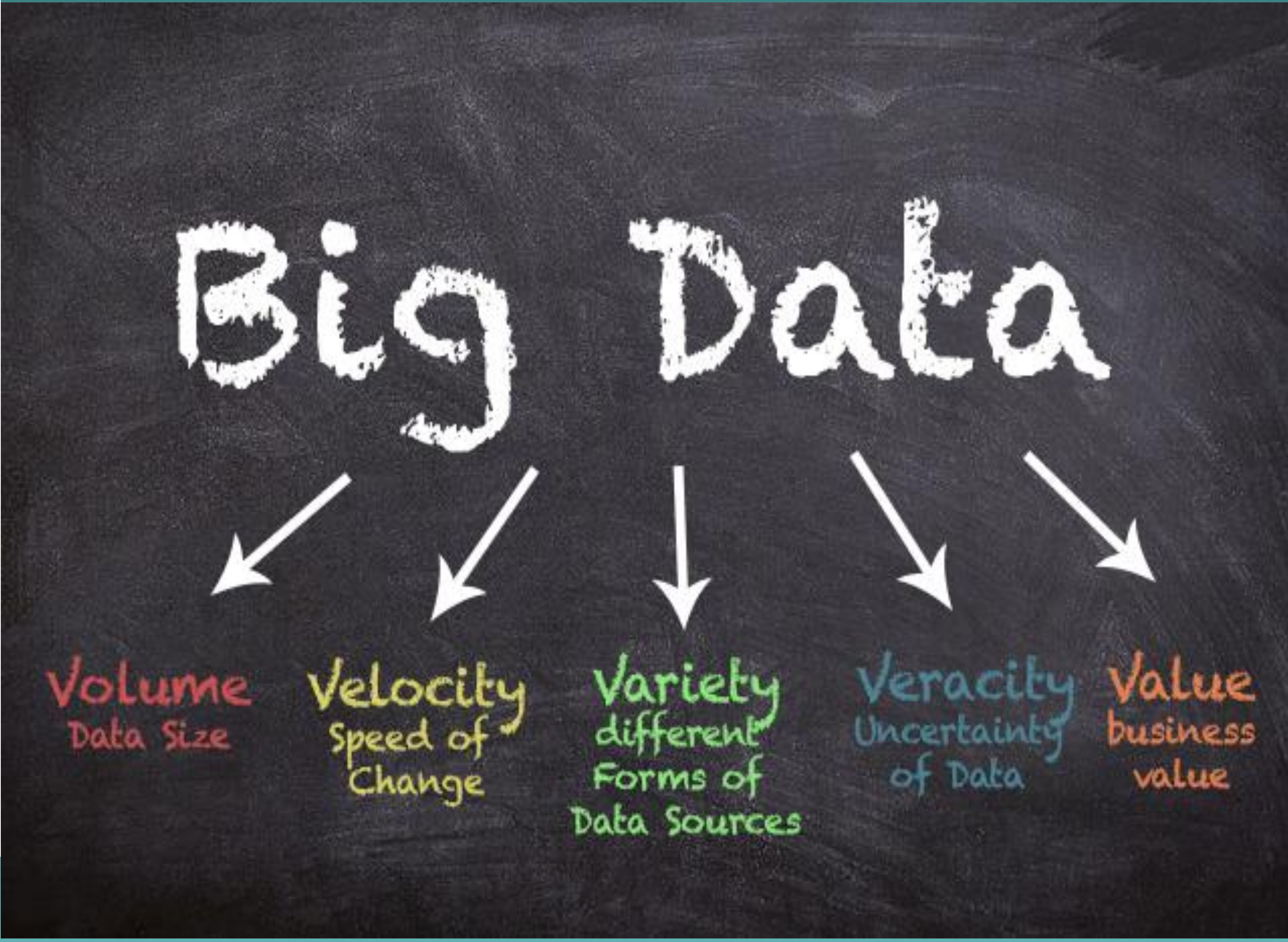
08/09/2018 | By: Ajit K Prasad



Big Data Engineering with Hadoop & Spark

Assignment on HBase Basics

Acadgild

Session 10: Assignment 10.1

This assignment is aimed at consolidating the concepts that was learnt during the HBase Basics session of the course.

Problem Statement

Task 1:

1. What is NoSQL data base?

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.”

