APACHE

H
BASE
hadoop database

tutorialspoint
SIMPLY EASY LEARNING

About the Tutorial

This tutorial provides an introduction to HBase, the procedures to set up HBase on Hadoop File Systems, and ways to interact with HBase shell. It also describes how to connect to HBase using java, and how to perform basic operations on HBase using java.

Audience

This tutorial will help professionals aspiring to make a career in Big Data Analytics using Hadoop Framework. Software professionals, analytics Professionals, and ETL developers are the key beneficiaries of this course.

Prerequisites

Before you start proceeding with this tutorial, we assume that you are already aware of Hadoop's architecture and APIs, have experience in writing basic applications using java, and have a working knowledge of any database.

Copyright & Disclaimer

Table of Contents

iii

tutorialspoint
SIMPLY EASY LEARNING

Since 1970, RDBMS is the solution for data storage and maintenance related problems. After the advent of big data, companies realized the benefit of processing big data and started opting for solutions like Hadoop.

Hadoop uses distributed file system for storing big data, and MapReduce to process it. Hadoop excels in storing and processing of huge data of various formats such as arbitrary, semi-, or even unstructured.

Limitations of Hadoop

Hadoop can perform only batch processing, and data will be accessed only in a sequential manner. That means one has to search the entire dataset even for the simplest of jobs.

A huge dataset when processed results in another huge data set, which should also be processed sequentially. At this point, a new solution is needed to access any point of data in a single unit of time (random access).

Hadoop Random Access Databases

Applications such as HBase, Cassandra, couchDB, Dynamo, and MongoDB are some of the databases that store huge amounts of data and access the data in a random manner.

What is HBase?

HBase is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable.

HBase is a data model that is similar to Google's big table designed to provide quick random access to huge amounts of structured data. It leverages the fault tolerance provided by the Hadoop File System (HDFS).

It is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System.

One can store the data in HDFS either directly or through HBase. Data consumer reads/accesses the data in HDFS randomly using HBase. HBase sits on top of the Hadoop File System and provides read and write access.
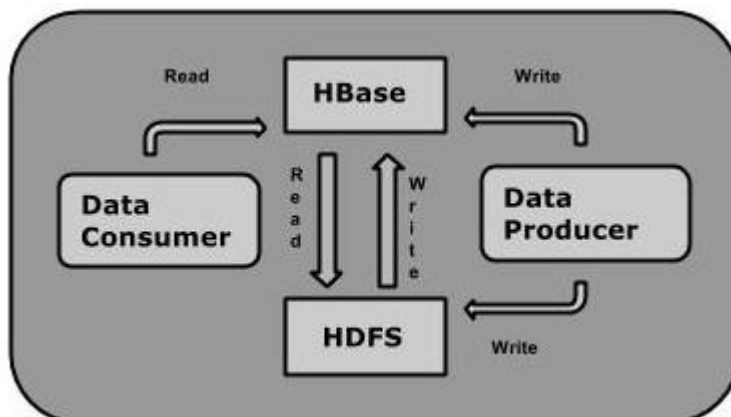
HBase and HDFS

HDFS	HBase
HDFS is a distributed file system suitable for storing large files.	HBase is a database built on top of the HDFS.
HDFS does not support fast individual record lookups.	HBase provides fast lookups for larger tables.
It provides high latency batch processing; no concept of batch processing.	It provides low latency access to single rows from billions of records (Random access).
It provides only sequential access of data.	HBase internally uses Hash tables and provides random access, and it stores the data in indexed HDFS files for faster lookups.

Storage Mechanism in HBase

HBase is a **column-oriented database** and the tables in it are sorted by row. The table schema defines only column families, which are the key value pairs. A table have multiple column families and each column family can have any number of columns. Subsequent column values are stored contiguously on the disk. Each cell value of the table has a timestamp. In short, in an HBase:

- Table is a collection of rows.

- Row is a collection of column families.

- Column family is a collection of columns.

- Column is a collection of key value pairs.

Given below is an example schema of table in HBase.

Rowid	Column Family			Column Family			Column Family			Column Family		
	clo1	col 2	col 3	col 1	col 2	col 3	col1	col2	col3	col1	col2	col3
1												
2												
3												

Column Oriented and Row Oriented

Column-oriented databases are those that store data tables as sections of columns of data, rather than as rows of data. Shortly, they will have column families.

Row-Oriented Database	Column-Oriented Database
It is suitable for Online Transaction Process (OLTP).	It is suitable for Online Analytical Processing (OLAP).
Such databases are designed for small number of rows and columns.	Column-oriented databases are designed for huge tables.

The following image shows column families in a column-oriented database:
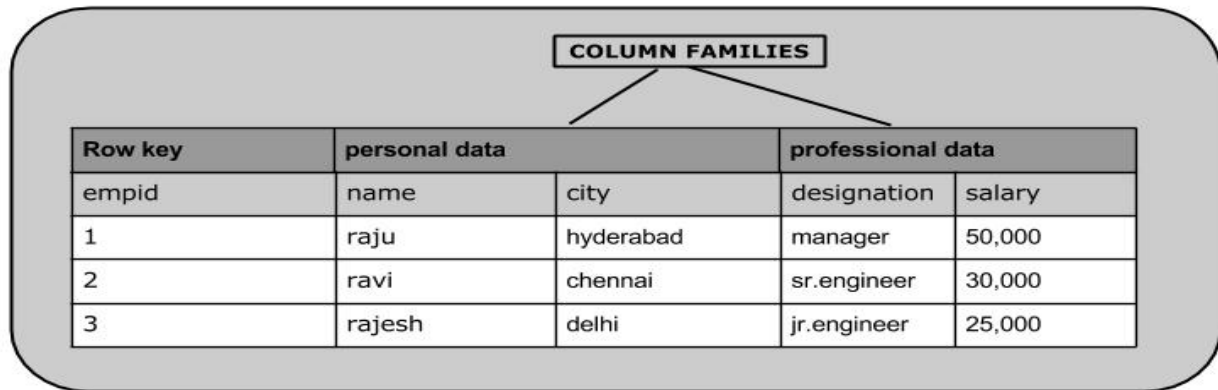
HBase and RDBMS

HBase	RDBMS
HBase is schema-less, it doesn't have the concept of fixed columns schema; defines only column families.	An RDBMS is governed by its schema, which describes the whole structure of tables.
It is built for wide tables. HBase is horizontally scalable.	It is thin and built for small tables. Hard to scale.
No transactions are there in HBase.	RDBMS is transactional.
It has de-normalized data.	It will have normalized data.
It is good for semi-structured as well as structured data.	It is good for structured data.

Features of HBase

- HBase is linearly scalable.

- It has automatic failure support.

- It provides consistent read and writes.

- It integrates with Hadoop, both as a source and a destination.

- It has easy java API for client.

- It provides data replication across clusters.

Where to Use HBase

- Apache HBase is used to have random, real-time read/write access to Big Data.

- It hosts very large tables on top of clusters of commodity hardware.

- Apache HBase is a non-relational database modeled after Google's Bigtable. Bigtable acts up on Google File System, likewise Apache HBase works on top of Hadoop and HDFS.

Applications of HBase

- It is used whenever there is a need to write heavy applications.

- HBase is used whenever we need to provide fast random access to available data.

- Companies such as Facebook, Twitter, Yahoo, and Adobe use HBase internally.

HBase History

Year	Event
Nov 2006	Google released the paper on BigTable.
Feb 2007	Initial HBase prototype was created as a Hadoop contribution.
Oct 2007	The first usable HBase along with Hadoop 0.15.0 was released.
Jan 2008	HBase became the sub project of Hadoop.
Oct 2008	HBase 0.18.1 was released.
Jan 2009	HBase 0.19.0 was released.
Sept 2009	HBase 0.20.0 was released.
May 2010	HBase became Apache top-level project.

2. ARCHITECTURE

HBase Architecture

In HBase, tables are split into regions and are served by the region servers. Regions are vertically divided by column families into "Stores". Stores are saved as files in HDFS. Shown below is the architecture of HBase.

