Assignment 3.7

Problem Statement

1. What is NoSQL data base?

2. How does data get stored in NoSQl database?

3. What is a column family in HBase?

4. How many maximum number of columns can be added to HBase table?

5. Why columns are not defined at the time of table creation in HBase?

6. How does data get managed in HBase?

7. What happens internally when new data gets inserted into HBase table?

# **WHAT IS NOSQL?**

NoSQL is an approach to databases that represents a shift away from traditional relational database management systems (RDBMS). To define NoSQL, it is helpful to start by describing SQL, which is a query language used by RDBMS. Relational databases rely on tables, columns, rows, or schemas to organize and retrieve data. In contrast, NoSQL databases do not rely on these structures and use more flexible data models. NoSQL can mean “not SQL” or “not only SQL.” As RDBMS have increasingly failed to meet the performance, scalability, and flexibility needs that next-generation, data-intensive applications require, NoSQL databases have been adopted by mainstream enterprises. **NoSQL is particularly useful for storing unstructured data**, which is growing far more rapidly than structured data and does not fit the relational schemas of RDBMS. Common types of unstructured data include: user and session data; chat, messaging, and log data; time series data such as IoT and device data; and large objects such as video and images.

# How does data get stored in NoSQl database?

There are various NoSQL Databases. Each one uses a different method to store data. Some might use column store, some document, some graph, etc., Each database has its own unique characteristics.

**In the in-memory databases** like Redis/CouchBase/Tarantool/Aerospike everything is stored in RAM in balanced trees like RB-Tree or in hash tables. All the writes are applied on both RAM and disk, but on disk it goes in an append-only way. A file append can be done as fast as 100Mbytes per second on a normal magnetic disk. If a record size is, say, 1K, then the data will be written at 100krps.

**In the on-disk NoSQL databases and db-engines** like Cassandra/HBase/RocksDB/LevelDB/Sophia the main idea is that you have a snapshot file and a write ahead log (WAL) file. Snapshot contains already prepared data in a form of B-Tree with upper levels of that tree being permanently in RAM, that can be accesses for reading by doing only one disk seek. A WAL contains all the new changes on top of a current snapshot. A snapshot file is being totally rebuilt on a regular basis using current snapshot and a WAL. All the writes are done nearly as fast as with in-memory databases. "Nearly" because disk is partially busy by doing regular snapshot converting that was described earlier. Reads are significantly slower than that are in in-memory databases, because they take at least one disk seek, but good news is that they can be cached in optimized in-memory structures like RB-Trees/hash tables.

# What is a column family in HBase?

Column families are the base storage mechanism in HBase.   A HBase table is comprised of one or more column families,  each of which is stored in a separate set of region files sharing a common key.  
  
To express it in terms of an RDBMS, a column family is roughly analogous to a RDBMS table with the rowkey as a clustered primary key index.    A HBase table would then be a view which does a full outer join on a set of RDBMS tables which all share the same primary key (thus having a 1:1 relationship).    In this analogy, HBase region files map to pages in an RDBMS.

Columns in Apache HBase are grouped into *column families*. All column members of a column family have the same prefix. For example, the columns *courses:history* and *courses:math* are both members of the *courses* column family. The colon character (:) delimits the column family from the . The column family prefix must be composed of *printable* characters. The qualifying tail, the column family *qualifier*, can be made of any arbitrary bytes. Column families must be declared up front at schema definition time whereas columns do not need to be defined at schema time but can be conjured on the fly while the table is up an running.

Physically, all column family members are stored together on the filesystem. Because tunings and storage specifications are done at the column family level, it is advised that all column family members have the same general access pattern and size characteristics.

# How many maximum number of columns can be added to HBase table?

There is no limit on number of column families in HBase, in theory. In reality, there are several factors which can limit useable number of column families in HBase:

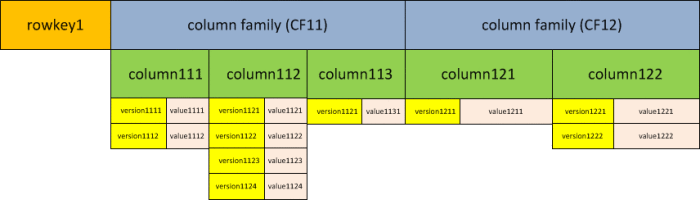
HBase Admin web UI usability. It will be very hard to show even 100s of column families in a table configuration page.

