Kudo is like Hbase:

Hbase is a column-oriented database management system which runs on top of HDFS (Hadoop Distribute File System). Hbase is not a relational data store, and it does not support structured query language like SQL.

As we know Hadoop distributed file system provides sequential data access, on the other hand Hbase is used for Random data access.

Hbase is the best solution for huge dataset for fast, random, real time, read and write. It is not suitable for small data. Hbase provides flexible schema for the table which is completely differ from RDBMS.

**Features of HBase:**

**Easy to use Java API for client access:** It provides easy to use Java API for programmatic access.

**Thrift gateway and a REST-ful Web services:** It also supports Thrift and REST API for non-Java front-ends.

**Block Cache and Bloom Filters:**HBase supports a Block Cache and Bloom Filters for high volume query optimization .

**Automatic failure support:**HBase with HDFS provides WAL (Write Ahead Log) across clusters which provides automatic failure support.

**Sorted rowkeys:** As searching is done on range of rows, HBase stores rowkeys in a lexicographical order. Using these sorted rowkeys and timestamp, we can build an optimized request.

HBase VS HDFS:

HDFS is a Java based distributed file system that allows you to store large data across multiple nodes in a Hadoop cluster. So, HDFS is an underlying storage system for storing the data in the distributed environment. HDFS is a file system, whereas HBase is a database (similar as NTFS and Mysql).

As Both HDFS and HBase stores any kind of data (i.e. structured, semi-structured and unstructured) in a distributed environment so lets look at the differences between HDFS file system and HBase, a NoSQL database.

HBase provides low latency access to small amounts of data within large data sets while HDFS provides high latency operations.

HBase supports random read and writes while HDFS supports WORM (Write once Read Many or Multiple times).

HDFS is basically or primarily accessed through MapReduce jobs while HBase is accessed through shell commands, Java API, REST, Avro of Thrift API.

**Difference between hbase and RDBMS:**

1. Hbase is column oriented database but RDBMS is Row Oriented Database.

2. Hbase is flexible schema database on the other hand RDBMS is fixed schema database.

3. Hbase doesn’t support triggers, secondary indexes etc.

4. Hbase is suitable for large amount of data on the other hand we RDBMS is best suitable for small data.

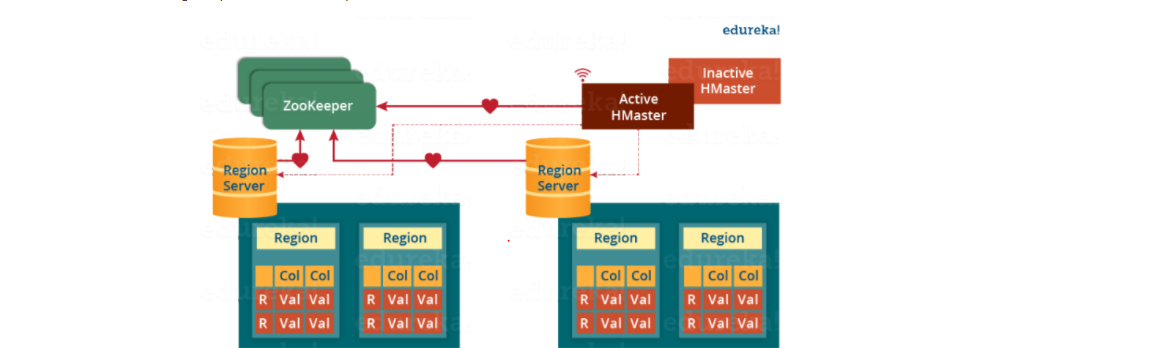
**Component of HBase:**

1. HMaster:- HMaster manages all the cluster.

2. HRegion Server: - it manages the data. These are slaves and running on data node.

3. Zookeeper: - Zookeeper is integrated with zookeeper. It maintains all the configuration information and keep track on region server. Hbase Region Server node registered with zookeeper. So HMaster gets all the information of region server from zookeeper.

**Components of HBase Architecture:**



HBase has three major components i.e., **HMaster Server**, **HBase Region Server, Regions** and **Zookeeper**.

all these Servers (HMaster, Region Server, Zookeeper) are placed to coordinate and manage Regions and perform various.

