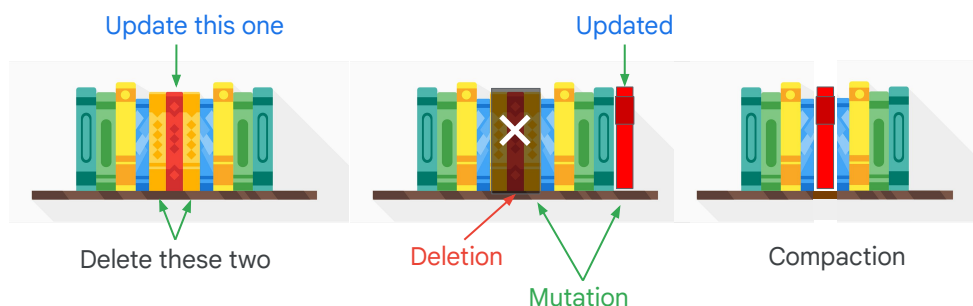
Query: Current arrival delay for flights from Atlanta

```
// key is ORIGIN#arrival#REVTS
String key = info.getORIGIN() //
    + "#arrival" //
    + "#" + (Long.MAX_VALUE - ts.getMillis()); // reverse timestamp
```

You can reverse timestamps by subtracting the timestamp from your programming language's maximum value for long integers, such as Java's `java.lang.Long.MAX_VALUE`, for example `LONG MAX timestamp.millisecondsSinceEpoch`.

By reversing the timestamp, you can design a row key where the most recent event appears at the start of the table instead of the end. As a result, you can get the N most recent events simply by retrieving the first N rows of the table.

What happens when data in Bigtable is changed?

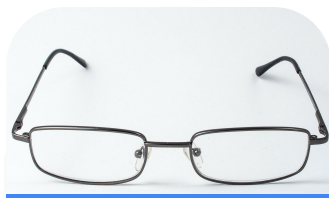


When you delete data, the row is marked for deletion and skipped during subsequent processing. It is not immediately removed.

If you make a change to data, the new row is appended sequentially to the end of the table, and the previous version is marked for deletion. So both rows exist for a period of time.

Periodically, Bigtable compacts the table, removing rows marked for deletion and reorganizing the data for read and write efficiency.

Optimizing data organization for performance



Group related data for more efficient reads



Distribute data evenly for more efficient writes



Place identical values in the same row or adjoining rows for more efficient compression

Example row key:

DehliIndia#2019031411841

Use column families

Use row keys to organize identical data

Google Cloud

Distributing the writes across nodes provides the best write performance. One way to accomplish this is by choosing row keys that are randomly distributed.

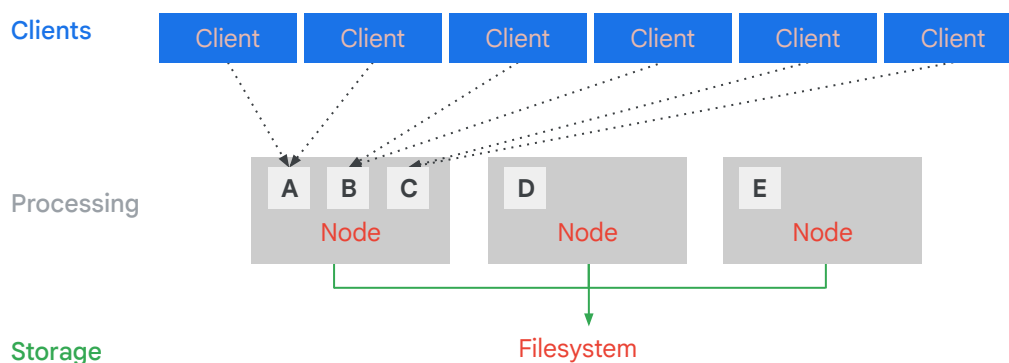
However, choosing a row key that groups related rows so they are adjacent makes it much more efficient to read multiple rows at one time.

In our airline example, if we were collecting weather data from the airport cities, we might construct a key consisting of a hash of the city name along with a timestamp. The example row key shown would enable pulling all the data for Delhi, India, as a contiguous range of rows.