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.

It is an approach to database design that can accommodate a wide variety of data models, including key-value, document, columnar and graph formats.

<u>Benefits of NoSQL:</u>

NoSQL databases offer enterprises important advantages over traditional RDBMS, including:

Scalability: NoSQL databases use a horizontal scale-out methodology that makes it easy to add or reduce capacity quickly and non-disruptively with commodity hardware. This eliminates the tremendous cost and complexity of manual sharding that is necessary when attempting to scale RDBMS.

Performance: By simply adding commodity resources, enterprises can increase performance with NoSQL databases. This enables organizations to continue to deliver reliably fast user experiences with a predictable return on investment for adding resources—again, without the overhead associated with manual sharding.

High Availability: NoSQL databases are generally designed to ensure high availability and avoid the complexity that comes with a typical RDBMS architecture that relies on primary and secondary nodes. Some “distributed” NoSQL databases use a masterless architecture that automatically distributes data equally among multiple resources so that the application remains available for both read and write operations even when one node fails.

Global Availability: By automatically replicating data across multiple servers, data centers, or cloud resources, distributed NoSQL databases can minimize latency and ensure a consistent application experience wherever users are located. An added benefit is a significantly reduced database management burden from manual RDBMS configuration, freeing operations teams to focus on other business priorities.

Flexible Data Modeling: NoSQL offers the ability to implement flexible and fluid data models. Application developers can leverage the data types and query options that are the most natural fit to the specific application use case rather than those that fit the database schema. The result is a simpler interaction between the application and the database and faster, more agile development.

Examples of NoSQL: Cassandra, MongoDB, HBase, Redis etc.

2. How does data get stored in NoSQL database?

Several different varieties of NoSQL databases have been created to support specific needs and use cases. These fall into four main categories:

Key-value data stores: Key-value NoSQL databases emphasize simplicity and are very useful in accelerating an application to support high-speed read and write processing of non-transactional data. Stored values can be any type of binary object (text, video, JSON document, etc.) and are accessed via a key. The application has complete control over what is stored in the value, making this the most flexible NoSQL model. Data is partitioned and replicated across a cluster to get scalability and availability. For this reason, key value stores often do not support transactions. However, they are highly effective at scaling applications that deal with high-velocity, non-transactional data.

e.g. Riak, MemchacheDB, Aerospike, Berkeley DB and Redis.

Document stores: Document databases typically store self-describing JSON, XML, and BSON documents. They are similar to key-value stores, but in this case, a value is a single document that stores all data related to a specific key. Popular fields in the document can be indexed to provide fast retrieval without knowing the key. Each document can have the same or a different structure.

e.g. Couchbase Server, MarkLogic, CouchDB, Document DB and MongoDB.

Wide-column stores: Wide-column NoSQL databases store data in tables with rows and columns similar to RDBMS, but names and formats of columns can vary from row to row across the table. Wide-column databases group columns of related data together. A query can retrieve related data in a single operation because only the columns associated with the query are retrieved. In an