**Region**

A region contains all the rows between the start key and the end key assigned to that region. HBase tables can be divided into a number of regions in such a way that all the columns of a column family is stored in one region. Each region contains the rows in a sorted order.

Many regions are assigned to a Region Server, which is responsible for handling, managing, executing reads and writes operations on that set of regions.

A Region has a default size of 256MB which can be configured according to the need

A Region Server can serve approximately 1000 regions to the client.

**HMaster**

HBase HMaster performs DDL operations (create and delete tables) and assigns regions to the Region servers as you can see in the above image.

It coordinates and manages the Region Server (similar as NameNode manages DataNode in HDFS).

It assigns regions to the Region Servers on startup and re-assigns regions to Region Servers during recovery and load balancing.

It monitors all the Region Server’s instances in the cluster (with the help of Zookeeper) and performs recovery activities whenever any Region Server is down.

It provides an interface for creating, deleting and updating tables.

HBase has a distributed and huge environment where HMaster alone is not sufficient to manage everything. So, you would be wondering what helps HMaster to manage this huge environment? That’s where ZooKeeper comes into the picture

**ZooKeeper – The Coordinator**

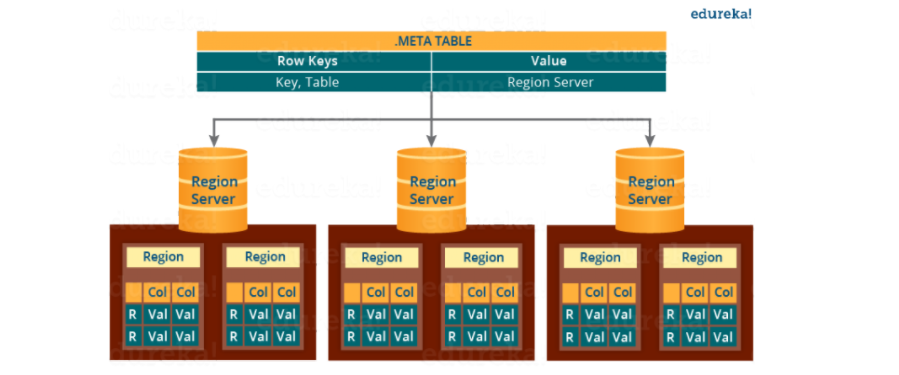
Zookeeper acts like a coordinator inside HBase distributed environment. It helps in maintaining server state inside the cluster by communicating through sessions.

Every Region Server along with HMaster Server sends continuous heartbeat at regular interval to Zookeeper and it checks which server is alive and available as mentioned in above image. It also provides server failure notifications so that, recovery measures(actions) can be executed.

The active HMaster sends heartbeats to the Zookeeper while the inactive HMaster listens for the notification send by active HMaster. If the active HMaster fails to send a heartbeat the session is deleted and the inactive HMaster becomes active.

While if a Region Server fails to send a heartbeat, the session is expired and all listeners are notified about it. Then HMaster performs suitable recovery actions.

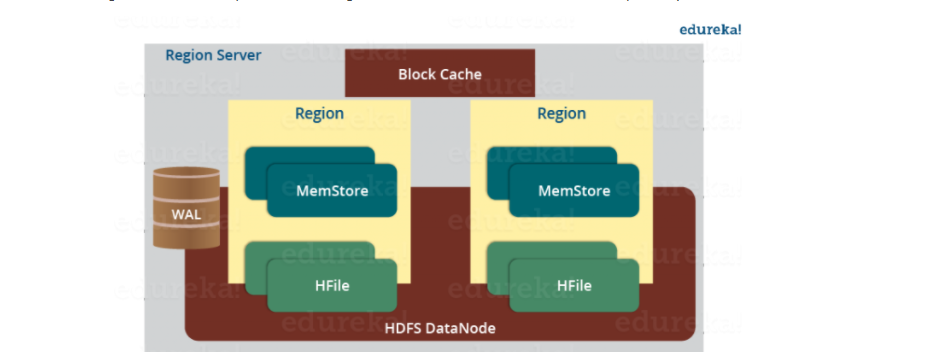
Zookeeper also maintains the. META Server’s path, which helps any client in searching for any region. The Client first has to check with. META Server in which Region Server a region belongs, and it gets the path of that Region Server.