HDFS practical limit of maximum number of files. Say, 100m. If your table has N regions, M column families you will need NxM directories to support this configuration. Every region/column family, in turn, can contain up to K store files (depends on write load and many other configuration options). With very modest N = 100 and K = 10 we can say practical limit of maximum number of column families is less than 100K. Usually, much less than 100K.

Each column family has its own directory in HDFS and set of store files and, from performance point of view, the fewer directories (column families) you have the better performance for scan operations you get.

# Why columns are not defined at the time of table creation in HBase?

In Hbase columns are the final entity, which stores the data. HBase data model consists of tables containing rows. Data is organized into column families grouping columns in each row. This is where similarities between HBase and relational databases end. Now we will explain what is under the HBase table/rows/column families/columns hood.



This is to summarize an HBase table’s mappings:

* a row key maps to a list of column families
* a column family maps to a list of column qualifiers (columns)
* a column qualifier maps to a list of timestamps (versions)
* a timestamp maps to a value (the cell itself)

Based on this you will get the following:

* if you are retrieving data that a row key maps to, you’d get data from all column families related to the row that the row key identifies
* if you are retrieving data which a particular column family maps to, you’d get all column qualifiers and associated data (maps with timestamps as keys and corresponding values)
* if you are retrieving data that a particular column qualifier maps to, you’d get all timestamps (versions) for that column qualifier and all associated values.
* Tables are declared up front at schema definition time. Row keys are arrays of bytes and they are lexicographically sorted with the lowest order appearing first.

## Column

Columns are usually physically co-located in column families. A column is identified by column family and column qualifier separated by a colon character (:). For example, *courses:math*. The column family prefix must be composed of printable characters**. The column qualifiers (columns) do not have to be defined at schema definition time and they can be added on the fly while the database is up and running.**

A column qualifier is an index for a given data and it is added to a column family. Data within a column family is addressed via the column qualifier. Column qualifiers are mutable and they may vary between rows. They do not have data types and they are always treated as arrays of bytes.

A row key, column family and column qualifier form a cell that has a value and timestamp that represents the value’s version. Values also do not have data types and they are always treated as arrays of bytes. A timestamp is recorded for each value and it is the time on the region server when the value was written.

All cell’s values are stored in a descending order by its timestamp. When values are retrieved and if the timestamp is not provided then HBase will return the cell value with the latest (the most recent) timestamp. If a timestamp is not specified during the write, the current timestamp is used.

The maximum number of versions (timestamps) for a given column to store is part of the column schema. It is specified at table creation. It can be specified via alter table command as well. The default value is 1. The minimum number of versions can be also set up per column family. You can also globally set up a maximum number of versions per column.

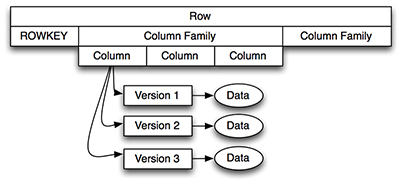
HBase does not overwrite row values. It stores different values per row by time and column qualifier. Extra versions above the current max version setup are removed during major compactions. If it is not necessary it is not recommended to have very high maximum number of versions since it will increase the HFile size significantly.

**It is worth to mention that the column metadata is only stored in internal key/value instances for a column family.** You have to keep track of the column names since HBase can support very high number of columns per row and columns can differ between the rows as well. If you do not record these column names by yourself and you forget them you will have to retrieve all rows from a column family in order to find out the column name

# How does data get managed in HBase?

HBase is not a relational database and requires a different approach to modeling your data. HBase actually defines a four-dimensional data model and the following four coordinates define each cell (see Figure 1):

* Row Key: Each row has a unique row key; the row key does not have a data type and is treated internally as a byte array.
* Column Family: Data inside a row is organized into column families; each row has the same set of column families, but across rows, the same column families do not need the same column qualifiers. Under-the-hood, HBase stores column families in their own data files, so they need to be defined upfront, and changes to column families are difficult to make.
* Column Qualifier: Column families define actual columns, which are called column qualifiers. You can think of column qualifiers as the columns themselves.
* Version: Each column can have a configurable number of versions, and you can access the data for a specific version of a column qualifier.