Note: The term 'store' is used for regions to explain the storage structure.
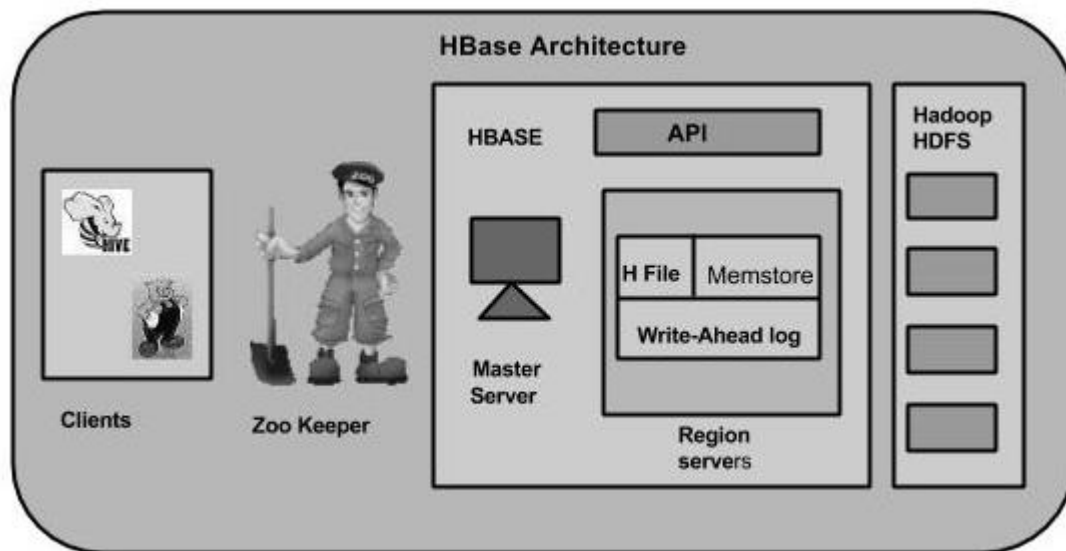
HBase has three major components: the client library, a master server, and region servers. Region servers can be added or removed as per requirement.

Master Server

The master server -

- Assigns regions to the region servers and takes the help of Apache ZooKeeper for this task.

- Handles load balancing of the regions across region servers. It unloads the busy servers and shifts the regions to less occupied servers.

- Maintains the state of the cluster by negotiating the load balancing.

- Is responsible for schema changes and other metadata operations such as creation of tables and column families.

Regions

Regions are nothing but tables that are split up and spread across the region servers.

Region server

The region servers have regions that -

- Communicate with the client and handle data-related operations.

- Handle read and write requests for all the regions under it.

- Decide the size of the region by following the region size thresholds.

When we take a deeper look into the region server, it contain regions and stores as shown below:
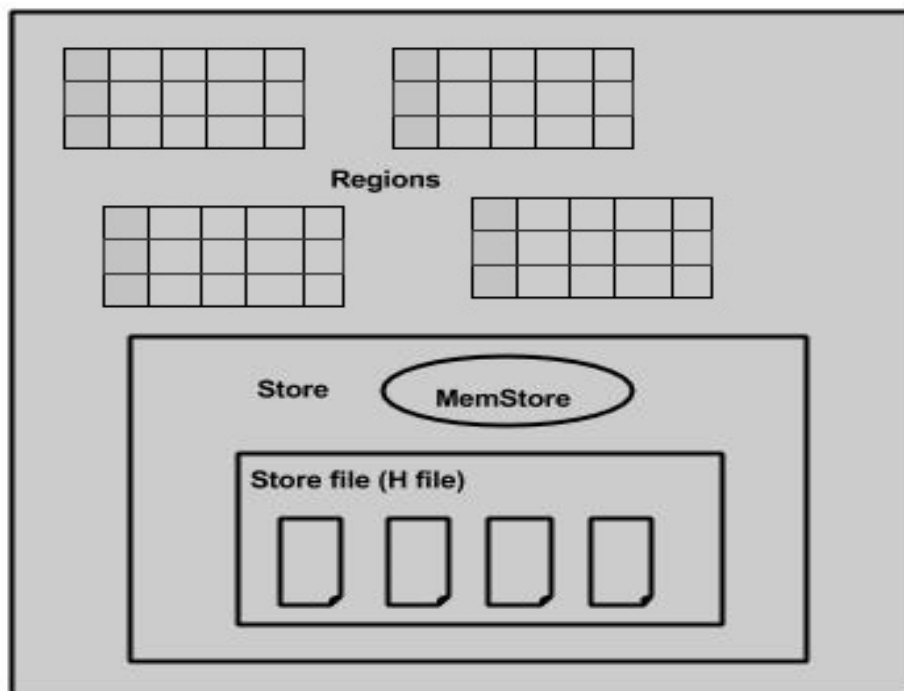
The store contains memory store and HFiles. Memstore is just like a cache memory. Anything that is entered into the HBase is stored here initially. Later, the data is transferred and saved in Hfiles as blocks and the memstore is flushed.

Zookeeper

- Zookeeper is an open-source project that provides services like maintaining configuration information, naming, providing distributed synchronization, etc.

- Zookeeper has ephemeral nodes representing different region servers. Master servers use these nodes to discover available servers.

- In addition to availability, the nodes are also used to track server failures or network partitions.

- Clients communicate with region servers via zookeeper.

- In pseudo and standalone modes, HBase itself will take care of zookeeper.

This chapter explains how HBase is installed and initially configured. Java and Hadoop are required to proceed with HBase, so you have to download and install java and Hadoop in your system.

Pre-Installation Setup

Before installing Hadoop into Linux environment, we need to set up Linux using **ssh** (Secure Shell). Follow the steps given below for setting up the Linux environment.

Creating a User

First of all, it is recommended to create a separate user for Hadoop to isolate the Hadoop file system from the Unix file system. Follow the steps given below to create a user.

1. Open the root using the command "su".

2. Create a user from the root account using the command "useradd username".

3. Now you can open an existing user account using the command "su username".

Open the Linux terminal and type the following commands to create a user.

```
$ su
password:
# useradd hadoop
# passwd hadoop
New passwd:
Retype new passwd
```

SSH Setup and Key Generation

SSH setup is required to perform different operations on the cluster such as start, stop, and distributed daemon shell operations. To authenticate different users of Hadoop, it is required to provide public/private key pair for a Hadoop user and share it with different users.

The following commands are used to generate a key value pair using SSH. Copy the public keys form id_rsa.pub to authorized_keys, and provide owner, read and write permissions to authorized_keys file respectively.

```
$ ssh-keygen -t rsa
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys
```

Verify ssh

```
ssh localhost
```

Installing Java

Java is the main prerequisite for Hadoop and HBase. First of all, you should verify the existence of java in your system using "java -version". The syntax of java version command is given below.

```
$ java -version
```

If everything works fine, it will give you the following output.

```
java version "1.7.0_71"

Java(TM) SE Runtime Environment (build 1.7.0_71-b13)

Java HotSpot(TM) Client VM (build 25.0-b02, mixed mode)
```

If java is not installed in your system, then follow the steps given below for installing java.

Step 1

Download java (JDK <latest version> - X64.tar.gz) by visiting the following link http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html.

Then **jdk-7u71-linux-x64.tar.gz** will be downloaded into your system.

Step 2

Generally you will find the downloaded java file in Downloads folder. Verify it and extract the **jdk-7u71-linux-x64.gz** file using the following commands.

```
$ cd Downloads/
$ ls
jdk-7u71-linux-x64.gz
```

```
$ tar zxf jdk-7u71-linux-x64.gz

$ ls

jdk1.7.0_71    jdk-7u71-linux-x64.gz
```

Step 3

To make java available to all the users, you have to move it to the location "/usr/local/". Open root and type the following commands.

```
$ su

password:

# mv jdk1.7.0_71 /usr/local/

# exit
```

Step 4

For setting up **PATH** and **JAVA_HOME** variables, add the following commands to **~/.bashrc** file.

```
export JAVA_HOME=/usr/local/jdk1.7.0_71

export PATH= $PATH:$JAVA_HOME/bin
```

Now apply all the changes into the current running system.

```
$ source ~/.bashrc
```

Step 5

Use the following commands to configure java alternatives:

```
# alternatives --install /usr/bin/java java usr/local/java/bin/java 2

# alternatives --install /usr/bin/javac javac usr/local/java/bin/javac 2

# alternatives --install /usr/bin/jar jar usr/local/java/bin/jar 2


# alternatives --set java usr/local/java/bin/java

# alternatives --set javac usr/local/java/bin/javac

# alternatives --set jar usr/local/java/bin/jar
```

Now verify the installation using the command **java -version** from the terminal as explained above.

Downloading Hadoop

After installing java, you have to install Hadoop. First of all, verify the existence of Hadoop using " Hadoop version " command as shown below.

```
hadoop version
```

If everything works fine, it will give you the following output.

```
Hadoop 2.6.0

Compiled by jenkins on 2014-11-13T21:10Z

Compiled with protoc 2.5.0

From source with checksum 18e43357c8f927c0695f1e9522859d6a

This command was run using
/home/hadoop/hadoop/share/hadoop/common/hadoop-common-2.6.0.jar
```

If your system is unable to locate Hadoop, then download Hadoop in your system. Follow the commands given below to do so.

Download and extract hadoop-2.6.0 from Apache Software Foundation using the following commands.

```
$ su

password:

# cd /usr/local

# wget http://mirrors.advancedhosters.com/apache/hadoop/common/hadoop-
2.6.0/hadoop-2.6.0-src.tar.gz

# tar xzf hadoop-2.6.0-src.tar.gz

# mv hadoop-2.6.0/* hadoop/

# exit
```

Installing Hadoop

Install Hadoop in any of the required mode. Here, we are demonstrating HBase functionalities in pseudo distributed mode, therefore install Hadoop in pseudo distributed mode.