**Meta Table:**

The META table is a special HBase catalog table. It maintains a list of all the Regions Servers in the HBase storage system

**META** file maintains the table in form of keys and values. Key represents the start key of the region and its id whereas the value contains the path of the Region Server.

**Components of Region Server**



A Region Server maintains various regions running on the top of [**HDFS**](https://www.edureka.co/blog/apache-hadoop-hdfs-architecture/). Components of a Region Server are

**WAL:** As you can conclude from the above image, Write Ahead Log (WAL) is a file attached to every Region Server inside the distributed environment. The WAL stores the new data that hasn’t been persisted or committed to the permanent storage. It is used in case of failure to recover the data sets.

**Block Cache:** From the above image, it is clearly visible that Block Cache resides in the top of Region Server. It stores the frequently read data in the memory. If the data in BlockCache is least recently used, then that data is removed from BlockCache.

**MemStore:** It is the write cache. It stores all the incoming data before committing it to the disk or permanent memory. There is one MemStore for each column family in a region. As you can see in the image, there are multiple MemStores for a region because each region contains multiple column families. The data is sorted in lexicographical order before committing it to the disk.

**HFile:** From the above figure you can see HFile is stored on HDFS. Thus it stores the actual cells on the disk. MemStore commits the data to HFile when the size of MemStore exceeds.

**How Search Initializes in HBase:**

As you know, Zookeeper stores the META table location. Whenever a client approaches with a read or writes requests to HBase following operation occurs:

The client retrieves the location of the META table from the ZooKeeper.

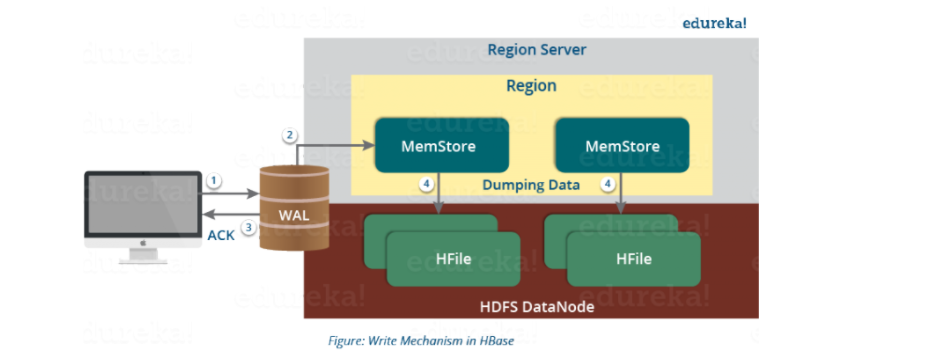
The client then requests for the location of the Region Server of corresponding row key from the META table to access it. The client caches this information with the location of the META Table.

Then it will get the row location by requesting from the corresponding Region Server.

For future references, the client uses its cache to retrieve the location of META table and previously read row key’s Region Server. Then the client will not refer to the META table, until and unless there is a miss because the region is shifted or moved. Then it will again request to the META server and update the cache.

As every time, clients does not waste time in retrieving the location of Region Server from META Server, thus, this saves time and makes the search process faster.

**HBase Write Mechanism**



The write mechanism goes through the following process sequentially (refer to the above image):

Step 1: Whenever the client has a write request, the client writes the data to the WAL (Write Ahead Log).

The edits are then appended at the end of the WAL file.

This WAL file is maintained in every Region Server and Region Server uses it to recover data which is not committed to the disk.

Step 2: Once data is written to the WAL, then it is copied to the MemStore.

There is one MemStore for each column family, and thus the updates are stored in a sorted manner for each column family.

Step 3: Once the data is placed in MemStore, then the client receives the acknowledgment.

Step 4: When the MemStore reaches the threshold, it dumps or commits the data into new HFile in a sorted manner. This HFile is stored in HDFS. HBase contains multiple HFiles for each Column Family.

**Compaction:**

**HBase**combines HFiles to reduce the storage and reduce the number of disk seeks needed for a read. This process is called **compaction**. Compaction chooses some HFiles from a region and combines them. There are two types of compaction as you can see in the above image.

**Minor Compaction**: HBase automatically picks smaller HFiles and recommits them to bigger HFiles as shown in the above image. This is called Minor Compaction. It performs merge sort for committing smaller HFiles to bigger HFiles. This helps in storage space optimization.