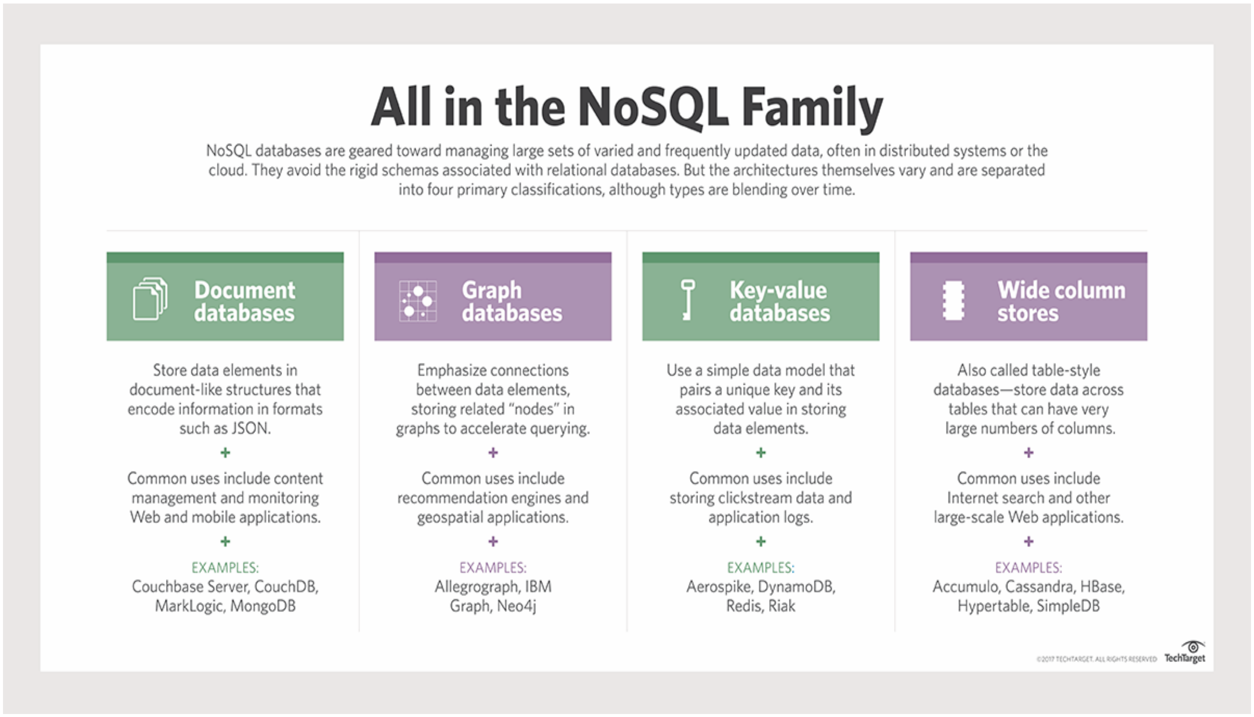
RDBMS, the data would be in different rows stored in different places on disk, requiring multiple disk operations for retrieval.

e.g. HBase, Google Bigtable and Cassandra.

Graph stores: A graph database uses graph structures to store, map, and query relationships. They provide index-free adjacency, so that adjacent elements are linked together without using an index.

e.g. AllegroGraph, Neo4j, IBM Graph and Titan.

Multi-modal databases leverage some combination of the four types described above and therefore can support a wider range of applications.



All in the NoSQL Family

NoSQL databases are geared toward managing large sets of varied and frequently updated data, often in distributed systems or the cloud. They avoid the rigid schemas associated with relational databases. But the architectures themselves vary and are separated into four primary classifications, although types are blending over time.

Document databases

Store data elements in document-like structures that encode information in formats such as JSON.

+

Common uses include content management and monitoring Web and mobile applications.

+

EXAMPLES: Couchbase Server, CouchDB, MarkLogic, MongoDB

Graph databases

Emphasize connections between data elements, storing related "nodes" in graphs to accelerate querying.

+

Common uses include recommendation engines and geospatial applications.

+

EXAMPLES: Allegrograph, IBM Graph, Neo4j

Key-value databases

Use a simple data model that pairs a unique key and its associated value in storing data elements.

+

Common uses include storing clickstream data and application logs.

+

EXAMPLES: Aerospike, DynamoDB, Redis, Riak

Wide column stores

Also called table-style databases—store data across tables that can have very large numbers of columns.

+

Common uses include Internet search and other large-scale Web applications.

+

EXAMPLES: Accumulo, Cassandra, HBase, Hypertable, SimpleDB

©2017 TECHTARGET. ALL RIGHTS RESERVED TechTarget

3. What is a column family in HBase?

Columns in Apache HBase are grouped into column families. All column members of a column family have the same prefix.

For example, the columns **cf1: street** and **cf1: name** is both members of the **cf1** column family. The colon character **(:)** delimits the column family from 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 and 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.

e.g.:

R1 column=**cf1:dept_city**, timestamp=15334595875, value=**NEW YORK**

R1 column=**cf1:dept_name**, timestamp=15334425907, value=**ACCOUNTING**

R1 = Row

cf1 = Column Family

dept_city, dept_name = Columns

NEW YORK, ACCOUNTING = Values

Note: Column family cannot be updated. If a table consists of only one column family then that column family cannot be deleted. Column families are grouped together on disk, so grouping data with similar access patterns reduces overall disk access and increases performance.

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

There is no hard limit to number of columns in HBase, we can have more than 1 million columns but usually **three column families** are recommended (not more than three).

