Assignment 10 : Hbase Basics Assignment Problems

Problem Statement

Task 1

Answer in your own words with example.

1.What is NoSQL data base?

#Ans 1:

A NoSQL (often interpreted as Not only SQL) database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. Motivations for this approach include simplicity of design, horizontal scaling, and finer control over availability.

CAP Theorem – Consistency, Availability, Partition Tolerance

Consistency: all nodes see the same data at the same time

Availability: a guarantee that every request receives a response about whether it was successful or failed

Partition tolerance: the system continues to operate despite arbitrary message loss

The theorem states that you cannot simultaneously have all three; you must make tradeoffs among them.

NoSQL (Not only SQL) came as a solution to this problem of relational database management systems and allowed the companies to store massive amounts of structured, semi-structured and unstructured data in real-time. It does not certainly imply that it restricts the usage of SQL for these databases. Some of the popular NoSQL databases are HBase, Cassandra, IBm Informix, MongoDB, Amazon SimpleDB, Cloudata, etc.

- NoSQL is agile because it does not create schemas nor it statically defines the data models
- Instead of tables it uses objects, collections and nested collections
- Deployed over multiple cheap Intel-based servers
- Immediate failover with the help of uni-directional and bi-directional replication of data
- Equipped with the big data, cloud, mobile and web technologies
- Trades conventional ACID properties to incorporate more flexibility and agility.

Initially when the applications used relational databases, the developers found difficulties in matching the data structures supported by the two platforms. They had to convert the in-memory data structures into relational ones in order to transfer the data to and from the database. This reduced the agility and performance of the systems in a big way.

NO SQL can be deployed in four different manners :

Columnar Databases – Reads and writes columns of data rather than the rows. Each column is comparable to a container in RDBMS where a Key defines a row and single row has multiple columns.

Document Databases – These databases store and retrieve semi-structured data in the format of documents such as XML, JSON, etc. Some of the popular document databases like MongoDB provide a rich query language for ease of access and smooth transition of data models.

Graph Databases – Stores data as entities and relations between them allowing faster traversal and joining operations to be performed. However these graphs can be built using SQL as well as NoSQL databases.

In-Memory Key-Value Stores- Suitable for read-heavy workloads and compute-intensive workloads, these databases store critical data in memory which in turn improves the performance of the systems.

2.How does data get stored in NoSQl database?

#Ans 2:

The Structured Query Language used by traditional databases provides a uniform way to communicate with the server when storing and retrieving data. SQL syntax is highly standardized, so while individual databases may handle certain operations differently (for e.g Window functions), the basics remain the same.

By contrast, each NoSQL database tends to have its own syntax for querying and managing the data. CouchDB, for instance, uses requests in the form of JSON, sent via HTTP, to create or retrieve documents from its database. MongoDB sends JSON objects over a binary protocol, by way of a command-line interface or a language library.

Some NoSQL products *can* use SQL-like syntax to work with data, but only to a limited extent. For example, Apache Cassandra, a column store database, has its own SQL-like language, the Cassandra Query Language (CQL). Some of the CQL syntax is straight out of the SQL playbook, like the SELECT or INSERT keywords. But there is no way to perform a JOIN or subquery in Cassandra, and thus the related keywords don’t exist in CQL.

A design choice common to NoSQL systems is a “shared-nothing” architecture. In a shared-nothing design, each server node in the cluster operates independently of every other node. The system doesn’t have to get consensus from every single node to return a piece of data to a client. Queries are fast because they can be returned from whichever node is closest or most convenient.

Another advantage of shared-nothing is resiliency and scale-out. Scaling out the cluster is as easy as spinning up new nodes in the cluster and waiting for them to sync with the others. If a NoSQL node goes down, the other servers in the cluster will continue to chug along. All the data remains available, even if fewer nodes are available to serve requests.

Note that a shared-nothing design is not *exclusive* to NoSQL databases. Many conventional SQL systems can be set up in a shared-nothing fashion, although that typically involves sacrificing consistency across the cluster for performance.

