## **BigTable**

I have no professional experience working with BigTable so the knowledge that I have was mostly theoretical.

* There were 2 questions related to Row key performance and how you can update your cluster if the performance is not optimal due to high reads or write.
* How you can scale your cluster and synchronize the data.
* Single-cluster routing and multi-cluster routing.
* Key Visualizer Metrics
* Schema design, such as when to use tall and narrow tables or short and wide ones
* Schema that might cause slow performance and how to optimize performance
* When to use hard disk drive (HDD)
* How to switch between HDD and solid-state drive (SSD)

Cloud Bigtable stores data as unstructured columns in rows; each row has a row key, and row keys are sorted lexicographically.  
For time series, you should generally use tall and narrow tables. This is for two reasons:  
1) Storing one event per row makes it easier to run queries against your data.  
2) Storing many events per row makes it more likely that the total row size will exceed the recommended maximum.

[https://cloud.google.com/bigtable/docs/schema- design-time-series](https://cloud.google.com/bigtable/docs/schema-%C2%A0design-time-series)

Wide & short table schema is not optimal for time-series event data

BigTable is a **NoSQL database** by nature. Nonetheless, it supports SQL to query data. However, Bigtable is used if scaling is a critical issue. For this scenario, data is in CSV format and BigQuery is better structured to handle importing CSV data. While Bigtable requires extra prerequisites.

Cloud Bigtable is a sparsely populated table that can scale to billions of rows and thousands of columns, allowing you to store terabytes or even petabytes of data.  
**A single value in each row is indexed; this value is known as the row key**. Cloud Bigtable is ideal for storing very large amounts of **single-keyed** data with very low latency. It supports high read and write throughput at low latency, and it is an ideal data source for MapReduce operations.  
Reference: <https://cloud.google.com/bigtable/docs/overview>

Storing time-series data in Cloud Bigtable is a natural fit. Cloud Bigtable stores data as unstructured columns in rows; each row has a row key, and row keys are sorted lexicographically.  
For time series, you should generally use tall and narrow tables. This is for two reasons: Storing one event per row makes it easier to run queries against your data. Storing many events per row makes it more likely that the total row size will exceed the recommended maximum (see Rows can be big but are not infinite).  
When Cloud Bigtable stores rows, it sorts them by row key in lexicographic order. There is effectively a single index per table, which is the row key. Queries that access a single row, or a contiguous range of rows, execute quickly and efficiently. All other queries result in a full table scan, which will be far, far slower. A full table scan is exactly what it sounds like—every row of your table is examined in turn.  
For Cloud Bigtable, where you could be storing many petabytes of data in a single table, the performance of a full table scan will only get worse as your system grows.  
Choosing a row key that facilitates common queries is of paramount importance to the overall performance of the system. Enumerate your queries, put them in order of importance, and then design row keys that work for those queries.  
From the description, you need to combine both sensor ID and timestamp in order to fetch data you want fast. So, answers A & D are incorrect.  
If you start the row key with a timestamp, most recent data will be inserted at the bottom of the table since rows are sorted in lexicographic order. Starting the row key with sensor ID will allow writing all sensor’s events together and allow distributing data among nodes.

Reference(s):  
[https://cloud.google.com/bigtable/docs/schema- design-time-series](https://cloud.google.com/bigtable/docs/schema-%C2%A0design-time-series)

what is the recommended minimum amount of stored data?

Cloud Bigtable is not a relational database. It does not support SQL queries, joins, or multi-row transactions. It is not a good solution for **less than 1 TB of data**.  
Reference: <https://cloud.google.com/bigtable/docs/overview#title_short_and_other_storage_options>

Cloud Bigtable is **Google’s NoSQL Big Data database service**. It is the same database that Google uses for services, such as Search, Analytics, Maps, and Gmail.  
It is used for requirements that are low latency and high throughput including Internet of Things (IoT), user analytics, and financial data analysis.  
Reference: <https://cloud.google.com/bigtable/>

The HBase shell is a command-line tool that performs administrative tasks, such as creating and deleting tables. The Cloud Bigtable HBase client for Java makes it possible to use the HBase shell to connect to Cloud Bigtable.  
Reference: <https://cloud.google.com/bigtable/docs/installing-hbase-shell>

Data is not lost when cloud bigtable node is failed:

A Cloud Bigtable table is sharded into blocks of contiguous rows, called tablets, to help balance the workload of queries. Tablets are stored on Colossus, Google’s file system, in SSTable format. Each tablet is associated with a specific Cloud Bigtable node.  
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  
Reference: <https://cloud.google.com/bigtable/docs/overview>

The Cloud Bigtable cluster doesn’t have enough nodes. If your Cloud Bigtable cluster is overloaded, adding more nodes can improve performance. Use the monitoring tools to check whether the cluster is overloaded.  
Reference: <https://cloud.google.com/bigtable/docs/performance>

When you create a Cloud Bigtable instance, you choose whether its clusters store data on solid-state drives (SSD) or hard disk drives (HDD):  
SSD is significantly faster and has a more predictable performance than HDD.  
HDD throughput is much more limited than SSD throughput. In a cluster that uses HDD storage, it’s easy to reach the maximum throughput before CPU usage reaches 100%. To increase throughput, you must add more nodes, but the cost of the additional nodes can easily exceed your savings from using HDD storage. SSD storage does not have this limitation because it offers much more throughput per node.  
Individual row reads on HDD are very slow. Because of disk seek time, HDD storage supports only 5% of the read rows per second of SSD storage.  
The cost savings from HDD are minimal, relative to the cost of the nodes in your Cloud Bigtable cluster, unless you’re storing very large amounts of data.