**Major Compaction:** As illustrated in the above image, in Major compaction, HBase merges and recommits the smaller HFiles of a region to a new HFile. In this process, the same column families are placed together in the new HFile. It drops deleted and expired cell in this process. It increases read performance.

But during this process, input-output disks and network traffic might get congested. This is known as write amplification. So, it is generally scheduled during low peak load timings.

Note:

The HFile indexes are loaded in memory whenever an HFile is opened. This helps in finding a record in a single seek.

**HBase Crash and Data Recovery**

Whenever a Region Server fails, ZooKeeper notifies to the HMaster about the failure.

Then HMaster distributes and allocates the regions of crashed Region Server to many active Region Servers. To recover the data of the MemStore of the failed Region Server, the HMaster distributes the WAL to all the Region Servers.

Each Region Server re-executes the WAL to build the MemStore for that failed region’s column family.

The data is written in chronological order (in a timely order) in WAL. Therefore, Re-executing that WAL means making all the change that were made and stored in the MemStore file.

So, after all the Region Servers executes the WAL, the MemStore data for all column family is recovered.

**CRUD Operations**

There are four main operations performed in HBase and they are:

Create

Read (Get and Scan)

Add (update)

Delete.

**Create Table Operation:**

To create a table, we need to create an instance of HBaseAdmin and then ask it to create the table and with at least one column family name in it. To perform DDL operations, you must use HBaseAdmin class instance.

|  |
| --- |
| import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.hbase.HBaseConfiguration;  import org.apache.hadoop.hbase.HColumnDescriptor;  import org.apache.hadoop.hbase.HTableDescriptor;  import org.apache.hadoop.hbase.MasterNotRunningException;  import org.apache.hadoop.hbase.ZooKeeperConnectionException;  import org.apache.hadoop.hbase.client.HBaseAdmin;  import org.apache.hadoop.hbase.util.Bytes;    public class CreateHbaseTable {  Configuration config = HBaseConfiguration.create();  public void createtable(String name,String[] colfamily) throws MasterNotRunningException,                                                 ZooKeeperConnectionException, IOException  {  HBaseAdmin admin = new HBaseAdmin(config);  HTableDescriptor des = new HTableDescriptor(Bytes.toBytes(name));  for(int i=0;i<colfamily.length;i++){  des.addFamily(new HColumnDescriptor(colfamily[i]));  }  if(admin.tableExists(name)){  System.out.println("Table already exist");  }  else{  admin.createTable(des);  System.out.println("Table: "+name+ " Sucessfully created");  }  }  public static void main(String args[]) throws MasterNotRunningException,                                 ZooKeeperConnectionException,IOException{  CreateHbaseTable op = new CreateHbaseTable();  String tablename = "Acadgild";  String[] familys = {"Emp\_name","sal"};  op.createtable(tablename, familys);  }  } |

**Why do we need to integrate Apache Hive with HBase?**

Hive has some limitations of high latency and HBase does not have analytical capabilities, integrating the two technologies together is the best solution. Often, people working with big data have this question in mind on –“How to use HBase from Hive? How well does using hive and HBase together work and what is the best way to use them?

Commonly HBase and Hive are used together on the same Hadoop cluster. Hive can be used as an ETL tool for batch inserts into HBase or to execute queries that join data present in HBase tables with the data present in HDFS files or in external data stores.

Hive can store information of hundreds of millions of users effortlessly, but, faces some difficulties when it comes to keeping the warehouse up to date with the latest information. Apache Hive uses HDFS as an underlying storage which comes with limitations like append-only, block-oriented storage. This makes it impossible to directly apply individual updates to warehouse tables. Up till now the only practical option to overcome this limitation is to pull the snapshots from MySQL databases and dump them to new Hive partitions. This expensive operation of pulling the data from one location to another location is not frequently practiced.(leading to stale data in the warehouse), and it also does not scale well as the data volume continues to shoot through the roof.

Key: ” is specified at the beginning of “hbase.columns.mapping” property which automatically maps to first column (id int) in Hive table

**CREATE EXTERNAL TABLE hbase\_table\_1(key int, value string)**

**STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'**

**WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,ColumnFamily:Column1, columnFalimy:column2")**

**TBLPROPERTIES ("hbase.table.name" = "xyz");**

**EXTERNAL**: This is used if the table in HBase exists already, or if the table in HBase is new and you want Hive to manage only the metadata and not the actual data.