No schema

Even if you’re taking in free-form data, you almost always need to impose constraints on it to make it useful. With NoSQL, imposing constraints involves shifting the responsibility from the database to the application developer. For instance, the developer could impose structure through an object relational mapping system, or ORM. But if you want the schema to live *with the data itself*, NoSQL does not typically do that.

Some NoSQL solutions provide optional data typing and validation mechanisms for data. Apache Cassandra, for instance, has a slew of native data types that are reminiscent of those found in conventional SQL.

Eventual consistency

NoSQL systems trade strong or immediate consistency for better availability and performance. Conventional databases ensure that operations are *atomic* (all parts of a transaction succeed, or none do), *consistent* (all users have the same view of the data), *isolated* (transactions don’t compete), and *durable* (once completed they will survive a server failure).

These four properties, collectively referred to as ACID, are handled differently in most NoSQL systems. Instead of immediate consistency across the cluster, you have *eventual* consistency, due to the time needed to copy updates to other nodes in the cluster. Data inserted into the cluster is eventually available everywhere, but you can’t guarantee when.

Transaction semantics, which in a SQL system guarantee that all steps in a transaction (e.g. executing a sale *and* reducing inventory) are either completed or rolled back, aren’t typically available in NoSQL. For any system where there needs to be a “single source of truth,” such as a bank, the NoSQL approach

won’t work well. You don’t want your bank balance to be different depending on which ATM you go to; you want it to be reported as the same thing everywhere.

Some NoSQL databases have partial mechanisms for working around this. For instance, MongoDB has consistency guarantees for individual operations, but not for the database as a whole. Microsoft Azure CosmosDB lets you select a level of consistency per request, so you can choose the behavior that fits your use case. But with NoSQL, expect eventual consistency as the default behavior.

NoSQL lock-in

Most NoSQL systems are *conceptually* similar, but are *implemented* very differently. Each tends to have its own metaphors and mechanisms for how data is queried and managed.

One side effect of that is a potentially high degree of coupling between the application logic and the database. This isn’t so bad if you pick a NoSQL system and stick with it, but it can become a stumbling block if you change systems down the road.

If you migrate from, say, MongoDB to CouchDB (or vice versa), you must do more than just migrate data. You also must navigate the differences in data access and programmatic metaphors—in other words, you must rewrite the parts of your application that access the database.

Aggregate Data Models:

Relational database modelling is vastly different than the types of data structures that application developers use. Using the data structures as modelled by the developers to solve different problem domains has given rise to movement away from relational modelling and towards aggregate models, most of this is driven by *Domain Driven Design*, a book by Eric Evans. An aggregate is a collection of data that we interact with as a unit. These units of data or aggregates form the boundaries for ACID operations with the database, Key-value, Document, and Column-family databases can all be seen as forms of aggregate-oriented database.

Aggregates make it easier for the database to manage data storage over clusters, since the unit of data now could reside on any machine and when retrieved from the database gets all the related data along with it. Aggregate-oriented databases work best when most data interaction is done with the same aggregate, for example when there is need to get an order and all its details, it better to store order as an aggregate object but dealing with these aggregates to get item details on all the orders is not elegant.

Aggregate-oriented databases make inter-aggregate relationships more difficult to handle than intra-aggregate relationships. Aggregate-ignorant databases are better when interactions use data organized in many different formations. Aggregate-oriented databases often compute materialized views to provide data organized differently from their primary aggregates. This is often done with map-reduce computations, such as a map-reduce job to get items sold per day.

Distribution Models:

Aggregate oriented databases make distribution of data easier, since the distribution mechanism has to move the aggregate and not have to worry about related data, as all the related data is contained in the aggregate. There are two styles of distributing data:

- Sharding: Sharding distributes different data across multiple servers, so each server acts as the single source for a subset of data.
- Replication: Replication copies data across multiple servers, so each bit of data can be found in multiple places. Replication comes in two forms,
    - Master-slave replication makes one node the authoritative copy that handles writes while slaves synchronize with the master and may handle reads.
    - Peer-to-peer replication allows writes to any node; the nodes coordinate to synchronize their copies of the data.

Master-slave replication reduces the chance of update conflicts but peer-to-peer replication avoids loading all writes onto a single server creating a single point of failure. A system may use either or both techniques. Like Riak database shards the data and also replicates it based on the replication factor.