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

HBase has a Dynamic Shema. It uses query-first schema design; all possible queries should be identified first, and the schema model designed accordingly. By not defining columns at the time of creation it provides flexibility to add columns on need basis. Traditional RDBMS database, all the schemas are defined at the beginning. This is a constraint when project requirement changes, we need to rework on whole model. By keeping flexibility, any new requirements can be easily done without revisiting models.

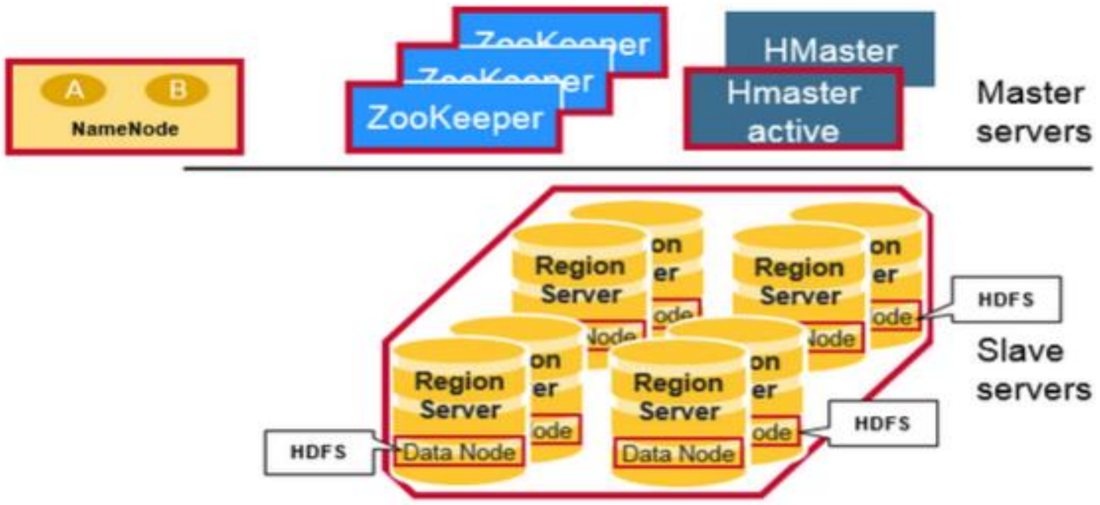
6. How does data get managed in HBase?

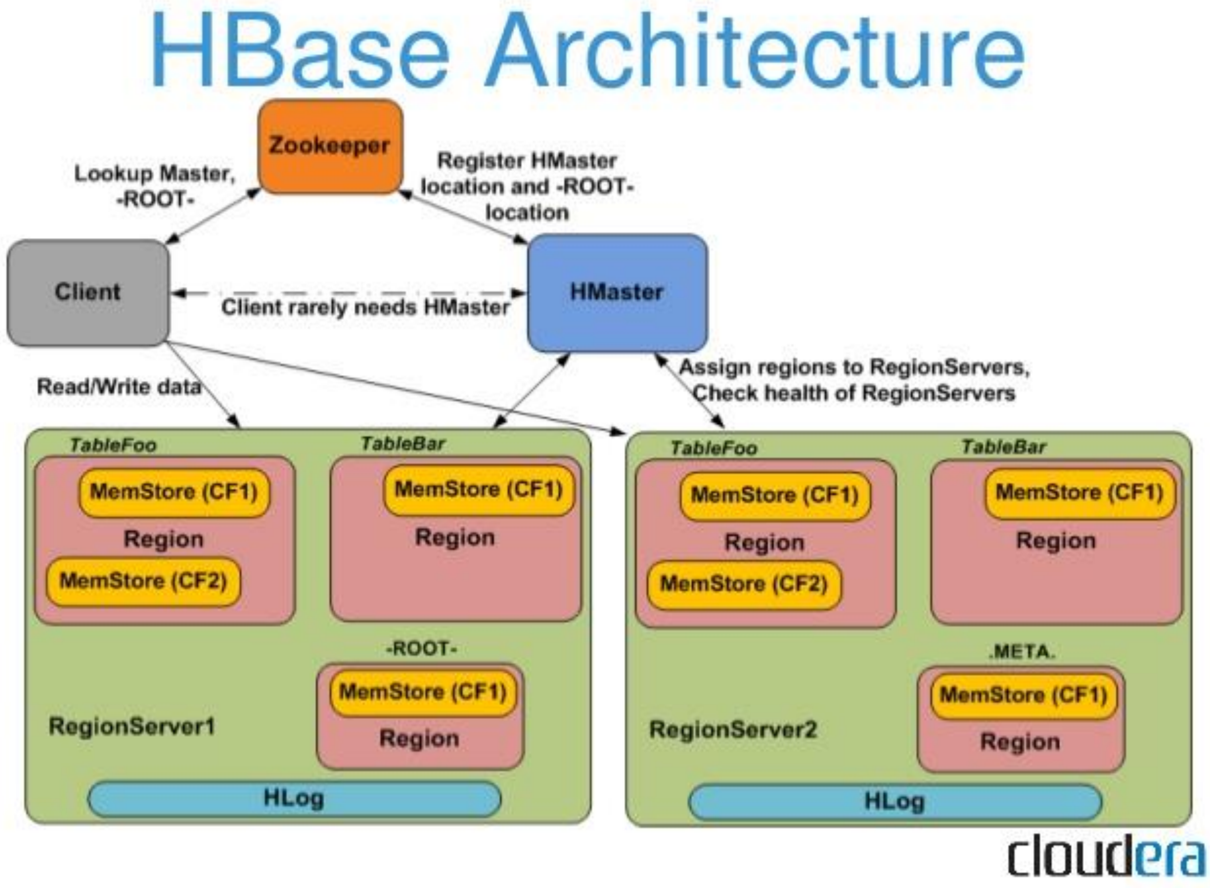
Data in HBase is organized into tables. Any characters that are legal in file paths are used to name tables. Tables are further organized into rows that store data. Each row is identified by a unique row key which does not belong to any data type but is stored as a bytearray.