Whenever there are rows containing multiple column values that are related, it is a good idea to group them into a column family. Some NoSQL databases suffer performance degradation if there are too many column families. Bigtable can handle up to 100 column families without losing performance. And it is much more efficient to retrieve data from one or more column families than retrieving all of the data in a row.

There are currently no configuration settings in Bigtable for compression. However, random data cannot be compressed as efficiently as organized data. 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 data are adjacent, the data can be compressed more efficiently.

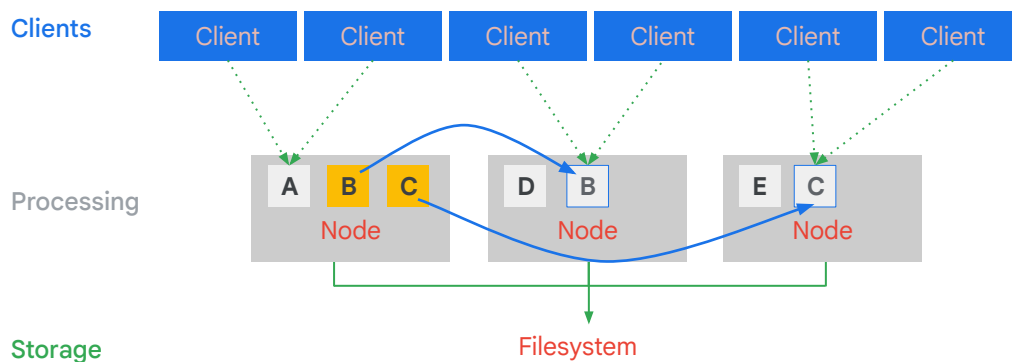
Bigtable self-improves by learning access patterns ...



Bigtable periodically rewrites your table to remove deleted entries, and to reorganize your data so that reads and writes are more efficient. It tries to distribute reads and writes equally across all Bigtable nodes.

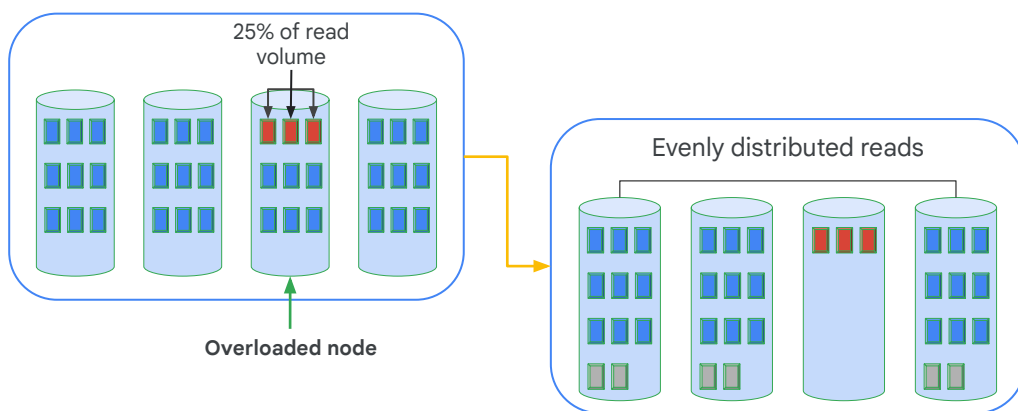
In this example, A, B, C, D, E are not data, but rather pointers or references and cache, which is why re-balancing is not time-consuming. We are just moving pointers. Actual data is in tablets in the Colossus file system.

...and rebalances data accordingly



Based on the learned access patterns, Bigtable re-balances data accordingly, and balances the workload across the nodes.

Rebalance strategy: Distribute reads



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, 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.

Real world use case: Spotify



In 2019, Spotify ran the largest Dataflow job ever at the time with Bigtable "...used as a remediation tool between Dataflow jobs in order for them to process and store more data in a parallel way, rather than the need to always regroup the data"



Google Cloud

In 2019 Spotify ran the largest Dataflow job ever at the time with Bigtable "...used as a remediation tool between Dataflow jobs in order for them to process and store more data in a parallel way, rather than the need to always regroup the data"

By using Bigtable, Spotify was able to break down Dataflow jobs into smaller components — and reusing core functionality — and was able to speed up jobs and make them more resilient.