CAP theorem:

In a distributed system, managing consistency(C), availability(A) and partition toleration(P) is important, Eric Brewer put forth the CAP theorem which states that in any distributed system we can choose only two of consistency, availability or partition tolerance. Many NoSQL databases try to provide options where the developer has choices where they can tune the database as per their needs. For example if you consider Riak a distributed key-value database. There are essentially three variables r, w, n where

- r=number of nodes that should respond to a read request before its considered successful.
- w=number of nodes that should respond to a write request before its considered successful.
- n=number of nodes where the data is replicated aka replication factor.

In a Riak cluster with 5 nodes, we can tweak the r,w,n values to make the system very consistent by setting r=5 and w=5 but now we have made the cluster susceptible to network partitions since any write will not be considered successful when any node is not responding. We can make the same cluster highly available for writes or reads by setting r=1 and w=1 but now consistency can be compromised since some nodes may not have the latest copy of the data. The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency of data. Durability can also be traded off against latency, particularly if you want to survive failures with replicated data.

NoSQL databases provide developers lot of options to choose from and fine tune the system to their specific requirements. Understanding the requirements of how the data is going to be consumed by the

system, questions such as is it read heavy vs write heavy, is there a need to query data with random query parameters, will the system be able handle inconsistent data.

Understanding these requirements becomes much more important, for long we have been used to the default of RDBMS which comes with a standard set of features no matter which product is chosen and there is no possibility of choosing some features over other. The availability of choice in NoSQL databases, is both good and bad at the same time. Good because now we have choice to design the system according to the requirements. Bad because now you have a choice and we have to make a good choice based on requirements and there is a chance where the same database product may be used properly or not used properly.

An example of feature provided by default in RDBMS is transactions, our development methods are so used to this feature that we have stopped thinking about what would happen when the database does not provide transactions. Most NoSQL databases do not provide transaction support by default, which means the developers have to think how to implement transactions, does every write have to have the safety of transactions or can the write be segregated into “critical that they succeed” and “its okay if I lose this write” categories. Sometimes deploying external transaction managers like ZooKeeper can also be a possibility.

3.What is a column family in Hbase?

#Ans 3:

An HBase table is made of column families which are the logical and physical grouping of columns. The columns in one family are stored separately from the columns in another family. If you have data that is not often queried, assign that data to a separate column family.

The column family and column qualifier names are repeated for each row. Therefore, keep the names as short as possible to reduce the amount of data that HBase stores and reads. For example, use c*f:c1* instead of *mycolumnfamily:mycolumnqualifier*.

Because column families are stored in separate HFiles, keep the number of column families as small as possible. You also want to reduce the number of column families to reduce the frequency of MemStore flushes, and the frequency of compactions. And, by using the smallest number of column families possible, you can improve the LOAD time and reduce disk consumption.

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

#Ans 4:

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 to avoid degradation in performance.

Depending on your data access patterns, we should consider wide table vs tall table layout trade-off.

Join like operations between different tables should be avoided as HBase will be not giving good performance for such type of queries.

There is a limit to the number of column families also in HBase. There is one MemStore which is a write cache that stores new data before writing it into Hfiles, per Column Family, when one is full, they all flush.

Thus, the more column families are added, the more MemStore is created and Memstore flush will be more frequent. It will degrade the performance.

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

#Ans 5:

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.