**STORED BY:** The HBaseStorageHandler has to be used to handle the input and output from HBase.

**SERDEPROPERTIES**: Hive column to HBase ColumnFamily:Column mapping has to be specified here. In this example, key maps as a rowkey and value maps to val column of ColumnFamily cf1.

**TBLPROPERTIES**: Maps the HBase table name.

**SerDe Overview:**

SerDe is short for Serializer/Deserializer. Hive uses the SerDe interface for IO. The interface handles both serialization and deserialization and also interpreting the results of serialization as individual fields for processing.

A SerDe allows Hive to read in data from a table, and write it back out to HDFS in any custom format. Anyone can write there own SerDe for their own data formats.

**HBase with Hive integration and table creation from hive shell:**

HBase Integration with Hive and we will test this integration with the creation of some test hbase tables from hive shell and populate the contents of it from another hive table and finally verify these contents in hbase table.

Reasons to use Hive on HBase is that a lot of data sitting in HBase due to its usage in a real-time environment, but never used for analysis as there are less connectivity tools to HBase directly.

We will use storage handler mechanism to create hbase tables via hive. **HBaseStorageHandler** allows Hive DDL for managing table definitions in both Hive metastore and HBase’s catalog simultaneously and consistently.

We can not directly load data into hbase table “emp” with load data inpath hive command. We have to copy data into it from another Hive table. Lets create another test hive table with the same schema as hbase\_table\_emp and we will insert records into it with hive load data input command.

|  |
| --- |
| create table in habse:  create 'hbase\_emp', 'details', 'deptDetails'  Insert records into hbase table  put 'hbase\_emp', 'eid-01', 'details:fname', 'Nagendra'  put 'hbase\_emp', 'eid-01', 'details:lname', 'Mekala'  put 'hbase\_emp', 'eid-01', 'details:sal', '65000'  put 'hbase\_emp', 'eid-01', 'deptDetails:name', 'It'  put 'hbase\_emp', 'eid-01', 'deptDetails:location', 'Banglore'  put 'hbase\_emp', 'eid-02', 'details:fname', 'Scala'  put 'hbase\_emp', 'eid-02', 'details:lname', 'Mekala'  put 'hbase\_emp', 'eid-02', 'details:sal', '63000'  put 'hbase\_emp', 'eid-02', 'deptDetails:name', 'Scana developer'  put 'hbase\_emp', 'eid-02', 'deptDetails:location', 'Hyderbad'  create table in hive:  create external table hive\_emp(eid string, fname string,lname string,sal int,name string,location string) STORED BY "org.apache.hadoop.hive.hbase.HBaseStorageHandler" WITH SERDEPROPERTIES ("hbase.columns.mapping" =":key,details:fname,details:lname,details:sal,deptDetails:name,deptDetails:location") TBLPROPERTIES("hbase.table.name"="hbase\_emp");  Hive:  select \* from hive\_emp  insert into new recrod from hive;  insert into table hive\_emp(eid,fname,lname,sal,name,location) values('eid-03','NewRecord', 'New', 5000, 'newLocation', 'kalayandurg');  Hive:  select \* from hive\_emp  Hbase:  scan 'hbase\_emp';  **creation hbase table from hive shell with HBaseStorageHandler class:**  create Hbase table from hive:  create table hive\_emp\_temp(eid string, fname string,lname string,sal int,name string,location string) STORED BY "org.apache.hadoop.hive.hbase.HBaseStorageHandler" WITH SERDEPROPERTIES("hbase.columns.mapping" =":key,details:fname,details:lname,details:sal,deptDetails:name,deptDetails:location") TBLPROPERTIES("hbase.table.name"="hbase\_emp\_teamp");  insert into new recrod from hive;  insert into table hive\_emp\_temp(eid,fname,lname,sal,name,location) values('eid-03','NewRecord', 'New', 5000, 'newLocation', 'kalayandurg');  Hive:  select \* from hive\_emp\_temp;  Hbase:  scan 'hbase\_emp\_teamp';  insert records from hbase shell:  put 'hbase\_emp\_teamp', 'eid-01', 'details:fname', 'Nagendra'  put 'hbase\_emp\_teamp', 'eid-01', 'details:lname', 'Mekala'  put 'hbase\_emp\_teamp', 'eid-01', 'details:sal', '65000'  put 'hbase\_emp\_teamp', 'eid-01', 'deptDetails:name', 'It'  put 'hbase\_emp\_teamp', 'eid-01', 'deptDetails:location', 'Banglore'  Hive:  select \* from hive\_emp\_temp;  Hbase:  scan 'hbase\_emp\_teamp'; |