The following steps are used for installing **Hadoop 2.4.1**.

Step 1 - Setting up Hadoop

You can set Hadoop environment variables by appending the following commands to **~/.bashrc** file.

```
export HADOOP_HOME=/usr/local/hadoop

export HADOOP_MAPRED_HOME=$HADOOP_HOME

export HADOOP_COMMON_HOME=$HADOOP_HOME

export HADOOP_HDFS_HOME=$HADOOP_HOME

export YARN_HOME=$HADOOP_HOME

export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native

export PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HOME/bin

export HADOOP_INSTALL=$HADOOP_HOME
```

Now apply all the changes into the current running system.

```
$ source ~/.bashrc
```

Step 2 - Hadoop Configuration

You can find all the Hadoop configuration files in the location "$HADOOP_HOME/etc/hadoop". You need to make changes in those configuration files according to your Hadoop infrastructure.

```
$ cd $HADOOP_HOME/etc/hadoop
```

In order to develop Hadoop programs in java, you have to reset the java environment variable in **hadoop-env.sh** file by replacing **JAVA_HOME** value with the location of java in your system.

```
export JAVA_HOME=/usr/local/jdk1.7.0_71
```

You will have to edit the following files to configure Hadoop.

core-site.xml

The **core-site.xml** file contains information such as the port number used for Hadoop instance, memory allocated for file system, memory limit for storing data, and the size of Read/Write buffers.

Open core-site.xml and add the following properties in between the <configuration>and </configuration> tags.

```
<configuration>
```

```
  <property>
     <name>fs.default.name</name>
     <value>hdfs://localhost:9000</value>
  </property>

</configuration>
```

hdfs-site.xml

The **hdfs-site.xml** file contains information such as the value of replication data, namenode path, and datanode path of your local file systems, where you want to store the Hadoop infrastructure.

Let us assume the following data.

```
dfs.replication (data replication value) = 1

(In the below given path /hadoop/ is the user name.

hadoopinfra/hdfs/namenode is the directory created by hdfs file system.)

namenode path = //home/hadoop/hadoopinfra/hdfs/namenode

(hadoopinfra/hdfs/datanode is the directory created by hdfs file
system.)

datanode path = //home/hadoop/hadoopinfra/hdfs/datanode
```

Open this file and add the following properties in between the <configuration>, </configuration> tags.

```
<configuration>
  <property>
     <name>dfs.replication</name>
     <value>1</value>
  </property>

  <property>
     <name>dfs.name.dir</name>
     <value>file:///home/hadoop/hadoopinfra/hdfs/namenode</value>
  </property>

  <property>
     <name>dfs.data.dir</name>
     <value>file:///home/hadoop/hadoopinfra/hdfs/datanode</value>
  </property>

</configuration>
```

Note: In the above file, all the property values are user-defined and you can make changes according to your Hadoop infrastructure.

yarn-site.xml

This file is used to configure yarn into Hadoop. Open the yarn-site.xml file and add the following property in between the <configuration>, </configuration> tags in this file.

```
<configuration>

 <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>

</configuration>
```

mapred-site.xml

This file is used to specify which MapReduce framework we are using. By default, Hadoop contains a template of yarn-site.xml. First of all, it is required to copy the file from **mapred-site.xml.template** to **mapred-site.xml** file using the following command.

```
$ cp mapred-site.xml.template mapred-site.xml
```

Open **mapred-site.xml** file and add the following properties in between the <configuration> and </configuration> tags.

```
<configuration>

 <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>

</configuration>
```

Verifying Hadoop Installation

The following steps are used to verify the Hadoop installation.

Step 1 - Name Node Setup

Set up the namenode using the command "hdfs namenode -format" as follows.

```
$ cd ~
```

```
$ hdfs namenode -format
```

The expected result is as follows.

```
10/24/14 21:30:55 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = localhost/192.168.1.11
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 2.4.1
...
...
10/24/14 21:30:56 INFO common.Storage: Storage directory
/home/hadoop/hadoopinfra/hdfs/namenode has been successfully formatted.
10/24/14 21:30:56 INFO namenode.NNStorageRetentionManager: Going to
retain 1 images with txid >= 0
10/24/14 21:30:56 INFO util.ExitUtil: Exiting with status 0
10/24/14 21:30:56 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at localhost/192.168.1.11
************************************************************/
```

Step 2 - Verifying Hadoop dfs

The following command is used to start dfs. Executing this command will start your Hadoop file system.

```
$ start-dfs.sh
```

The expected output is as follows.

```
10/24/14 21:37:56
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/hadoop/hadoop-
2.4.1/logs/hadoop-hadoop-namenode-localhost.out
localhost: starting datanode, logging to /home/hadoop/hadoop-
2.4.1/logs/hadoop-hadoop-datanode-localhost.out
Starting secondary namenodes [0.0.0.0]
```

Step 3 - Verifying Yarn Script

The following command is used to start the yarn script. Executing this command will start your yarn daemons.

```
$ start-yarn.sh
```

The expected output is as follows.

```
starting yarn daemons

starting resourcemanager, logging to /home/hadoop/hadoop-

2.4.1/logs/yarn-hadoop-resourcemanager-localhost.out

localhost: starting nodemanager, logging to /home/hadoop/hadoop-

2.4.1/logs/yarn-hadoop-nodemanager-localhost.out
```

Step 4 - Accessing Hadoop on Browser

The default port number to access Hadoop is 50070. Use the following url to get Hadoop services on your browser.

```
http://localhost:50070/
```


Step 5 - Verify all Applications of Cluster

The default port number to access all the applications of cluster is 8088. Use the following url to visit this service.

```
http://localhost:8088/
```


Installing HBase

We can install HBase in any of the three modes: Standalone mode, Pseudo Distributed mode, and Fully Distributed mode.

Installing HBase in Standalone Mode

Download the latest stable version of HBase form http://www.interior-dsgn.com/apache/hbase/stable/ using "wget" command, and extract it using the tar "zxvf" command. See the following command.

```
$cd usr/local/

$wget http://www.interior-dsgn.com/apache/hbase/stable/hbase-0.98.8-
hadoop2-bin.tar.gz

$tar -zxvf hbase-0.98.8-hadoop2-bin.tar.gz
```

Shift to super user mode and move the HBase folder to /usr/local as shown below.

```
$su

$password: enter your password here

mv hbase-0.99.1/* Hbase/
```

Configuring HBase in Standalone Mode

Before proceeding with HBase, you have to edit the following files and configure HBase.

hbase-env.sh

Set the java Home for HBase and open **hbase-env.sh** file from the conf folder. Edit JAVA_HOME environment variable and change the existing path to your current JAVA_HOME variable as shown below.

```
cd /usr/local/Hbase/conf
gedit hbase-env.sh
```

This will open the env.sh file of HBase. Now replace the existing **JAVA_HOME** value with your current value as shown below.

```
export JAVA_HOME=/usr/lib/jvm/java-1.7.0
```

hbase-site.xml

This is the main configuration file of HBase. Set the data directory to an appropriate location by opening the HBase home folder in /usr/local/HBase. Inside the conf folder, you will find several files, open the **hbase-site.xml** file as shown below.

```
#cd /usr/local/HBase/
#cd conf
# gedit hbase-site.xml
```

Inside the **hbase-site.xml** file, you will find the <configuration> and </configuration> tags. Within them, set the HBase directory under the property key with the name "hbase.rootdir" as shown below.

```
<configuration>


//Here you have to set the path where you want HBase to store its files.

<property>

   <name>hbase.rootdir</name>

   <value>file:/home/hadoop/HBase/HFiles</value>

</property>


//Here you have to set the path where you want HBase to store its built
in zookeeper files.

<property>
   <name>hbase.zookeeper.property.dataDir</name>
   <value>/home/hadoop/zookeeper</value>
</property>


</configuration>
```

With this, the HBase installation and configuration part is successfully complete. We can start HBase by using **start-hbase.sh** script provided in the bin folder of HBase. For that, open HBase Home Folder and run HBase start script as shown below.

```
$cd /usr/local/HBase/bin

$./start-hbase.sh
```

If everything goes well, when you try to run HBase start script, it will prompt you a message saying that HBase has started.

```
starting master, logging to /usr/local/HBase/bin/../logs/hbase-tp-
master-localhost.localdomain.out
```

Installing HBase in Pseudo-Distributed Mode

Let us now check how HBase is installed in pseudo-distributed mode.