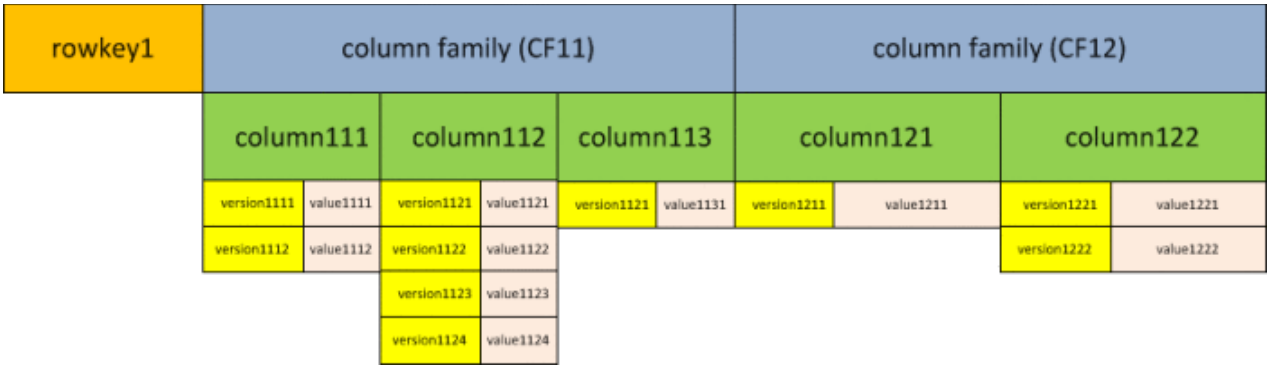
Thus columns are not defined at the time of table creation in Hbase because the column metadata is only stored in internal key/value instances for a column family. You have to keep track of the column names since HBase can support very high number of columns per row and columns can differ between the 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 names.

6.How does data get managed in Hbase?

#Ans 6:

HBase stores data in a form of a distributed sorted multidimensional persistence maps called Tables. The table terminology makes it easier for people coming from the relational data management world to abstract data organization in HBase. HBase is designed to manage tables with billions of rows and millions of columns.

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



Summary of 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.

HBASE returns the latest version of data by default but you can ask for multiple versions in your query. HBase returns data sorted first by the row key values, then by column family, column qualifier and finally by the timestamp value, with the most recent data returned first.

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

#Ans 7:

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, a distributed coordination service, to facilitate that task.

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

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.

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.

HBASE schema can be created either by HBASE shell or by Java API. When changes are made on a table, it has to be disabled until the changes are complete.

Changes on tables and column families take place when the next major compaction is done and HFiles re-written.

Task 2

Initial Terminal Execution :

```
[acadgild@localhost ~]$ jps
3115 Jps
[acadgild@localhost ~]$ sudo service sshd start
[sudo] password for acadgild:
Sorry, try again.
[sudo] password for acadgild:
[acadgild@localhost ~]$ sudo service mysqld start
Starting mysqld:                                           [  OK  ]
[acadgild@localhost ~]$ start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
18/07/17 21:33:32 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your
platform... using builtin-java classes where applicable
Starting namenodes on [localhost]
localhost: starting namenode, logging to
/home/acadgild/install/hadoop/hadoop-2.6.5/logs/hadoop-acadgild-namenode-localhost.localdomain.ou
t
localhost: starting datanode, logging to
/home/acadgild/install/hadoop/hadoop-2.6.5/logs/hadoop-acadgild-datanode-localhost.localdomain.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to
/home/acadgild/install/hadoop/hadoop-2.6.5/logs/hadoop-acadgild-secondarynamenode-localhost.local
domain.out
18/07/17 21:34:23 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your
platform... using builtin-java classes where applicable
starting yarn daemons
starting resourcemanager, logging to
/home/acadgild/install/hadoop/hadoop-2.6.5/logs/yarn-acadgild-resourcemanager-localhost.localdomai
n.out
localhost: starting nodemanager, logging to
/home/acadgild/install/hadoop/hadoop-2.6.5/logs/yarn-acadgild-nodemanager-localhost.localdomain.ou
t
[acadgild@localhost ~]$ jps
3410 DataNode
3747 ResourceManager
```