**Why do we need Filters:**

HBase can query data very quickly on demand but specific use cases may require to only return a subset of the scan results. Instead of scanning the entire dataset only to return a subset to the client, we can use Filters to get the data closer to what we need in less amount of time.

Thus, HBase has a set of predefined Filters as well as custom filters that we can use to scan and get filtered results from the HBase database.

How HBase Filters Work

There are two prominent ways to read data from HBase.

* Get is simply a Scan limited by the API to one row.
* Scan fetches zero or more rows of a table. By default, a Scan reads the entire table from start to end. We can limit our Scan results in several different ways, which affect the Scan’s load in terms of IO, network, or both, as well as processing load on the client side.

When reading data from HBase using Get or Scan operations, we can use custom filters to return a subset of results to the client. It does reduce network bandwidth and reduces the amount of data the client needs to process.

Filters are generally used when using the Java API and take zero or more arguments, in parentheses. Where the argument is a string, it is surrounded by single quotes (‘string’).

|  |
| --- |
| **20140315,1234567890,23,45,65**  **20140315,4567890123,43,32,56**  **20140315,6789012345,29,67,34**  **20140315,3214567890,45,78,20**  **20140316,1234567890,27,44,85**  **20140316,4567890123,12,30,54**  **import** java.io.BufferedReader;  **import** java.io.FileReader;  **import** java.io.IOException;  **import** java.util.HashMap;  **import** org.apache.hadoop.conf.Configuration;  **import** org.apache.hadoop.hbase.HBaseConfiguration;  **import** org.apache.hadoop.hbase.HColumnDescriptor;  **import** org.apache.hadoop.hbase.HTableDescriptor;  **import** org.apache.hadoop.hbase.KeyValue;  **import** org.apache.hadoop.hbase.MasterNotRunningException;  **import** org.apache.hadoop.hbase.ZooKeeperConnectionException;  **import** org.apache.hadoop.hbase.client.HBaseAdmin;  **import** org.apache.hadoop.hbase.client.HTable;  **import** org.apache.hadoop.hbase.client.Put;  **import** org.apache.hadoop.hbase.client.Result;  **import** org.apache.hadoop.hbase.client.ResultScanner;  **import** org.apache.hadoop.hbase.client.Scan;  **import** org.apache.hadoop.hbase.util.Bytes;  **class** SampleHbasePoc {  **private** Configuration conf = **null**;  **private** **static** HTable *htable* = **null**;  **public** SampleHbasePoc(String tableName, String colsName)  **throws** MasterNotRunningException, ZooKeeperConnectionException,  IOException {  conf = HBaseConfiguration.*create*();  *htable* = **new** ~~HTable~~(conf, tableName);  createHbaseTable(tableName, colsName);  }  **public** **void** createHbaseTable(String tblName, String colsName)  **throws** MasterNotRunningException, ZooKeeperConnectionException,  IOException {  System.***out***.println("Perform DDL operation in Habse Database");  HBaseAdmin admin = **new** ~~HBaseAdmin~~(conf);  HTableDescriptor ht = **new** ~~HTableDescriptor~~(Bytes.*toBytes*(tblName));  **if** (admin.tableExists(Bytes.*toBytes*(tblName))) {  System.***out***.println("Table alereday created in Hbase database");  } **else** {  HColumnDescriptor hc = **new** HColumnDescriptor(  Bytes.*toBytes*(colsName));  ht.addFamily(hc);  admin.createTable(ht);  }  }  **public** **void** addColumnEnter(String tableName, String columnFamily,  String rowKey, String columnName, String values) **throws** IOException {  System.***out***.println("Perform DML Put operation in Habse Database");  Put put = **new** Put(Bytes.*toBytes*(rowKey));  put.~~add~~(Bytes.*toBytes*(columnFamily), Bytes.*toBytes*(columnName), Bytes.*toBytes*(values));  *htable*.put(put);  }    @SuppressWarnings("deprecation")  **public** **void** getAllRecords(String startKey,String endKey){    Scan scan;  **if**(startKey == **null** || endKey==**null**){  scan = **new** Scan();  }**else**{  scan = **new** Scan(Bytes.*toBytes*(startKey),Bytes.*toBytes*(endKey));  }  //scan.setFilter(new PageFilter(25));  // scan.setFilter(new PageFilter(25));  // scan.setFilter(new PageFilter(25));  HashMap<String, HashMap<String,String>> outRec = **new** HashMap<String, HashMap<String,String>>();    **try** {  ResultScanner rs = *htable*.getScanner(scan);  **for**(Result rs1 : rs){  HashMap<String,String> keyVal = **new** HashMap<String,String>();  **for**(KeyValue kv : rs1.~~raw~~()){  String imi = **new** String(kv.~~getRow~~()).substring(10);  keyVal.put(**new** String(kv.~~getQualifier~~()), **new** String(kv.~~getValue~~()));  outRec.put(imi, keyVal);  **if**(keyVal.size() == 3){  System.***out***.println(imi +"\t" +"inComing mintes: "+  keyVal.get("c1") +"\t"+ "outGoing mintes: "+ keyVal.get("c2") +"\t"  +"messages: "+ keyVal.get("c3"));  }  }    }      } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    }  }  **public** **class** HbaseCreate {  **public** **static** **void** main(String[] args) **throws** MasterNotRunningException,  ZooKeeperConnectionException, IOException {  String tableName = "phoneData";  String columnFamily = "i";  String dataFileName = "/home/cloudera/Desktop/phoneData.txt";  SampleHbasePoc poc = **new** SampleHbasePoc(tableName, columnFamily);  FileReader fileReader = **new** FileReader(dataFileName);  BufferedReader br = **new** BufferedReader(fileReader);  String line = **null**;  **while** ((line = br.readLine()) != **null**) {  String[] values = line.split(",");  String rowKey = values[0] +"-"+ values[1];  poc.addColumnEnter(tableName, columnFamily, rowKey, "c1", values[2]);  poc.addColumnEnter(tableName, columnFamily, rowKey, "c2", values[3]);  poc.addColumnEnter(tableName, columnFamily, rowKey, "c3", values[4]);  }  br.close();  poc.getAllRecords("20140315", "20140316");  }  } |