HBase stores data in a table-like format with the ability to store billions of rows with millions of columns. Columns can be grouped together in “column families” which allows physical distribution of row values onto different cluster nodes.

HBase group rows into “regions” which define how table data is split over multiple nodes in a cluster. If a region gets too large, it is automatically split to share the load across more servers.

Two different graphical representations of the HBase Architecture:





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

When ***Put*** is used to store data, it uses the row, the column, and the timestamp. The timestamp is unique per version of the cell and can be generated automatically or specified programmatically by the application and must be a long integer.

When a ***put/insert*** request is initiated, the value is first written into the WAL and then into the memstore. The values in the Memstore is stored in the same sorted manner as in the HFile. Once the Memstore is full, it is then flushed into a new HFile.

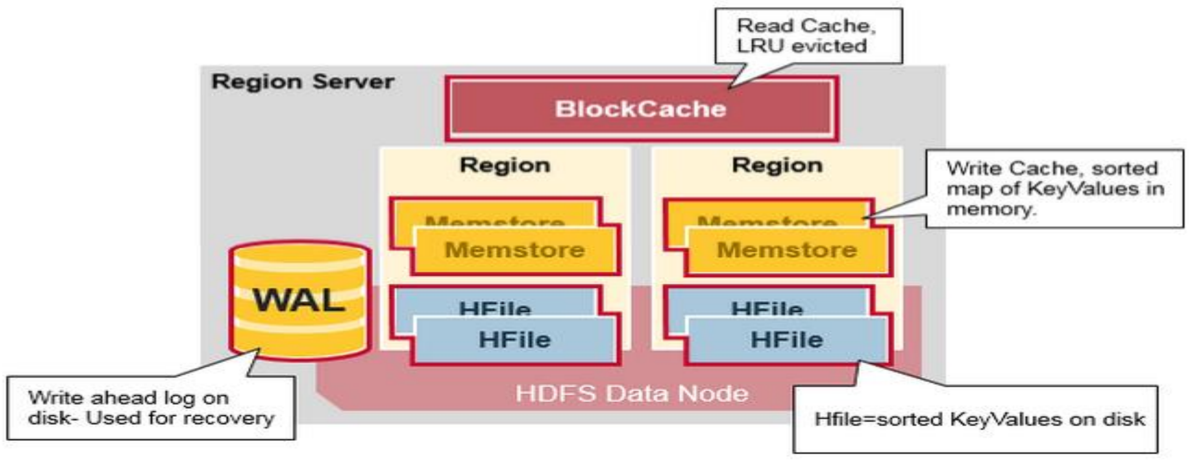
HFile stores the data in sorted order i.e. the sequential rowkeys will be next to each other.