3315 NameNode
3542 SecondaryNameNode
3847 NodeManager
7996 Jps
[acadgild@localhost ~]$ start-hbase.sh
localhost: running zookeeper, logging to
/home/acadgild/install/hbase/hbase-1.4.4/logs/hbase-acadgild-zookeeper-localhost.localdomain.out
running master, logging to
/home/acadgild/install/hbase/hbase-1.4.4/logs/hbase-acadgild-master-localhost.localdomain.out
: running regionserver, logging to
/home/acadgild/install/hbase/hbase-1.4.4/logs/hbase-acadgild-regionserver-localhost.localdomain.out
[acadgild@localhost ~]$ jps
3410 DataNode
8514 Jps
3747 ResourceManager
3315 NameNode
3542 SecondaryNameNode
8391 HRegionServer
3847 NodeManager
8297 HMaster
8207 HQuorumPeer
[acadgild@localhost ~]$ hbase shell
2018-07-18 06:59:25,539 WARN [main] util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in
[jar:file:/home/acadgild/install/hbase/hbase-1.4.4/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in
[jar:file:/home/acadgild/install/hadoop/hadoop-2.6.5/share/hadoop/common/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
Version 1.4.4, rfe146eb48c24d56dbcd2f669bb5ff8197e6c918b, Sun Apr 22 20:42:02 PDT 2018
hbase(main):001:0>

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.

Terminal Execution :

hbase(main):003:0> create 'clicks',{NAME=>'hits',VERSIONS=>5}
Unknown argument ignored for column family hits: 1.8.7

```
0 row(s) in 3.0930 seconds

=> Hbase::Table - clicks
hbase(main):004:0> describe 'clicks'
Table clicks is ENABLED
clicks
COLUMN FAMILIES DESCRIPTION
{NAME => 'hits', BLOOMFILTER => 'ROW', VERSIONS => '5', IN_MEMORY => 'false', KE
EP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER',
CO
MPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65
536', REPLICATION_SCOPE => '0'}
1 row(s) in 0.3510 seconds
hbase(main):005:0> put 'clicks','172.10.17.182','hits:hitcount','10'
0 row(s) in 0.2440 seconds

hbase(main):006:0> put 'clicks','172.10.17.182','hits:hitcount','20'
0 row(s) in 0.0290 seconds

hbase(main):007:0> put 'clicks','172.10.17.182','hits:hitcount','30'
0 row(s) in 0.0240 seconds

hbase(main):008:0> put 'clicks','172.10.17.182','hits:hitcount','40'
0 row(s) in 0.0270 seconds

hbase(main):009:0> put 'clicks','172.10.17.182','hits:hitcount','50'
0 row(s) in 0.0290 seconds

hbase(main):010:0> put 'clicks','172.10.17.182','hits:hitcount','60'
0 row(s) in 0.0360 seconds

hbase(main):011:0> put 'clicks','172.10.17.182','hits:hitcount','70'
0 row(s) in 0.0270 seconds

hbase(main):018:0> scan 'clicks',{NAME =>'hits' , VERSIONS => 5 }
ROW                    COLUMN+CELL
 172.10.17.182           column=hits:hitcount, timestamp=1531963444698, value=60
 172.10.17.182           column=hits:hitcount, timestamp=1531963426695, value=50
 172.10.17.182           column=hits:hitcount, timestamp=1531963419665, value=30
 172.10.17.182           column=hits:hitcount, timestamp=1531963340892, value=20
 172.10.17.182           column=hits:hitcount, timestamp=1531962475897, value=180
1 row(s) in 0.0350 seconds
```

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.