**Hbase Servers run:**

**-lm:** long message

sudo jps –lm | grep hbase

sudo service hbase-master status

sudo service hbase-master start

sudo service hbase-regionserver start

sudo service zookeeper-server restart

**deleteColumns and deleteColumn:**

deleteColumnms: delete all the version of the cell.

deleteColumn : only deletes the latest

<http://www.geoinsyssoft.com/hivehbase-integration/>

<http://hadooptutorial.info/hbase-integration-with-hive/>

<http://www.geoinsyssoft.com/rdds-vs-dataframes-apache-spark/>

https://acadgild.com/blog/integrating-hive-hbase/

<http://dwgeek.com/apache-hbase-data-model-explanation.html/>

**Primary Skills:**   
(Please enter the Mandatory skills) :  
Proficient in designing efficient and robust ETL workflows  
Experience with Hadoop, Hive and Spark at scale  
Experience with Kinesis, Kafka or equivalent messaging systems at scale  
Proficient in Python(preferable) or Scala or Java  
Extensive hands on experience with AWS and cloud-based computing such as EC2, EMR, Kinesis and Redshift  
Excellent communication and collaboration skills  
Think at scale, and with high availability  
  
**Secondary Skills**Experience with streaming data pipelines using Spark streaming  
You have built a data pipeline and the infrastructure required to deploy machine learning algorithms and real-time analytics in low latency environments.  
Understanding of ci/cd automation  
Having shipped an SDK is a plus.

- Strong Knowledge in Spark & Scala/Java/Python(more than one)   
  
- Must have working knowledge of at least one NoSql DB (Cassandra, Mongo Db, Hbase, Redis etc.).   
  
- Knowledge on Kafka, Flume, Oozie, Zookeeper is added advantage.   
  
- Designing, building, installing, configuring and supporting Hadoop/Spark