HBase periodically performs compactions which will merge multiple HFiles and rewrite's them to a single HFile, this new HFile which is a result of compaction is also sorted.

The READ-WRITE data in the HBASE is managed in the following ways:

- **WAL:** Write Ahead Log is a file on the distributed file system. The WAL is used to store new data that hasn't yet been persisted to permanent storage; it is used for recovery in the case of failure.
- **BlockCache:** It is the read cache. It stores frequently read data in memory. Least Recently Used data is evicted when full.
- **MemStore:** It is the write cache. It stores new data which has not yet been written to disk. It is sorted before writing to disk. There is one MemStore per column family per region.
- **Hfiles:** It stores the rows as sorted KeyValues on disk.

All the above are sub components of ***Region Server*** which manages the data in the HBase.



Task 2:

1. Create an HBase table named 'clicks' with a column family 'hits' such that it should be able to store last 5 values of qualifiers inside 'hits' column family.

Solution:

– This can be performed using the command below, this would create a table called 'clicks' with a column family 'hits', also be able to store last 5 values of the qualifiers inside 'hits' column family.

```
hbase(main):004:0> create 'clicks','hits',{NAME=>'hits', VERSIONS => 5}
```

2. Add few records in the table and update some of them. Use IP Address as row-key. Scan the table to view if all the previous versions are getting displayed.

Solution

– The above task can be performed using the following commands, 'put' command inserts the data, as shown below:

```
hbase(main):005:0>     put 'clicks','172.16.254.1','hits:browser','Firefox'
hbase(main):006:0>     put 'clicks','172.16.254.1','hits:browser','Opera'
hbase(main):007:0>     put 'clicks','172.16.254.1','hits:browser','Edge'
hbase(main):008:0>     put 'clicks','172.16.254.1','hits:browser','Chrome'
hbase(main):009:0>     put 'clicks','172.16.254.1','hits:browser','Safari'
hbase(main):010:0>     put 'clicks','172.16.254.1','hits:browser','Maxthon'
```