If you’re running a performance test that depends upon Cloud Bigtable, be sure to follow these steps as you plan and execute your test:  
- Use a production instance. A development instance will not give you an accurate sense of how a production instance performs under load.  
- Use at least 300 GB of data. Cloud Bigtable **performs best with 1 TB or more** of data. **However, 300 GB of data is enough to provide reasonable results in a performance test on a 3-node cluster**. On larger clusters, use at least 100 GB of data per node.  
- Stay below the recommended storage utilization per node.  
- Before you test, run a heavy pre-test for several minutes. This step gives Cloud Bigtable a chance to balance data across your nodes based on the access patterns it observes.  
- Run your test for at least 10 minutes. This step lets Cloud Bigtable optimize your data, and it helps ensure that you will test reads from disk as well as cached reads from memory.

Bigtable is a ideal solution for storing time series data. Storing time-series data in Cloud Bigtable is a natural fit. Cloud Bigtable stores data as unstructured columns in rows; each row has a row key, and row keys are sorted lexicographically.

<https://cloud.google.com/bigtable/docs/replication-settings#batch-vs-serve>

The percentage of the cluster’s storage capacity that is being used. The capacity is based on the number of nodes in your cluster. In general, do not use more than 70% of the hard limit on total storage, so you have room to add more data. If you do not plan to add significant amounts of data to your instance, you can use up to 100% of the hard limit. Important: If any cluster in an instance exceeds the hard limit on the amount of storage per node, writes to all clusters in that instance will fail until you add nodes to each cluster that are over the limit. Also, if you try to remove nodes from a cluster, and the change would cause the cluster to exceed the hard limit on storage, Cloud Bigtable will deny the request. If you are using more than the recommended percentage of the storage limit, add nodes to the cluster. You can also delete existing data, but deleted data takes up more space, not less, until a compaction occurs.

Google recommends adding nodes when storage utilization is > 70% <https://cloud.google.com/bigtable/docs/modifying-instance#nodes>

**The storage for the cluster cannot be updated.** You need to define the new cluster and copy or import the data to it.

Switching between SSD and HDD storage  
When you create a Cloud Bigtable instance and cluster, your choice of SSD or HDD storage for the cluster is permanent. You cannot use the Google Cloud Platform Console to change the type of storage that is used for the cluster.  
If you need to convert an existing HDD cluster to SSD, or vice-versa, you can export the data from the existing instance and import the data into a new instance. Alternatively, you can use a Cloud Dataflow or Hadoop MapReduce job to copy the data from one instance to another. Keep in mind that migrating an entire instance takes time, and you might need to add nodes to your Cloud Bigtable clusters before you migrate your instance.

Bigtable Choosing HDD vs SSD:- <https://cloud.google.com/bigtable/docs/choosing-ssd-hdd>

* SSD storage is the most efficient and cost-effective choice for most use cases.
* HDD storage is sometimes appropriate for very large data sets (>10 TB) that are not latency-sensitive or are infrequently accessed.

The Cloud Dataflow connector for Cloud Bigtable makes it possible to use Cloud Bigtable in a Cloud Dataflow pipeline. You can use the connector for both batch and streaming operations.  
Reference: <https://cloud.google.com/bigtable/docs/dataflow-hbase>

Bigtable is an HBase managed service alternative on Google Cloud. However, it **does not support Coprocessors**. So the best solution is to use HBase with Dataproc which can be installed using initialization actions.

Bigtable HBase differences:- <https://cloud.google.com/bigtable/docs/hbase-differences#coprocessors>Coprocessors are not supported. You cannot create classes that implement the interfaceorg.apache.hadoop.hbase.coprocessor.

In a Cloud Bigtable architecture all **client requests go through a front-end server before they are sent to a Cloud Bigtable node.**  
The nodes are organized into a Cloud Bigtable cluster, which belongs to a Cloud Bigtable instance, which is a container for the cluster. Each node in the cluster handles a subset of the requests to the cluster.  
When additional nodes are added to a cluster, you can increase the number of simultaneous requests that the cluster can handle, as well as the maximum throughput for the entire cluster.  
Reference: <https://cloud.google.com/bigtable/docs/overview>

What is the general recommendation when designing your row keys for a Cloud Bigtable schema?