[](javascript:popUp('/content/images/art_haines_hbases1_1/elementLinks/haines_hbase_1_1_01.jpg'))

[**Figure 1.**](javascript:popUp('/content/images/art_haines_hbases1_1/elementLinks/haines_hbase_1_1_01.jpg'))  HBase Four-Dimensional Data Model

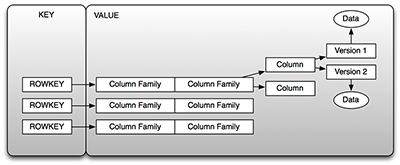
As shown in [Figure 1](javascript:popUp('/content/images/art_haines_hbases1_1/elementLinks/haines_hbase_1_1_01.jpg')), an individual row is accessible through its row key and is composed of one or more column families. Each column family has one or more column qualifiers (called “column” in [Figure 1](javascript:popUp('/content/images/art_haines_hbases1_1/elementLinks/haines_hbase_1_1_01.jpg'))) and each column can have one or more versions. To access an individual piece of data, you need to know its row key, column family, column qualifier, and version.

When designing an HBase data model, it is helpful to think about how the data is going to be accessed. You can access HBase data in two ways:

* Through their row key or via a table scan for a range of row keys
* In a batch manner using map-reduce

This dual-approach to data access is something that makes HBase particularly powerful. Typically, storing data in Hadoop means that it is good for offline or batch analysis (and it is very, very good at batch analysis) but not necessarily for real-time access. HBase addresses this by being both a key/value store for real-time analysis and supporting map-reduce for batch analysis.

Let’s first look at the real-time access. As a key/value store, the key is the row key, and the value is the collection of column families, as shown in [Figure 2](javascript:popUp('/content/images/art_haines_hbases1_1/elementLinks/haines_hbase_1_1_02.jpg')).

[](javascript:popUp('/content/images/art_haines_hbases1_1/elementLinks/haines_hbase_1_1_02.jpg'))

[**Figure 2.**](javascript:popUp('/content/images/art_haines_hbases1_1/elementLinks/haines_hbase_1_1_02.jpg'))  HBase as a Key/Value Store

As you can see in [Figure 2](javascript:popUp('/content/images/art_haines_hbases1_1/elementLinks/haines_hbase_1_1_02.jpg')), the key is the row key we have been talking about, and the value is the collection of column families (that have their associated columns that have versions of the data). You can retrieve the value associated with a key; or in other words, you can “get” the row associated with a row key, or you can retrieve a set of rows by giving the starting row key and ending row key, which is referred to as a table scan. You cannot query for values contained in columns in a real-time query, which leads to an important topic: row key design.

The design of the row key is important for two reasons:

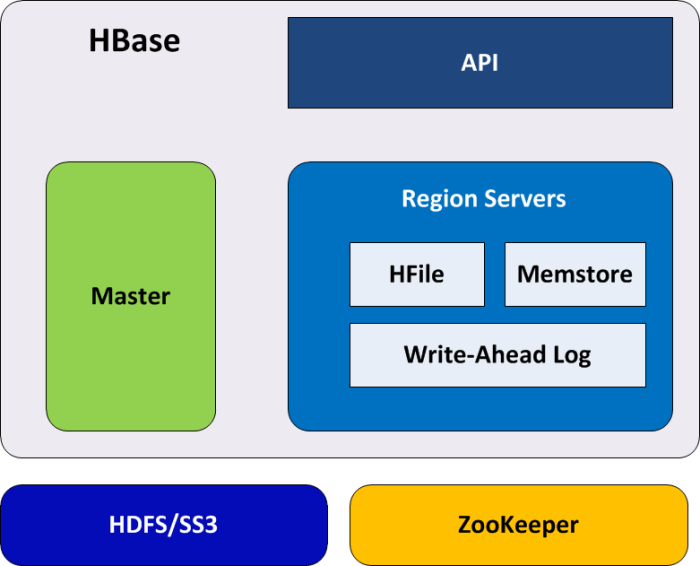
* Table scans operate against the row key, so the design of the row key controls how much real-time/direct access you can perform against HBase.
* When running HBase in a production environment, it runs on top of the Hadoop Distributed File System (HDFS) and the data is distributed across the HDFS based on the row key. If all your row keys start with “user-” then most likely the majority of your data will be isolated to a single node (which defeats the purpose of distributing the data in the first place). Your row keys, therefore, should be different enough to be distributed across the entire deployment/

The manner in which you design your row keys depends on how you intend to access those rows. If you store data on a per user basis, then one strategy is to leverage the fact that row keys are ultimately stored as byte arrays in HBase, so we can create a hash (such as an MD5 or SHA-1 hash code) of the user ID and then append the time (as a long) to hash. The importance in using a hash is two-fold: (1) it distributes values so that the data can be distributed across the cluster and (2) it ensures that the length (in bytes) of the key is consistent and hence easier to use in table scans.