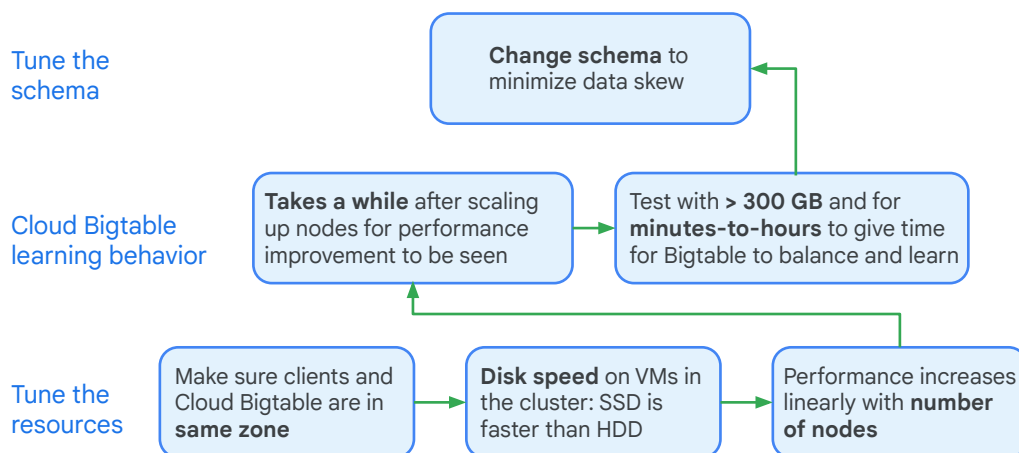
[\[https://techcrunch.com/2020/02/18/how-spotify-ran-the-largest-google-dataflow-job-ever-for-wrapped-2019/\]](https://techcrunch.com/2020/02/18/how-spotify-ran-the-largest-google-dataflow-job-ever-for-wrapped-2019/)



Optimizing Cloud Bigtable Performance

We will look now at how you can further optimize Bigtable performance.

Optimizing Bigtable performance



Google Cloud

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 Bigtable cluster. Otherwise, individual nodes can get overloaded, slowing performance.

The workload isn't appropriate for Bigtable. If you are testing with a small amount (less than 300 gigabytes) of data, or for a very short period of time (seconds rather than minutes or hours), 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 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 Bigtable cluster was scaled up very recently. While nodes are available in your cluster almost immediately, Bigtable can take up to 20 minutes under load to optimally distribute cluster workload across the new nodes.

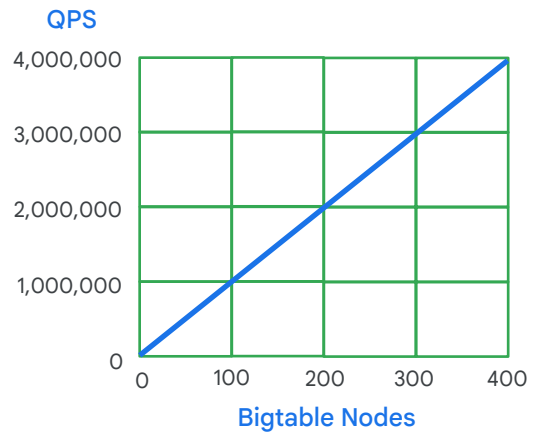
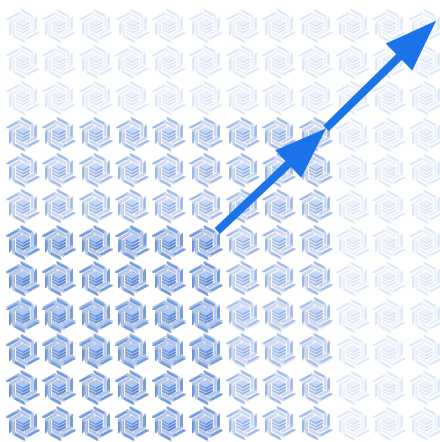
The Bigtable cluster uses HDD disks. Using HDD disks instead of SSD disks means slower response times and a significantly lower cap on the number of read requests handled per second (500 QPS for HDD disks versus 10,000 QPS for SSD).

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 Bigtable cluster.