Configuring HBase

Before proceeding with HBase, configure Hadoop and HDFS on your local system or on a remote system and make sure they are running. Stop HBase if it is running.

hbase-site.xml

Edit hbase-site.xml file to add the following properties.

```
<property>

   <name>hbase.cluster.distributed</name>

   <value>true</value>

</property
```

It will mention in which mode HBase should be run. In the same file from the local file system, change the hbase.rootdir, your HDFS instance address, using the hdfs://// URI syntax. We are running HDFS on the localhost at port 8030.

```
<property>
   <name>hbase.rootdir</name>
```

```
   <value>hdfs://localhost:8030/hbase</value>
</property>
```

Starting HBase

After configuration is over, browse to HBase home folder and start HBase using the following command.

```
$cd /usr/local/HBase

$bin/start-hbase.sh
```

Note: Before starting HBase, make sure Hadoop is running.

Checking the HBase Directory in HDFS

HBase creates its directory in HDFS. To see the created directory, browse to Hadoop bin and type the following command.

```
$ ./bin/hadoop fs -ls /hbase
```

If everything goes well, it will give you the following output.

```
Found 7 items
drwxr-xr-x   - hbase users          0 2014-06-25 18:58 /hbase/.tmp
drwxr-xr-x   - hbase users          0 2014-06-25 21:49 /hbase/WALs
drwxr-xr-x   - hbase users          0 2014-06-25 18:48 /hbase/corrupt
drwxr-xr-x   - hbase users          0 2014-06-25 18:58 /hbase/data
-rw-r--r--   3 hbase users         42 2014-06-25 18:41 /hbase/hbase.id
-rw-r--r--   3 hbase users          7 2014-06-25 18:41 /hbase/hbase.version
drwxr-xr-x   - hbase users          0 2014-06-25 21:49 /hbase/oldWALs
```

Starting and Stopping a Master

Using the "local-master-backup.sh" you can start up to 10 servers. Open the home folder of HBase, master and execute the following command to start it.

```
$ ./bin/local-master-backup.sh 2 4
```

To kill a backup master, you need its process id, which will be stored in a file named **"/tmp/hbase-USER-X-master.pid."** you can kill the backup master using the following command.

21

```
$ cat /tmp/hbase-user-1-master.pid |xargs kill -9
```

Starting and Stopping Region Servers

You can run multiple region servers from a single system using the following command.

```
$ .bin/local-regionservers.sh start 2 3
```

To stop a region server, use the following command.

```
$ .bin/local-regionservers.sh stop 3
```

Starting HBase Shell

Given below are the sequence of steps that are to be followed before starting the HBase shell. Open the terminal, and login as super user.

Strat Hadoop File System

Browse through Hadoop home sbin folder and start Hadoop file system as shown below.

```
$cd $HADOOP_HOME/sbin

$start-all.sh
```

Start HBase

Browse through the HBase root directory bin folder and start HBase.

```
$cd /usr/local/HBase

$./bin/start-hbase.sh
```

Start HBase Master Server

This will be the same directory. Start it as shown below.

```
$./bin/local-master-backup.sh start 2 (number signifies specific
server.)
```

Start Region

Start the region server as shown below.

```
$./bin/./local-regionservers.sh start 3
```

Start HBase Shell

You can start HBase shell using the following command.

```
$cd bin

$./hbase shell
```

This will give you the HBase Shell Prompt as shown below.

```
2014-12-09 14:24:27,526 INFO [main] Configuration.deprecation:
hadoop.native.lib is deprecated. Instead, use io.native.lib.available

HBase Shell; enter 'help<RETURN>' for list of supported commands.

Type "exit<RETURN>" to leave the HBase Shell

Version 0.98.8-hadoop2, r6cfc8d064754251365e070a10a82eb169956d5fe, Fri
Nov 14 18:26:29 PST 2014


hbase(main):001:0>
```

HBase Web Interface

To access the web interface of HBase, type the following url in the browser.

```
http://localhost:60010
```

This interface lists your currently running Region servers, backup masters and HBase tables.

HBase Region servers and Backup Masters

HBase Tables

This chapter explains how to start HBase interactive shell that comes along with HBase.

HBase Shell

HBase contains a shell using which you can communicate with HBase. HBase uses the Hadoop File System to store its data. It will have a master server and region servers. The data storage will be in the form of regions (tables). These regions will be split up and stored in region servers.

The master server manages these region servers and all these tasks take place on HDFS. Given below are some of the commands supported by HBase Shell.

General Commands

- **status:** Provides the status of HBase, for example, the number of servers.
- **version:** Provides the version of HBase being used.
- **table_help:** Provides help for table-reference commands.
- **whoami:** Provides information about the user.

Data Definition Language

These are the commands that operate on the tables in HBase.

- **create:** Creates a table.
- **list**: Lists all the tables in HBase.
- **disable**: Disables a table.
- **is_disabled:** Verifies whether a table is disabled.
- **enable:** Enables a table.
- **is_enabled**: Verifies whether a table is enabled.
- **describe:** Provides the description of a table.
- **alter:** Alters a table.
- **exists:** Verifies whether a table exists.
- **drop:** Drops a table from HBase.

- **drop_all:** Drops the tables matching the 'regex' given in the command.

- **Java Admin API:** Prior to all the above commands, Java provides an Admin API to achieve DDL functionalities through programming. Under **org.apache.hadoop.hbase.client** package, HBaseAdmin and HTableDescriptor are the two important classes in this package that provide DDL functionalities.

Data Manipulation Language

- **put:** Puts a cell value at a specified column in a specified row in a particular table.

- **get:** Fetches the contents of row or a cell.

- **delete:** Deletes a cell value in a table.

- **deleteall:** Deletes all the cells in a given row.

- **scan:** Scans and returns the table data.

- **count:** Counts and returns the number of rows in a table.

- **truncate:** Disables, drops, and recreates a specified table.

- **Java client API:** Prior to all the above commands, Java provides a client API to achieve DML functionalities, **CRUD** (Create Retrieve Update Delete) operations and more through programming, under org.apache.hadoop.hbase.client package. **HTable Put** and **Get** are the important classes in this package.

Starting HBase Shell

To access the HBase shell, you have to navigate to the HBase home folder.

```
cd /usr/localhost/
cd Hbase
```

You can start the HBase interactive shell using "**hbase shell**" command as shown below.

```
./bin/hbase shell
```

If you have successfully installed HBase in your system, then it gives you the HBase shell prompt as shown below.

```
HBase Shell; enter 'help<RETURN>' for list of supported commands.
```

```
Type "exit<RETURN>" to leave the HBase Shell

Version 0.94.23, rf42302b28aceaab773b15f234aa8718fff7eea3c, Wed Aug 27
00:54:09 UTC 2014


hbase(main):001:0>
```

To exit the interactive shell command at any moment, type exit or use <ctrl+c>. Check the shell functioning before proceeding further. Use the **list** command for this purpose. **List** is a command used to get the list of all the tables in HBase. First of all, verify the installation and the configuration of HBase in your system using this command as shown below.

```
hbase(main):001:0> list
```

When you type this command, it gives you the following output.

```
hbase(main):001:0> list
TABLE
```

5. GENERAL COMMANDS

The general commands in HBase are status, version, table_help, and whoami. This chapter explains these commands.

status

This command returns the status of the system including the details of the servers running on the system. Its syntax is as follows:

```
hbase(main):009:0> status
```

If you execute this command, it returns the following output.

```
hbase(main):009:0> status
3 servers, 0 dead, 1.3333 average load
```

version

This command returns the version of HBase used in your system. Its syntax is as follows:

```
hbase(main):010:0> version
```

If you execute this command, it returns the following output.

```
hbase(main):009:0> version
0.98.8-hadoop2, r6cfc8d064754251365e070a10a82eb169956d5fe, Fri Nov 14
18:26:29 PST 2014
```

table_help

This command guides you what and how to use table-referenced commands. Given below is the syntax to use this command.

```
hbase(main):02:0> table_help
```

When you use this command, it shows help topics for table-related commands. Given below is the partial output of this command.

```
hbase(main):002:0> table_help

Help for table-reference commands.


You can either create a table via 'create' and then manipulate the table
via commands like 'put', 'get', etc.

See the standard help information for how to use each of these commands.


However, as of 0.96, you can also get a reference to a table, on which
you can invoke commands.

For instance, you can get create a table and keep around a reference to
it via:


    hbase> t = create 't', 'cf'…...
```

whoami

This command returns the user details of HBase. If you execute this command, returns the current HBase user as shown below.

```
hbase(main):008:0> whoami

hadoop (auth:SIMPLE)

     groups: hadoop
```

6. ADMIN API

HBase is written in java, therefore it provides java API to communicate with HBase. Java API is the fastest way to communicate with HBase. Given below is the referenced java Admin API that covers the tasks used to manage tables.

Class HBaseAdmin

HBaseAdmin is a class representing the Admin. This class belongs to the **org.apache.hadoop.hbase.client** package. Using this class, you can perform the tasks of an administrator. You can get the instance of Admin using **Connection.getAdmin()** method.