# What happens internally when new data gets inserted into HBase table?

HBase is built upon distributed filesystems with file storage distributed across commodity machines. The distributed file systems HBase works with include

* Hadoop’s Distributed File System (HDFS) and
* Amazon’s Simple Storage Service (SS3).



HDFS provides a scalable and replicated storage layer for HBase. It guarantees that data is never lost by writing the changes across a configurable number of physical servers.

The data is stored in HFiles, which are ordered immutable key/value maps. Internally, the HFiles are sequences of blocks with a block index stored at the end. The block index is loaded when the HFile is opened and kept in memory.  The default block size is 64 KB but it can be changed since it is configurable. HBase API can be used to access specific values and also scan ranges of values given a start and end key.

Since every HFile has a block index, lookups can be performed with a single disk seek. First, HBase does a binary search in the in-memory block index to find a block containing the given key and then the block is read from disk.

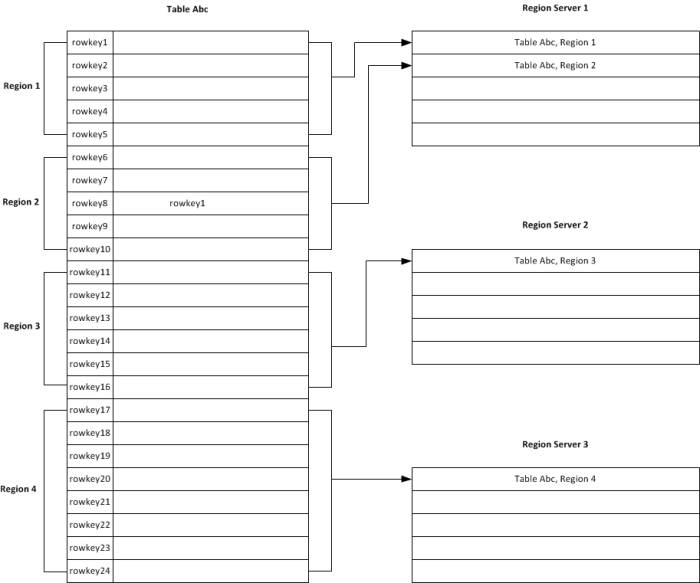
When data is updated it is first written to a commit log, called a write-ahead log (WAL) and then it is stored in the in-memory memstore.

When the data in memory exceeds a given maximum value, it is flushed as an HFile to disk and after that the commit logs are discarded up to the last unflushed modification. The system can continue to serve readers and writers without blocking them while it is flushing the memstore to disk. This is done by rolling the memstore in memory where the new empty one is taking the updates and the old full one is transferred into an HFile. At the same time, no sorting or other special processing has to be performed since the data in the memstores is already sorted by keys matching what HFiles represent on disk.

The write-ahead log (WAL) is used for recovery purposes only. Since flushing memstores to disk causes creation of HFiles, HBase has a housekeeping job that merges the HFiles into larger ones using compaction. Various compaction algorithms are supported.

Other HBase architectural components include the client library (API), at least one master server, and many region servers. The region servers can be added or removed while the system is up and running to accommodate increased workloads. The master is responsible for assigning regions to region servers. It uses [Apache ZooKeeper](https://zookeeper.apache.org/), a distributed coordination service, to facilitate that task.

Data is partitioned and replicated across a number of regions located on region servers.



As mentioned above, assignment and distribution of regions to region servers is automatic. However manual management of regions is also possible. When a region’s size reaches a pre-defined threshold, the region will automatically split into two child regions. The split happens along a row key boundary. A single region always manage an entire row. It means that a rows are never divided.

## How is it sorted?

Key/value pairs in HBASE maps are kept in an alphabetical order. The amount of data you can store in HBase can be huge and the data you are retrieving via your queries should be near each other.