A general guide is to, keep your row keys reasonably short. Long row keys take up additional memory and storage and increase the time it takes to get responses from the Cloud Bigtable server.  
Reference: <https://cloud.google.com/bigtable/docs/schema-design#row-keys>

For Cloud Bigtable, you can configure access control at the project level. For example, you can grant the ability to:  
Read from, but not write to, any table within the project.  
Read from and write to any table within the project, but not manage instances.  
Read from and write to any table within the project, and manage instances.  
Reference: <https://cloud.google.com/bigtable/docs/access-control>

It is recommended to create your Compute Engine instance in the same zone as your Cloud Bigtable instance for the best possible performance,  
If it’s not possible to create a instance in the same zone, you should create your instance in another zone within the same region. For example, if your Cloud  
Bigtable instance is located in us-central1-b, you could create your instance in us-central1-f. This change may result in several milliseconds of additional latency for each Cloud Bigtable request.  
It is recommended to avoid creating your Compute Engine instance in a different region from your Cloud Bigtable instance, which can add hundreds of milliseconds of latency to each Cloud Bigtable request.  
Reference: <https://cloud.google.com/bigtable/docs/creating-compute-instance>

Which row keys are likely to cause a disproportionate number of reads and/or writes on a particular node in a Bigtable cluster

…using a timestamp as the first element of a row key can cause a variety of problems.  
In brief, when a row key for a time series includes a timestamp, all of your writes will target a single node; fill that node; and then move onto the next node in the cluster, resulting in hotspotting.  
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. [https:// cloud.google.com/bigtable/docs/schema-design]  
Reference: <https://cloud.google.com/bigtable/docs/schema-design-time-series#ensure_that_your_row_key_avoids_hotspotting>

The most common issue for time series in Cloud Bigtable is hotspotting. This issue can affect any type of row key that contains a monotonically increasing value. In brief, when a row key for a time series includes a timestamp, all of your writes will target a single node; fill that node; and then move onto the next node in the cluster, resulting in hotspotting. Because Cloud Bigtable stores adjacent row keys on the same server node, all writes will focus only on one node until that node is full, at which point writes will move to the next node in the cluster.  
BigTable instances are best performed with big size of data (over 1TB). Having data less than 1TB of size will not give the same performance.

By default, prefer field promotion. Field promotion avoids hotspotting in almost all cases, and it tends to make it easier to design a row key that facilitates queries.  
Reference: <https://cloud.google.com/bigtable/docs/schema-design-time-series#ensure_that_your_row_key_avoids_hotspotting>

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.  
Reference: <https://cloud.google.com/bigtable/docs/schema-design#row-keys>

For example, if you plan to store extensive historical data for a large number of remote-sensing devices and then use the data to generate daily reports, the cost savings for HDD storage may justify the performance tradeoff. On the other hand, if you plan to use the data to display a real-time dashboard, it probably would not make sense to use HDD storagereads would be much more frequent in this case, and reads are much slower with HDD storage.  
Reference: <https://cloud.google.com/bigtable/docs/choosing-ssd-hdd>

You can change cluster IDs only by deleting and recreating the cluster. Also, you cannot change the instance ID or the instance’s storage type, and you cannot downgrade a production instance to a development instance. To change any of these settings, you must create a new instance with your preferred settings; export your data from the old instance; import your data into the new instance; and delete the old instance.  
From the description above, the best solution is using Dataflow to migrate data from the old BigTable instance to the new one.  
All other answers are incorrect based on the description.

[https://cloud.google.com/bigtable/docs/modifying- instance](https://cloud.google.com/bigtable/docs/modifying-%C2%A0instance)

When you create a Cloud Bigtable instance, you choose whether its clusters store data on solid-state drives (SSD) or hard disk drives (HDD).HDD storage is suitable for use cases that meet the following criteria:  
 – You expect to store at least 10 TB of data.  
 – You will not use the data to back a user-facing or latency-sensitive application.  
 – Your workload falls into one of the following categories:  
 – Batch workloads with scans and writes, and no more than occasional random reads of a small   
 number of rows.  
 – Data archival, where you write very large amounts of data and rarely read that data.  
From the scenario, system logs are to be stored to BigTable. This data will be only used for occasional debugging and security anomaly detection. So, using the HDD storage type for BigTable is the answer.

<https://cloud.google.com/bigtable/docs/choosing-ssd-hdd>

If you no longer want to use a development instance for development and testing, you can upgrade it to a production instance at any time. Upgrading a development instance is permanent.  
A cluster must have enough nodes to support its current workload and the amount of data it stores. Otherwise, the cluster might not be able to handle incoming requests, and latency could go up.

<https://cloud.google.com/bigtable/docs/keyvis-overview>

*Key Visualizer for Bigtable generates visual reports for your tables that detail your usage based on the row keys that you access, show you how Bigtable operates, and can help you troubleshoot performance issues.*