Methods and Description

S. No.	Methods and Description
1	**void createTable(HTableDescriptor desc)** Creates a new table.
2	**void createTable(HTableDescriptor desc, byte[][] splitKeys)** Creates a new table with an initial set of empty regions defined by the specified split keys.
3	**void deleteColumn(byte[] tableName, String columnName)** Deletes a column from a table.
4	**void deleteColumn(String tableName, String columnName)** Delete a column from a table.
5	**void deleteTable(String tableName)** Deletes a table.

Class Descriptor

This class contains the details about an HBase table such as:

- the descriptors of all the column families,

31

- if the table is a catalog table,

- if the table is read only,

- the maximum size of the mem store,

- when the region split should occur,

- co-processors associated with it, etc.

Constructors

S. No.	Constructor and summary
1	**HTableDescriptor(TableName name)** Constructs a table descriptor specifying a TableName object.

Methods and Description

S. No.	Methods and Description
1	**HTableDescriptor addFamily(HColumnDescriptor family)** Adds a column family to the given descriptor.

Setting the Classpath

Before proceeding with programming, set the classpath to HBase libraries in **.bashrc** file. Open **.bashrc** in any of the editors as shown below.

```
$ gedit ~/.bashrc
```

Set classpath for HBase libraries (lib folder in HBase) in it as shown below.

```
export CLASSPATH=$CLASSPATH://home/hadoop/hbase/lib/*
```

This is to prevent the "class not found" exception while accessing the HBase using java API.

Creating Table

You can create a table using the **create** command, here you must specify the table name and the Column Family name. The **syntax** to create a table in HBase shell is shown below.

```
create '<table name>','<column family>'
```

Example

Given below is a sample schema of a table named emp. It has two column families: "personal data" and "professional data".

Row key	personal data	professional data

You can create this table in HBase shell as shown below.

```
hbase(main):002:0> create 'emp', 'personal data', 'professional data'
```

And it will give you the following output.

```
0 row(s) in 1.1300 seconds

=> Hbase::Table - emp
```

Verifying the Creation

You can verify whether the table is created using the **list** command as shown below. Here you can observe the created emp table.

```
hbase(main):002:0> list

TABLE
```

```
emp

2 row(s) in 0.0340 seconds
```

Creating a Table Using java API

You can create a table in HBase using the **createTable()** method of **HBaseAdmin** class. This class belongs to the **org.apache.hadoop.hbase.client** package. Given below are the steps to create a table in HBase using java API.

Step 1: Instantiate HBaseAdmin

This class requires the Configuration object as a parameter, therefore initially instantiate the Configuration class and pass this instance to HBaseAdmin.

```
Configuration conf = HBaseConfiguration.create();

HBaseAdmin admin = new HBaseAdmin(conf);
```

Step 2: Create TableDescriptor

HTableDescriptor is a class that belongs to the **org.apache.hadoop.hbase** class. This class is like a container of table names and column families.

```
//creating table descriptor

HTableDescriptor table = new HTableDescriptor(toBytes("Table name"));


//creating column family descriptor

HColumnDescriptor family = new HColumnDescriptor(toBytes("column family"));


//adding coloumn family to HTable

table.addFamily(family);
```

Step 3: Execute through Admin

Using the **createTable()** method of **HBaseAdmin** class, you can execute the created table in Admin mode.

```
admin.createTable(table);
```

Given below is the complete program to create a table via admin.

```
import java.io.IOException;

import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.TableName;

public class CreateTable {

   public static void main(String[] args) throws IOException {

      // Instantiating configuration class
      Configuration con = HBaseConfiguration.create();

      // Instantiating HbaseAdmin class
      HBaseAdmin admin = new HBaseAdmin(con);

      // Instantiating table descriptor class
      HTableDescriptor tableDescriptor = new
       TableDescriptor(TableName.valueOf("emp"));

      // Adding column families to table descriptor
      tableDescriptor.addFamily(new HColumnDescriptor("personal"));
      tableDescriptor.addFamily(new HColumnDescriptor("professional"));

      // Execute the table through admin
      admin.createTable(tableDescriptor);
      System.out.println(" Table created ");

   }
```

```
}
```

Compile and execute the above program as shown below.

```
$javac CreateTable.java

$java CreateTable
```

The above compilation works fine only if you have set the classpath in "**.bashrc**". If you haven't, follow the procedure given below to compile your .java file.

```
//if " /home/home/hadoop/hbase " is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will give you the following output:

```
Table created
```

list

List is the command that is used to list all the tables in HBase. Given below is the syntax of the list command.

```
hbase(main):001:0 > list
```

When you type this command and execute in HBase prompt, it will display the list of all the tables in HBase as shown below.

```
hbase(main):001:0> list

TABLE

emp
```

Here you can observe a table named emp.

Listing Tables Using Java API

Follow the steps given below to get the list of tables from HBase using java API.

Step 1

You have a method called **listTables()** in the class **HBaseAdmin** to get the list of all the tables in HBase. This method returns an array of **HTableDescriptor** objects.

```
//creating a configuration object
Configuration conf = HBaseConfiguration.create();


 //Creating HBaseAdmin object
HBaseAdmin admin = new HBaseAdmin(conf);


//Getting all the list of tables using HBaseAdmin object
HTableDescriptor[] tableDescriptor =admin.listTables();
```

Step 2

You can get the length of the **HTableDescriptor[]** array using the **length** variable of the **HTableDescriptor** class. Get the name of the tables from this object using **getNameAsString()** method. Run the 'for' loop using these and get the list of the tables in HBase.

Given below is the program to list all the tables in HBase using Java API.

```java
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.MasterNotRunningException;
import org.apache.hadoop.hbase.client.HBaseAdmin;

public class ListTables {

   public static void main(String args[])throws
   MasterNotRunningException, IOException{

      // Instantiating a configuration class
      Configuration conf = HBaseConfiguration.create();

      // Instantiating HBaseAdmin class
      HBaseAdmin admin = new HBaseAdmin(conf);

      // Getting all the list of tables using HBaseAdmin object
      HTableDescriptor[] tableDescriptor =admin.listTables();

      // printing all the table names.
      for (int i=0;i< tableDescriptor.length;i++ ){

            System.out.println(tableDescriptor[i].getNameAsString());

      }
```

```
    }

}
```

Compile and execute the above program as shown below.

```
$javac ListTables.java
$java ListTables
```

The above compilation works fine only if you have set the classpath in "**.bashrc**". If you haven't, follow the procedure given below to compile your .java file.

```
//if " /home/home/hadoop/hbase " is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will give you the following output:

```
User
emp
```

Disable a Table

To delete a table or change its settings, you need to first disable the table using the disable command. You can re-enable it using the enable command.

Given below is the syntax to disable a table:

```
disable 'emp'
```

Given below is an example that shows how to disable a table.

```
hbase(main):025:0> disable 'emp'
0 row(s) in 1.2760 seconds
```

Verification

After disabling the table, you can still sense its existence through **list** and **exists** commands. You cannot scan it. It will give you the following error.

```
hbase(main):028:0> scan 'emp'


ROW                  COLUMN+CELL


ERROR: emp is disabled.
```

is_disabled

This command is used to find whether a table is disabled. Its syntax is as follows.

```
hbase> is_disabled 'table name'
```

The following example verifies whether the table named emp is disabled. If it is disabled, it will return true and if not, it will return false.

```
hbase(main):031:0> is_disabled 'emp'

true


0 row(s) in 0.0440 seconds
```

disable_all

This command is used to disable all the tables matching the given regex. The syntax for **disable_all** command is given below.

```
hbase> disable_all 'r.*'
```

Suppose there are 5 tables in HBase, namely raja, rajani, rajendra, rajesh, and raju. The following code will disable all the tables starting with **raj.**

```
hbase(main):002:0> disable_all 'raj.*'
raja
rajani
rajendra

rajesh

raju


Disable the above 5 tables (y/n)?

y
5 tables successfully disabled
```

Disable a Table Using Java API

To verify whether a table is disabled, **isTableDisabled(**) method is used and to disable a table, **disableTable()** method is used. These methods belong to the HBaseAdmin class. Follow the steps given below to disable a table.

Step 1

Instantiate **HBaseAdmin** class as shown below.

```
// Creating configuration object

Configuration conf = HBaseConfiguration.create();

```

```
// Creating HBaseAdmin object
HBaseAdmin admin = new HBaseAdmin(conf);
```

Step 2

Verify whether the table is disabled using **isTableDisabled()** method as shown below.

```
Boolean b=admin.isTableDisabled("emp");
```

Step 3

If the table is not disabled, disable it as shown below.

```
if(!b){

    admin.disableTable("emp");
    System.out.println("Table disabled");

}
```

Given below is the complete program to verify whether the table is disabled; if not, how to disable it.