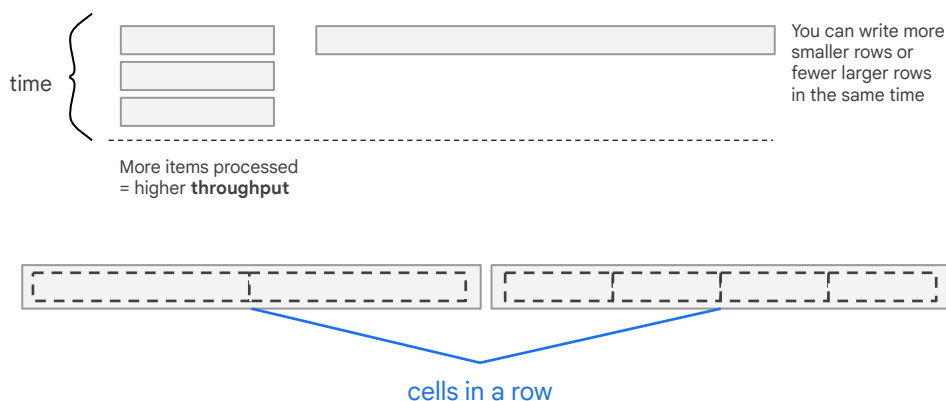
Because different workloads can cause performance to vary, you should perform tests with your own workloads to obtain the most accurate benchmarks.

Throughput can be controlled by node count



This is an example of some of the numbers that are possible in terms of throughput. With 100 nodes, you can handle 1 million queries per second. Throughput scales linearly well into the hundreds of nodes.

Schema design is the primary control for streaming



Google Cloud

A higher throughput means more items are processed in a given amount of time. If you have larger rows, then fewer of them will be processed in the same amount of time. In general, smaller rows offer higher throughput, and therefore are better for streaming performance.

Bigtable takes time to process cells within a row. So if there are fewer cells within a row, it will generally provide better performance than more cells.

Finally, selecting the right row key is critical. Rows are sorted lexicographically. The goal when optimizing for streaming is to avoid creating hotspots when writing, which would cause Bigtable to have to split tablets and adjust loads. To accomplish that, you want the data to be as evenly distributed as possible.

Reading delays, adding to processing delays leads to response time.

[\[https://cloud.google.com/bigtable/docs/performance\]](https://cloud.google.com/bigtable/docs/performance)

Use Bigtable replications to improve availability

Why perform replication?

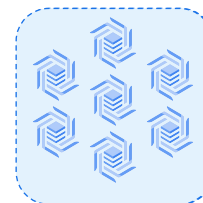
- Isolate serving applications from batch reads
- Improve availability
- Provide near-real-time backup
- Ensure your data has a global presence

```
gcloud bigtable clusters create CLUSTER_ID \
  --instance=INSTANCE_ID \
  --zone=ZONE \
  [--num-nodes=NUM_NODES] \
  [--storage-type=STORAGE_TYPE]
```

Batch analytic
read-only Cluster



App Traffic Cluster



Google Cloud

Replication for Bigtable enables you to increase the availability and durability of your data by copying it across multiple regions or multiple zones within the same region. You can also isolate workloads by routing different types of requests to different clusters.

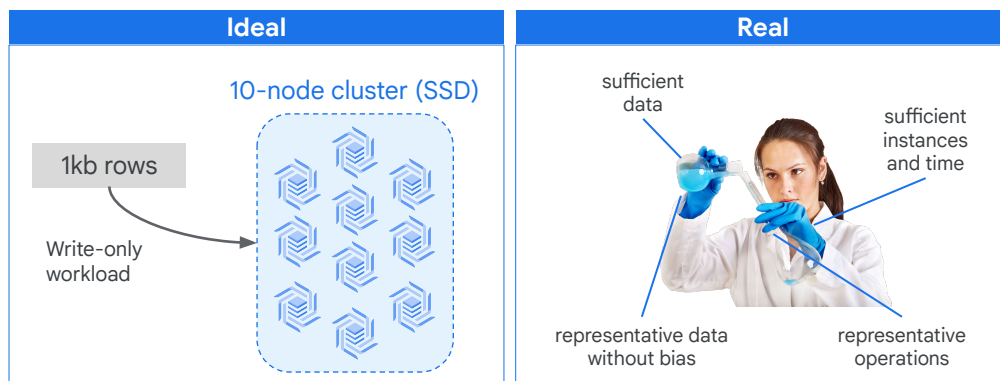
Use **gcloud bigtable clusters create** to create a cluster of Bigtable replicas.