Terminal Execution :

```
hbase(main):019:0> put 'clicks','172.10.17.185','hits:hitcount','100'
0 row(s) in 0.0470 seconds

hbase(main):020:0> put 'clicks','172.10.17.185','hits:hitcount','200'
0 row(s) in 0.0290 seconds

hbase(main):021:0> scan 'clicks',{NAME =>'hits' , VERSIONS => 5 }
ROW                    COLUMN+CELL
 172.10.17.182           column=hits:hitcount, timestamp=1531963444698, value=60
 172.10.17.182           column=hits:hitcount, timestamp=1531963426695, value=50
 172.10.17.182           column=hits:hitcount, timestamp=1531963419665, value=30
 172.10.17.182           column=hits:hitcount, timestamp=1531963340892, value=20
 172.10.17.182           column=hits:hitcount, timestamp=1531962475897, value=180
 172.10.17.185           column=hits:hitcount, timestamp=1531963692962, value=200
 172.10.17.185           column=hits:hitcount, timestamp=1531963685896, value=100
2 row(s) in 0.1200 seconds

hbase(main):022:0> is_enabled 'clicks'
true
0 row(s) in 0.0540 seconds

hbase(main):023:0> disable 'clicks'
0 row(s) in 2.3040 seconds

hbase(main):024:0> is_enabled 'clicks'
false
0 row(s) in 0.0150 seconds

hbase(main):025:0> alter 'clicks' , {NAME =>'users', VERSIONS=> 3}
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 2.2450 seconds

hbase(main):026:0> enable 'clicks'
0 row(s) in 1.2630 seconds

hbase(main):027:0> is_enabled 'clicks'
true
0 row(s) in 0.0370 seconds

hbase(main):028:0> describe 'clicks'
Table clicks is ENABLED
clicks
COLUMN FAMILIES DESCRIPTION
{NAME => 'hits', BLOOMFILTER => 'ROW', VERSIONS => '5', IN_MEMORY => 'false',
KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCOD
```

```
ING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0',
BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPL
ICATION_SCOPE => '0'}
{NAME => 'users', BLOOMFILTER => 'ROW', VERSIONS => '3', IN_MEMORY => 'false',
KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCO
DING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0',
BLOCKCACHE => 'true', BLOCKSIZE => '65536', REP
LICATION_SCOPE => '0'}
2 row(s) in 0.0850 seconds

hbase(main):030:0> put 'clicks','172.10.17.185','users:name','Steve'
0 row(s) in 0.0220 seconds

hbase(main):031:0> put 'clicks','172.10.17.185','users:name','Rock'
0 row(s) in 0.0130 seconds

hbase(main):032:0> put 'clicks','172.10.17.188','users:name','Voldmort'
0 row(s) in 0.0310 seconds

hbase(main):033:0> scan 'clicks',{NAME =>'users' , VERSIONS => 5 }
ROW                    COLUMN+CELL
 172.10.17.182           column=hits:hitcount, timestamp=1531963444698, value=60
 172.10.17.182           column=hits:hitcount, timestamp=1531963426695, value=50
 172.10.17.182           column=hits:hitcount, timestamp=1531963419665, value=30
 172.10.17.182           column=hits:hitcount, timestamp=1531963340892, value=20
 172.10.17.182           column=hits:hitcount, timestamp=1531962475897, value=180
 172.10.17.185           column=hits:hitcount, timestamp=1531963692962, value=200
 172.10.17.185           column=hits:hitcount, timestamp=1531963685896, value=100
 172.10.17.185           column=users:name, timestamp=1531964142093, value=Rock
 172.10.17.185           column=users:name, timestamp=1531964128913, value=Steve
 172.10.17.188           column=users:name, timestamp=1531964161245, value=Voldmort
3 row(s) in 0.0740 seconds




hbase(main):034:0> get 'clicks', '172.10.17.185',{COLUMN => 'users:name', VERSIONS =>3}
COLUMN                 CELL
 users:name             timestamp=1531964142093, value=Rock
 users:name             timestamp=1531964128913, value=Steve
1 row(s) in 0.1060 seconds

hbase(main):035:0> delete 'clicks', '172.10.17.185','hits:hitcount'
0 row(s) in 0.0750 seconds

hbase(main):038:0> scan 'clicks', {NAME =>'hits' , VERSIONS => 5 }
ROW                    COLUMN+CELL
 172.10.17.182           column=hits:hitcount, timestamp=1531963444698, value=60
 172.10.17.182           column=hits:hitcount, timestamp=1531963426695, value=50
```

172.10.17.182	column=hits:hitcount, timestamp=1531963419665, value=30
172.10.17.182	column=hits:hitcount, timestamp=1531963340892, value=20
172.10.17.182	column=hits:hitcount, timestamp=1531962475897, value=180
172.10.17.185	column=hits:hitcount, timestamp=1531963685896, value=100
172.10.17.185	column=users:name, timestamp=1531964142093, value=Rock
172.10.17.185	column=users:name, timestamp=1531964128913, value=Steve
172.10.17.188	column=users:name, timestamp=1531964161245, value=Voldmort

3 row(s) in 0.0520 seconds

So therefore, latest row with IP – 172.10.17.185 and value =200 gets deleted from the table on this.