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.MasterNotRunningException;
import org.apache.hadoop.hbase.client.HBaseAdmin;

public class DisableTable{

    public static void main(String args[]) throws
    MasterNotRunningException, IOException{

        // Instantiating configuration class
```

```
    Configuration conf = HBaseConfiguration.create();


    // Instantiating HBaseAdmin class
    HBaseAdmin admin = new HBaseAdmin(conf);


    // Verifying weather the table is disabled
    Boolean bool = admin.isTableDisabled("emp");
    System.out.println(bool);


    // Disabling the table using HBaseAdmin object
    if(!bool){

        admin.disableTable("emp");
        System.out.println("Table disabled");


    }


  }

}
```

Compile and execute the above program as shown below.

```
$javac DisableTable.java
$java DsiableTable
```

The above compilation works only if you have set the classpath in " **.bashrc** ". If you haven't, follow the procedure given below to compile your .java file.

```
//if "/home/home/hadoop/hbase" is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
false

Table disabled
```

Enable a Table

Syntax to enable a table:

```
enable 'emp'
```

Given below is an example to enable a table.

```
hbase(main):005:0> enable 'emp'
0 row(s) in 0.4580 seconds
```

Verification

After enabling the table, scan it. If you can see the schema, your table is successfully enabled.

```
hbase(main):006:0> scan 'emp'

ROW                        COLUMN+CELL

1    column=personal data:city, timestamp=1417516501, value=hyderabad

1    column=personal data:name, timestamp=1417525058, value=ramu

1    column=professional data:designation, timestamp=1417532601, value=manager

1    column=professional data:salary, timestamp=1417524244109, value=50000

2    column=personal data:city, timestamp=1417524574905, value=chennai

2    column=personal data:name, timestamp=1417524556125, value=ravi

2    column=professional data:designation, timestamp=14175292204, value=sr:engg

2    column=professional data:salary, timestamp=1417524604221, value=30000
```

```
3     column=personal data:city, timestamp=1417524681780, value=delhi

3     column=personal data:name, timestamp=1417524672067, value=rajesh

3     column=professional data:designation, timestamp=14175246987, value=jr:engg

3     column=professional data:salary, timestamp=1417524702514, value=25000

3 row(s) in 0.0400 seconds
```

is_enabled

This command is used to find whether a table is enabled. Its syntax is as follows:

```
hbase> is_enabled 'table name'
```

The following code verifies whether the table named **emp** is enabled. If it is enabled, it will return true and if not, it will return false.

```
hbase(main):031:0> is_enabled 'emp'

true

0 row(s) in 0.0440 seconds
```

Enable a Table Using Java API

To verify whether a table is enabled, **isTableEnabled()** method is used; and to enable a table, **enableTable()** method is used. These methods belong to **HBaseAdmin** class. Follow the steps given below to enable a table.

Step 1

Instantiate **HBaseAdmin** class as shown below.

```
// Creating configuration object
Configuration conf = HBaseConfiguration.create();


// Creating HBaseAdmin object
HBaseAdmin admin = new HBaseAdmin(conf);
```

Step 2

Verify whether the table is enabled using **isTableEnabled()** method as shown below.

```
Boolean bool=admin.isTableEnabled("emp");
```

Step 3

If the table is not disabled, disable it as shown below.

```
if(!bool){


    admin.enableTable("emp");

    System.out.println("Table enabled");


}
```

Given below is the complete program to verify whether the table is enabled and if it is not, then how to enable it.

```
import java.io.IOException;


import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.MasterNotRunningException;

import org.apache.hadoop.hbase.client.HBaseAdmin;


public class EnableTable{

    public static void main(String args[]) throws

    MasterNotRunningException, IOException{


      // Instantiating configuration class

      Configuration conf = HBaseConfiguration.create();


      // Instantiating HBaseAdmin class
```

```
        HBaseAdmin admin = new HBaseAdmin(conf);


        // Verifying weather the table is disabled
        Boolean bool = admin.isTableEnabled("emp");
        System.out.println(bool);


        // Disabling the table using HBaseAdmin object
        if(!bool){


        admin.enableTable("emp");
        System.out.println("Table Enabled");


        }


    }


}
```

Compile and execute the above program as shown below.

```
$javac EnableTable.java
$java EnableTable
```

The above compilation works only if you have set the classpath in " **.bashrc** ". If you haven't, follow the procedure given below to compile your .java file.

```
//if " /home/home/hadoop/hbase " is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
false
```

```
Table Enabled
```

11. DESCRIBE AND ALTER

describe

This command returns the description of the table. Its syntax is as follows:

```
hbase> describe 'table name'
```

Given below is the output of the describe command on the **emp** table.

```
hbase(main):006:0> describe 'emp'
DESCRIPTION
      ENABLED


'emp', {NAME => 'READONLY', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER
=> 'ROW', REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS =>
'1', TTL  true


=> 'FOREVER', MIN_VERSIONS => '0', KEEP_DELETED_CELLS => 'false',
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}, {NAME
=> 'personal


data', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
REPLICATION_SCOPE => '0', VERSIONS => '5', COMPRESSION => 'NONE',
MIN_VERSIONS => '0', TTL


 => 'FOREVER', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536',
IN_MEMORY => 'false', BLOCKCACHE => 'true'}, {NAME => 'professional
data', DATA_BLO


CK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0',
VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL =>
'FOREVER', K


EEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY =>
'false', BLOCKCACHE => 'true'}, {NAME => 'table_att_unset',
DATA_BLOCK_ENCODING => 'NO


NE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', COMPRESSION =>
'NONE', VERSIONS => '1', TTL => 'FOREVER', MIN_VERSIONS => '0',
```

```
KEEP_DELETED_CELLS

=> 'false', BLOCKSIZE => '6
```

alter

Alter is the command used to make changes to an existing table. Using this command, you can change the maximum number of cells of a column family, set and delete table scope operators, and delete a column family from a table.

Changing the Maximum Number of Cells of a Column Family

Given below is the syntax to change the maximum number of cells of a column family.

```
hbase> alter 't1', NAME => 'f1', VERSIONS => 5
```

In the following example, the maximum number of cells is set to 5.

```
hbase(main):003:0> alter 'emp', NAME => 'personal data', VERSIONS => 5

Updating all regions with the new schema...

0/1 regions updated.

1/1 regions updated.

Done.

0 row(s) in 2.3050 seconds
```

Table Scope Operators

Using alter, you can set and remove table scope operators such as MAX_FILESIZE, READONLY, MEMSTORE_FLUSHSIZE, DEFERRED_LOG_FLUSH, etc.

Setting Read Only

Below given is the syntax to make a table read only.

```
hbase>alter 't1', READONLY(option)
```

In the following example, we have made the **emp** table read only.

```
hbase(main):006:0> alter 'emp', READONLY

Updating all regions with the new schema...

0/1 regions updated.
```

tutorialspoint
SIMPLYEASYLEARNING

```
1/1 regions updated.

Done.

0 row(s) in 2.2140 seconds
```

Removing Table Scope Operators

We can also remove the table scope operators. Given below is the syntax to remove 'MAX_FILESIZE' from emp table.

```
hbase> alter 't1', METHOD => 'table_att_unset', NAME => 'MAX_FILESIZE'
```

Deleting a Column Family

Using alter, you can also delete a column family. Given below is the syntax to delete a column family using alter.

```
hbase> alter ' table name ', 'delete' => ' column family '
```

Given below is an example to delete a column family from the 'emp' table.

Assume there is a table named employee in HBase. It contains the following data:

```
hbase(main):006:0> scan 'employee'

 ROW                  COLUMN+CELL

  row1    column=personal:city, timestamp=1418193767, value=hyderabad

  row1    column=personal:name, timestamp=1418193806767, value=raju

  row1     column=professional:designation, timestamp=1418193767, value=manager

  row1      column=professional:salary, timestamp=1418193806767, value=50000

1 row(s) in 0.0160 seconds
```

Now let us delete the column family named **professional** using the alter command.

```
hbase(main):007:0> alter 'employee','delete'=>'professional'

Updating all regions with the new schema...

0/1 regions updated.
```

```
1/1 regions updated.

Done.

0 row(s) in 2.2380 seconds
```

Now verify the data in the table after alteration. Observe the column family 'professional' is no more, since we have deleted it.

```
hbase(main):003:0> scan 'employee'


ROW                    COLUMN+CELL


 row1    column=personal:city, timestamp=14181936767, value=hyderabad


 row1    column=personal:name, timestamp=1418193806767, value=raju


1 row(s) in 0.0830 seconds
```

Adding a Column Family Using Java API

You can add a column family to a table using the method **addColumn()** of **HBAseAdmin** class. Follow the steps given below to add a column family to a table.

Step 1

Instantiate the **HBaseAdmin** class.

```
    // Instantiating configuration object
    Configuration conf = HBaseConfiguration.create();


    // Instantiating HBaseAdmin class
    HBaseAdmin admin = new HBaseAdmin(conf);
```

Step 2

The **addColumn()** method requires a table name and an object of **HColumnDescriptor** class. Therefore instantiate the **HColumnDescriptor** class. The constructor of **HColumnDescriptor** in turn requires a column family

tutorialspoint
SIMPLYEASYLEARNING

name that is to be added. Here we are adding a column family named "contactDetails" to the existing "employee" table.