If a Bigtable cluster becomes unresponsive, replication makes it possible for incoming traffic to failover to another cluster in the same instance. Failovers can be either manual or automatic, depending on the app profile an application is using and how the app profile is configured.

The ability to create multiple clusters in an instance is valuable for performance, as one can be for writing and the replica cluster exclusively for reading. Bigtable also supports automatic failover for High Availability.

[\[https://cloud.google.com/bigtable/docs/replication-overview\]](https://cloud.google.com/bigtable/docs/replication-overview)

Run performance tests carefully for Bigtable streaming



Google Cloud

The generalizations of isolate the write workload, increase number of nodes, and decrease row size and cell size will not apply in all cases. In most circumstances, experimentation is the key to defining the best solution.

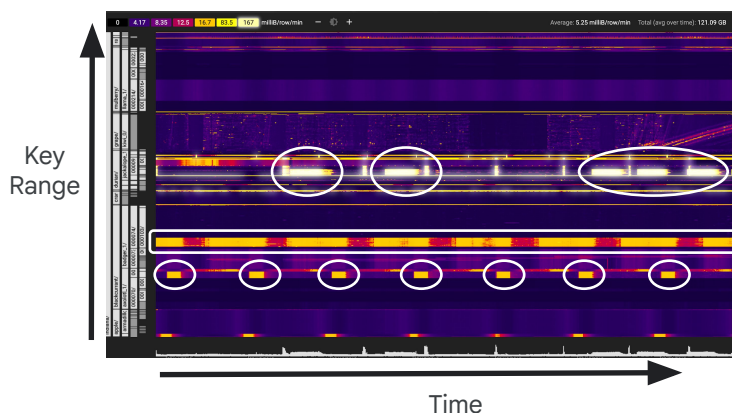
A performance estimate is given in the documentation online for write-only workloads. Of course, the purpose of writing data is to eventually read it, so the baseline is an ideal case.

At the time of this writing, a 10-node SSD cluster with 1 kilobyte rows and a write-only workload can process 10,000 rows per second at a 6 millisecond delay. This estimate will be affected by average row size, the balance and timing of reads distracting from writes, and other factors.

You will want to run performance tests with your actual data and application code. You need to run the tests on at least 300 gigabytes of data to get valid results. Also, to get valid results your test needs to perform enough actions over a long enough period of time to give Bigtable the time and conditions necessary to learn the usage pattern and perform its internal optimizations.

[<https://cloud.google.com/bigtable/docs/performance>]

Key Visualizer exposes read/write access patterns over time and key space



- find/prevent hotspots
- find rows with too much data
- see if your key schema is balanced

Google Cloud

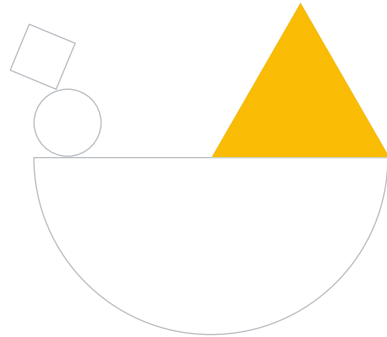
Key Visualizer is a tool that helps you analyze your Bigtable usage patterns. It generates visual reports for your tables that break down your usage based on the row keys that you access. Key Visualizer automatically generates hourly and daily scans for every table in your instance that meets at least one of the following criteria:

- During the previous 24 hours, the table contained at least 30 gigabytes of data at some point in time.
- During the previous 24 hours, the average of all reads or all writes was at least 10,000 rows per second.

The core of a Key Visualizer scan is the heatmap, which shows the value of a metric over time, broken down into contiguous ranges of row keys. The x-axis of the heatmap represents time, and the y-axis represents row keys. If the metric has a low value for a group of row keys at a point in time, the metric is "cold," and it appears in a dark color. A high value is "hot," and it appears in a bright color; the highest values appear in white.

Lab Intro

Streaming Data Processing: Streaming
Data Pipelines into Bigtable



Next, is the second hands-on lab for this module - Streaming Data Pipelines into Bigtable. In this lab, you will launch a Dataflow pipeline to read from PubSub and write into Bigtable, and open an HBase shell to query the Bigtable database.