For example, if you run a query on an HBase table that returns thousands of rows which are distributed across many machines, the latency affected by your network can be significant. This data distribution is determined by a row key of the HBase table. Because of that the row key design is one of the most important aspects of the HBase data modeling (schema design). If a row key is not properly designed it can create hot spotting where a large amount of client traffic is directed at one or few nodes of a cluster.

The row key should be defined in a way that allows related rows to be stored near each other. These related rows will be retrieved by queries and as long as they are stored near each other you should experience good performance. Otherwise the performance of your system will be impacted.

# Table

Data is stored in tables that have rows and columns. Table names are strings. They are composed of characters that are safe to use in file system paths. Tables are logically grouped into **namespaces** by applications, or users, or access control, etc. A namespace is analogous to a database in relational database management systems. A namespace membership is determined during the table creation when tables are fully named.

As previously mentioned, HBASE tables are multi-dimensional maps. A table has multiple rows. A row consists of a row key that is sortable and one or more columns with values associated with them. Rows are uniquely identified by their row key. A row key has no data type and is always treated as an array of bytes. A row key is an equivalent of a primary key in a relational database table. The row key is the only way to access the row. The number of columns per row is arbitrary. It can vary from row to row. Columns are organized in column families. At a conceptual level, tables may be viewed as a sparse set of rows that are physically stored by a column family.

# Column Family

Column families are specified when a table is created. They should be carefully designed before a table is created since it would be either impossible or difficult to change them later.

Column families’ names are strings that are composed of characters that are safe to use in file system paths.

All columns in a column family are stored and sorted together in the same HFile.

Column families group columns together physically and logically and they are usually used for a performance reason. A column family has a set of parameters that specify its storage (e.g., caching, compression, etc.). All tuning and storage specifications are done at the column family level. It is important that all column family members have the same or similar access pattern and sizes.

Some shortcomings in the current HBase implementation do not properly support large number of column families in a single table. That number should be in low tens. Most of the time up to three column families should work fine without any significant performance drawback. Ideally you should go with a single column family. The column family names should be as small as possible, preferably one character.

A column family can have an arbitrary number of columns denoted by a column qualifier which is like a column’s label. For example:

{

"row1": {"1": {"color": "green",

"size": 25},

"2": {"weight": 52,

"size": 18}

},

"row2": {"1": {"color": "blue"},

"2": {"height": 192,

"size": 43}

}

}

As you can see in the example above, the same column family (e.g., “1”) in two rows can have different columns. In row “row1”, it has columns “color” and “size”, while in row “row2”, it has only “color” column. It can also have a column that is none of the above. Since rows can have different columns in column families there is no a single way to query for a list of all columns in all column families. This means that you have to do a full table scan.

**There is no specific limit on the number of columns in a column family. Actually you can have millions of columns in the single column family.**

# Column

Columns are usually physically co-located in column families. A column is identified by column family and column qualifier separated by a colon character (:). For example, courses:math. The column family prefix must be composed of printable characters. The column qualifiers (columns) do not have to be defined at schema definition time and they can be added on the fly while the database is up and running.

A column qualifier is an index for a given data and it is added to a column family. Data within a column family is addressed via the column qualifier. Column qualifiers are mutable and they may vary between rows. They do not have data types and they are always treated as arrays of bytes.

A row key, column family and column qualifier form a cell that has a value and timestamp that represents the value’s version. Values also do not have data types and they are always treated as arrays of bytes. A timestamp is recorded for each value and it is the time on the region server when the value was written.

All cell’s values are stored in a descending order by its timestamp. When values are retrieved and if the timestamp is not provided then HBase will return the cell value with the latest (the most recent) timestamp. If a timestamp is not specified during the write, the current timestamp is used.

The maximum number of versions (timestamps) for a given column to store is part of the column schema. It is specified at table creation. It can be specified via alter table command as well. The default value is 1. The minimum number of versions can be also set up per column family. You can also globally set up a maximum number of versions per column.

HBase does not overwrite row values. It stores different values per row by time and column qualifier. Extra versions above the current max version setup are removed during major compactions. If it is not necessary it is not recommended to have very high maximum number of versions since it will increase the HFile size significantly.

It is worth to mention that the column metadata is only stored in internal key/value instances for a column family.