```
// Instantiating columnDescriptor object

HColumnDescriptor columnDescriptor = new

HColumnDescriptor("contactDetails");
```

Step 3

Add the column family using **addColumn** method. Pass the table name and the **HColumnDescriptor** class object as parameters to this method.

```
// Adding column family

admin.addColumn("employee", new HColumnDescriptor("columnDescriptor"));
```

Given below is the complete program to add a column family to an existing table.

```java
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.MasterNotRunningException;
import org.apache.hadoop.hbase.client.HBaseAdmin;

public class AddColoumn{

   public static void main(String args[]) throws
   MasterNotRunningException, IOException{

     // Instantiating configuration class.
     Configuration conf = HBaseConfiguration.create();


     // Instantiating HBaseAdmin class.
     HBaseAdmin admin = new HBaseAdmin(conf);
```

```
     // Instantiating columnDescriptor class
     HColumnDescriptor columnDescriptor =new
     HColumnDescriptor("contactDetails");


     // Adding column family
     admin.addColumn("employee", columnDescriptor);
     System.out.println("coloumn added");


   }


 }
```

Compile and execute the above program as shown below.

```
$javac AddColumn.java
$java AddColumn
```

The above compilation works only if you have set the classpath in " **.bashrc** ". If you haven't, follow the procedure given below to compile your .java file.

```
//if "/home/home/hadoop/hbase " is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
column added
```

Deleting a Column Family Using Java API

You can delete a column family from a table using the method **deleteColumn()** of **HBAseAdmin** class. Follow the steps given below to add a column family to a table.

Step 1

Instantiate the **HBaseAdmin** class.

```
  // Instantiating configuration object
  Configuration conf = HBaseConfiguration.create();
```

```
    // Instantiating HBaseAdmin class

    HBaseAdmin admin = new HBaseAdmin(conf);
```

Step 2

Add the column family using **deleteColumn()** method. Pass the table name and the column family name as parameters to this method.

```
// Deleting column family

admin.deleteColumn("employee", "contactDetails");
```

Given below is the complete program to delete a column family from an existing table.

```
import java.io.IOException;


import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.MasterNotRunningException;

import org.apache.hadoop.hbase.client.HBaseAdmin;


public class DeleteColoumn{

    public static void main(String args[]) throws

    MasterNotRunningException, IOException{


      // Instantiating configuration class.
      Configuration conf = HBaseConfiguration.create();


      // Instantiating HBaseAdmin class.
      HBaseAdmin admin = new HBaseAdmin(conf);


      // Deleting a column family
      admin.deleteColumn("employee","contactDetails");
      System.out.println("coloumn deleted");
```

```
    }

}
```

Compile and execute the above program as shown below.

```
$javac DeleteColumn.java

$java DeleteColumn
```

The above compilation works only if you have set the class path in " **.bashrc** ". If you haven't, follow the procedure given below to compile your .java file.

```
//if "/home/home/hadoop/hbase" is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
column deleted
```

exists

You can verify the existence of a table using the **exists** command. The following example shows how to use this command.

```
hbase(main):024:0> exists 'emp'

Table emp does exist


0 row(s) in 0.0750 seconds



========================================================================



hbase(main):015:0> exists 'student'

Table student does not exist


0 row(s) in 0.0480 seconds
```

Verifying the Existence of Table Using Java API

You can verify the existence of a table in HBase using the **tableExists()** method of the **HBaseAdmin** class. Follow the steps given below to verify the existence of a table in HBase.

Step 1

```
    Instantiate the HBaseAdimn class
    // Instantiating configuration object
    Configuration conf = HBaseConfiguration.create();


    // Instantiating HBaseAdmin class
    HBaseAdmin admin = new HBaseAdmin(conf);
```

Step 2

Verify the existence of the table using the **tableExists( )** method.

Given below is the java program to test the existence of a table in HBase using java API.

```
import java.io.IOException;

import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.client.HBaseAdmin;

public class TableExists{

   public static void main(String args[])throws IOException{

     // Instantiating configuration class
     Configuration conf = HBaseConfiguration.create();

     // Instantiating HBaseAdmin class
     HBaseAdmin admin = new HBaseAdmin(conf);

     // Verifying the existance of the table
     boolean bool = admin.tableExists("emp");
     System.out.println( bool);

   }

}
```

Compile and execute the above program as shown below.

```
$javac TableExists.java
$java TableExists
```

The above compilation works only if you have set the classpath in " **.bashrc** ". If you haven't, follow the procedure given below to compile your .java file.

```
//if "/home/home/hadoop/hbase" is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
true
```

drop

Using the drop command, you can delete a table. Before dropping a table, you have to disable it.

```
hbase(main):018:0> disable 'emp'

0 row(s) in 1.4580 seconds


hbase(main):019:0> drop 'emp'

0 row(s) in 0.3060 seconds
```

Verify whether the table is deleted using the exists command.

```
hbase(main):020:0> exists 'emp'

Table emp does not exist


0 row(s) in 0.0730 seconds
```

drop_all

This command is used to drop the tables matching the "regex" given in the command. Its syntax is as follows:

```
hbase> drop_all 't.*'
```

Note: Before dropping a table, you must disable it.

Example

Assume there are tables named raja, rajani, rajendra, rajesh, and raju.

```
hbase(main):017:0> list

TABLE

raja

rajani

rajendra
```

```
rajesh

raju

9 row(s) in 0.0270 seconds
```

All these tables start with the letters **raj**. First of all, let us disable all these tables using the **disable_all** command as shown below.

```
hbase(main):002:0> disable_all 'raj.*'

raja

rajani

rajendra

rajesh

raju


Disable the above 5 tables (y/n)?

y

5 tables successfully disabled
```

Now you can delete all of them using the **drop_all** command as given below.

```
hbase(main):018:0> drop_all 'raj.*'

raja

rajani

rajendra

rajesh

raju


Drop the above 5 tables (y/n)?

y

5 tables successfully dropped
```

Deleting a Table Using Java API

You can delete a table using the **deleteTable()** method in the **HBaseAdmin** class. Follow the steps given below to delete a table using java API.

Step 1

Instantiate the HBaseAdmin class.

```
// creating a configuration object
Configuration conf = HBaseConfiguration.create();


// Creating HBaseAdmin object
HBaseAdmin admin = new HBaseAdmin(conf);
```

Step 2

Disable the table using the **disableTable()** method of the **HBaseAdmin** class**.**

```
admin.disableTable("emp1");
```

Step 3

Now delete the table using the **deleteTable()** method of the **HBaseAdmin** class.

```
admin.deleteTable("emp12");
```

Given below is the complete java program to delete a table in HBase.

```
import java.io.IOException;

import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.client.HBaseAdmin;

public class DeleteTable {

    public static void main(String[] args) throws IOException {

        // Instantiating configuration class
        Configuration conf = HBaseConfiguration.create();
```

```
      // Instantiating HBaseAdmin class

      HBaseAdmin admin = new HBaseAdmin(conf);


      // disabling table named emp

      admin.disableTable("emp12");


      // Deleting emp

      admin.deleteTable("emp12");

      System.out.println("Table deleted");


   }


}
```

Compile and execute the above program as shown below.

```
$javac DeleteTable.java

$java DeleteTable
```

The above compilation works only if you have set the classpath in "**.bashrc**". If you haven't, follow the procedure given below to compile your .java file.

```
//if "/home/home/hadoop/hbase" is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
Table deleted
```

14. SHUTTING DOWN HBASE

exit

You exit the shell by typing the **exit** command.

```
hbase(main):021:0> exit
```

Stopping HBase

To stop HBase, browse to the HBase home folder and type the following command.

```
./bin/stop-hbase.sh
```

Stopping HBase Using Java API

You can shut down the HBase using the **shutdown()** method of the **HBaseAdmin** class. Follow the steps given below to shut down HBase:

Step 1

Instantiate the HbaseAdmin class.

```
// Instantiating configuration object
Configuration conf = HBaseConfiguration.create();


// Instantiating HBaseAdmin object
HBaseAdmin admin = new HBaseAdmin(conf);
```

Step 2

Shut down the HBase using the **shutdown()** method of the **HBaseAdmin** class.

```
admin.shutdown();
```

Given below is the program to stop the HBase.

```
import java.io.IOException;


import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.client.HBaseAdmin;


public class ShutDownHbase{


    public static void main(String args[])throws IOException {


        // Instantiating configuration class
        Configuration conf = HBaseConfiguration.create();


        // Instantiating HBaseAdmin class
        HBaseAdmin admin = new HBaseAdmin(conf);


        // Shutting down HBase
        System.out.println("Shutting down  hbase");
        admin.shutdown();


    }


}
```

Compile and execute the above program as shown below.