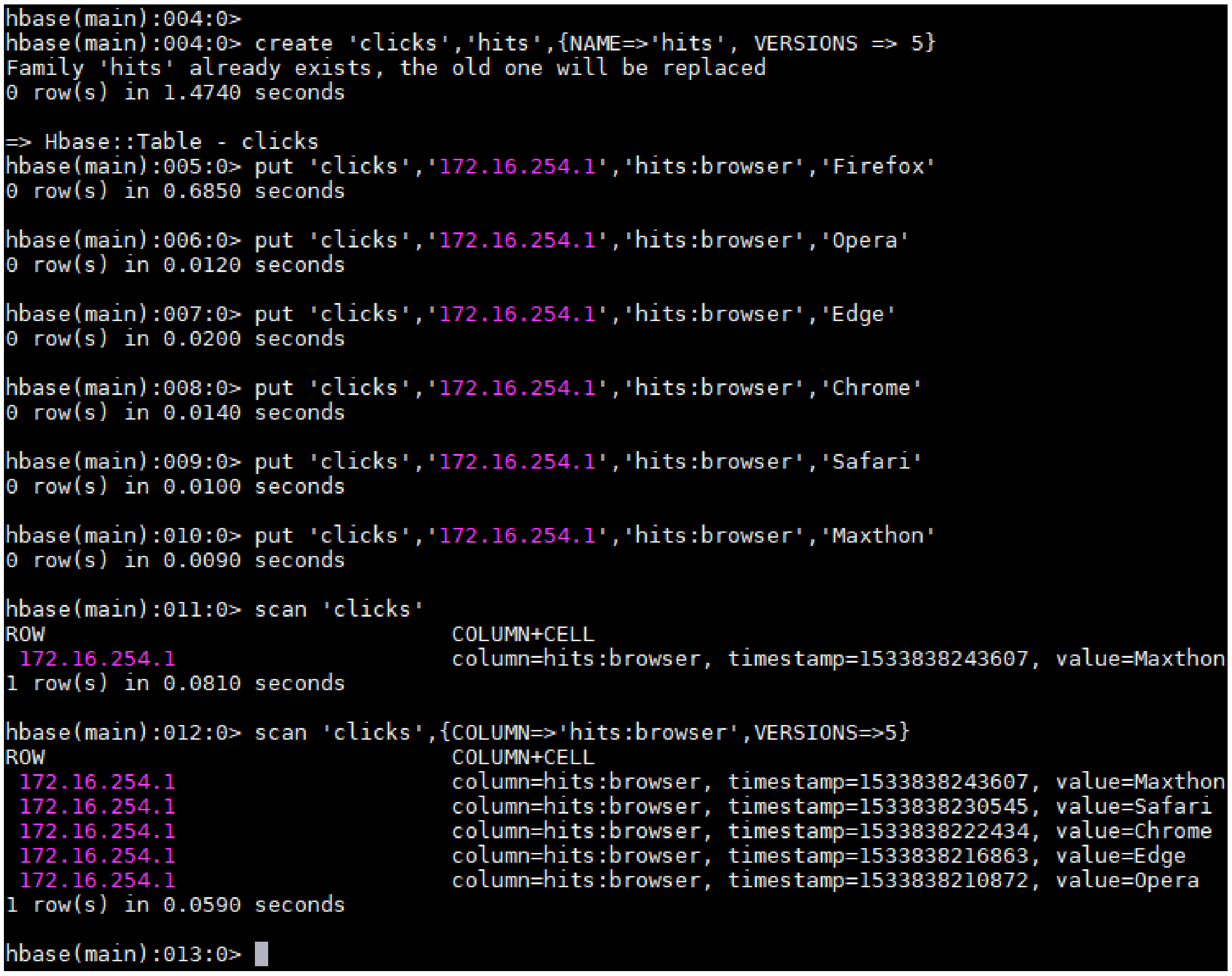
– To check the data inserted, use the command below:

```
hbase(main):011:0>     scan 'clicks'
```

– Although table clicks retains the latest updated value for browser, i.e. 'Maxthon'

```
hbase(main):012:0>     scan 'clicks', {COLUMN=>'hits:browser',VERSIONS=>5}
```

– "Versions => 5" shows the last five values that were updated/changed for column 'browser' in column family 'hits'



```
hbase(main):004:0>
hbase(main):004:0> create 'clicks','hits',{NAME=>'hits', VERSIONS => 5}
Family 'hits' already exists, the old one will be replaced
0 row(s) in 1.4740 seconds

=> Hbase::Table - clicks
hbase(main):005:0> put 'clicks','172.16.254.1','hits:browser','Firefox'
0 row(s) in 0.6850 seconds

hbase(main):006:0> put 'clicks','172.16.254.1','hits:browser','Opera'
0 row(s) in 0.0120 seconds

hbase(main):007:0> put 'clicks','172.16.254.1','hits:browser','Edge'
0 row(s) in 0.0200 seconds

hbase(main):008:0> put 'clicks','172.16.254.1','hits:browser','Chrome'
0 row(s) in 0.0140 seconds

hbase(main):009:0> put 'clicks','172.16.254.1','hits:browser','Safari'
0 row(s) in 0.0100 seconds

hbase(main):010:0> put 'clicks','172.16.254.1','hits:browser','Maxthon'
0 row(s) in 0.0090 seconds

hbase(main):011:0> scan 'clicks'
ROW                                  COLUMN+CELL
 172.16.254.1                        column=hits:browser, timestamp=1533838243607, value=Maxthon
1 row(s) in 0.0810 seconds

hbase(main):012:0> scan 'clicks',{COLUMN=>'hits:browser',VERSIONS=>5}
ROW                                  COLUMN+CELL
 172.16.254.1                        column=hits:browser, timestamp=1533838243607, value=Maxthon
 172.16.254.1                        column=hits:browser, timestamp=1533838230545, value=Safari
 172.16.254.1                        column=hits:browser, timestamp=1533838222434, value=Chrome
 172.16.254.1                        column=hits:browser, timestamp=1533838216863, value=Edge
 172.16.254.1                        column=hits:browser, timestamp=1533838210872, value=Opera
1 row(s) in 0.0590 seconds

hbase(main):013:0>
```