```
$javac ShutDownHbase.java
$java ShutDownHbase
```

The above compilation works only if you have set the classpath in "**.bashrc**". If you haven't, follow the procedure given below to compile your .java file.

```
// if "/home/home/hadoop/hbase" is your Hbase home folder then.

$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
Shutting down hbase
```

This chapter describes the java client API for HBase that is used to perform **CRUD** operations on HBase tables. HBase is written in Java and has a Java Native API. Therefore it provides programmatic access to Data Manipulation Language (DML).

Class HBaseConfiguration

Adds HBase configuration files to a Configuration. This class belongs to the **org.apache.hadoop.hbase** package.

Methods and description

S. No.	Methods and Description
1	**static org.apache.hadoop.conf.Configuration create()** This method creates a Configuration with HBase resources.

Class HTable

HTable is an HBase internal class that represents an HBase table. It is an implementation of table that is used to communicate with a single HBase table. This class belongs to the **org.apache.hadoop.hbase.client** class.

Constructors

S. No.	Constructors and Description
1	**HTable()**
2	**HTable(TableName tableName, ClusterConnection connection, ExecutorService pool)** Using this constructor, you can create an object to access an HBase table.

Methods and description

S. No.	Methods and Description
1	**void close()** Releases all the resources of the HTable.
2	**void delete(Delete delete)** Deletes the specified cells/row.
3	**boolean exists(Get get)** Using this method, you can test the existence of columns in the table, as specified by Get.
4	**Result get(Get get)** Retrieves certain cells from a given row.
5	**org.apache.hadoop.conf.Configuration getConfiguration()** Returns the Configuration object used by this instance.
6	**TableName getName()** Returns the table name instance of this table**.**
7	**HTableDescriptor getTableDescriptor()** Returns the table descriptor for this table.
8	**byte[] getTableName()** Returns the name of this table.
9	**void put(Put put)** Using this method, you can insert data into the table.

Class Put

This class is used to perform Put operations for a single row. It belongs to the **org.apache.hadoop.hbase.client** package.

Constructor

S. No.	Constructors and Description
1	**Put(byte[] row)** Using this constructor, you can create a Put operation for the specified row.
2	**Put(byte[] rowArray, int rowOffset, int rowLength)** Using this constructor, you can make a copy of the passed-in row key to keep local.
3	**Put(byte[] rowArray, int rowOffset, int rowLength, long ts)** Using this constructor, you can make a copy of the passed-in row key to keep local.
4	**Put(byte[] row, long ts)** Using this constructor, we can create a Put operation for the specified row, using a given timestamp.

Methods

S. No.	Methods and Description
1	**Put add(byte[] family, byte[] qualifier, byte[] value)** Adds the specified column and value to this Put operation.
2	**Put add(byte[] family, byte[] qualifier, long ts, byte[] value)** Adds the specified column and value, with the specified timestamp as its version to this Put operation.
3	**Put add(byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value)** Adds the specified column and value, with the specified timestamp as its version to this Put operation.
4	**Put add(byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value)** Adds the specified column and value, with the specified timestamp as its version to this Put operation.

Class Get

This class is used to perform Get operations on a single row. This class belongs to the **org.apache.hadoop.hbase.client** package.

Constructor

S. No.	Constructor and Description
1	**Get(byte[] row)** Using this constructor, you can create a Get operation for the specified row.
2	**Get(Get get)**

Methods

S. No.	Methods and Description
1	**Get addColumn(byte[] family, byte[] qualifier)** Retrieves the column from the specific family with the specified qualifier.
2	**Get addFamily(byte[] family)** Retrieves all columns from the specified family.

Class Delete

This class is used to perform Delete operations on a single row. To delete an entire row, instantiate a Delete object with the row to delete. This class belongs to the **org.apache.hadoop.hbase.client** package.

Constructor

S. No.	Constructor and Description
1	**Delete(byte[] row)**

	Creates a Delete operation for the specified row.
2	**Delete(byte[] rowArray, int rowOffset, int rowLength)** Creates a Delete operation for the specified row and timestamp.
3	**Delete(byte[] rowArray, int rowOffset, int rowLength, long ts)** Creates a Delete operation for the specified row and timestamp.
4	**Delete(byte[] row, long timestamp)** Creates a Delete operation for the specified row and timestamp.

Methods

S. No.	Methods and Description
1	**Delete addColumn(byte[] family, byte[] qualifier)** Deletes the latest version of the specified column.
2	**Delete addColumns(byte[] family, byte[] qualifier, long timestamp)** Deletes all versions of the specified column with a timestamp less than or equal to the specified timestamp.
3	**Delete addFamily(byte[] family)** Deletes all versions of all columns of the specified family.
4	**Delete addFamily(byte[] family, long timestamp)** Deletes all columns of the specified family with a timestamp less than or equal to the specified timestamp.

Class Result

This class is used to get a single row result of a Get or a Scan query.

Constructors

S. No.	Constructors
1	**Result()** Using this constructor, you can create an empty Result with no KeyValue payload; returns null if you call raw Cells().

Methods

S. No.	Methods and Description
1	**byte[] getValue(byte[] family, byte[] qualifier)** This method is used to get the latest version of the specified column.
2	**byte[] getRow()** This method is used to retrieve the row key that corresponds to the row from which this Result was created.

Creating Data

This chapter demonstrates how to create data in an HBase table. To create data in an HBase table, the following commands and methods are used:

- **put** command,
- **add()** method of **Put** class**,** and
- **put()** method of **HTable** class.

As an example, we are going to create the following table in HBase.

Using **put** command, you can insert rows into a table. Its syntax is as follows:

```
put '<table name>','row1','<colfamily:colname>','<value>'
```

Inserting the First Row

Let us insert the first row values into the emp table as shown below.

```
hbase(main):005:0> put 'emp','1','personal data:name','raju'
0 row(s) in 0.6600 seconds


hbase(main):006:0> put 'emp','1','personal data:city','hyderabad'
0 row(s) in 0.0410 seconds
```

```
hbase(main):007:0> put 'emp','1','professional
data:designation','manager'

0 row(s) in 0.0240 seconds


hbase(main):007:0> put 'emp','1','professional data:salary','50000'

0 row(s) in 0.0240 seconds
```

Insert the remaining rows using the put command in the same way. If you insert the whole table, you will get the following output.

```
hbase(main):022:0> scan 'emp'
ROW                              COLUMN+CELL

 1    column=personal data:city, timestamp=1417524216501, value=hyderabad

 1    column=personal data:name, timestamp=1417524185058, value=ramu

 1    column=professional data:designation, timestamp=1417524232601,
      value=manager

 1    column=professional data:salary, timestamp=1417524244109, value=50000

 2    column=personal data:city, timestamp=1417524574905, value=chennai

 2    column=personal data:name, timestamp=1417524556125, value=ravi

 2    column=professional data:designation, timestamp=1417524592204,
      value=sr:engg

 2    column=professional data:salary, timestamp=1417524604221, value=30000

 3    column=personal data:city, timestamp=1417524681780, value=delhi

 3    column=personal data:name, timestamp=1417524672067, value=rajesh

 3    column=professional data:designation, timestamp=1417524693187,
      value=jr:engg

 3    column=professional data:salary, timestamp=1417524702514,
```

```
    value=25000
```

Inserting Data Using Java API

You can insert data into Hbase using the **add()** method of the **Put** class. You can save it using the **put()** method of the **HTable** class. These classes belong to the **org.apache.hadoop.hbase.client** package. Below given are the steps to create data in a Table of HBase.

Step 1: Instantiate the Configuration Class

The **Configuration** class adds HBase configuration files to its object. You can create a configuration object using the **create()** method of the **HbaseConfiguration** class as shown below.

```
Configuration conf = HbaseConfiguration.create();
```

Step 2: Instantiate the HTable Class

You have a class called **HTable**, an implementation of Table in HBase. This class is used to communicate with a single HBase table. While instantiating this class, it accepts configuration object and table name as parameters. You can instantiate HTable class as shown below.

```
HTable hTable = new HTable(conf, tableName);
```

Step 3: Instantiate the Put Class

To insert data into an HBase table, the **add()** method and its variants are used. This method belongs to **Put**, therefore instantiate the put class. This class requires the row name you want to insert the data into, in string format. You can instantiate the **Put** class as shown below.

```
Put p = new Put(Bytes.toBytes("row1"));
```

Step 4: Insert Data

The **add()** method of **Put** class is used to insert data. It requires 3 byte arrays representing column family, column qualifier (column name), and the value to be inserted, respectively. Insert data into the HBase table using the add() method as shown below.

```
p.add(Bytes.toBytes("coloumn family "), Bytes.toBytes("column

name"),Bytes.toBytes("value"));
```

Step 5: Save the Data in Table

After inserting the required rows, save the changes by adding the put instance to the **put()** method of HTable class as shown below.

```
hTable.put(p);
```

Step 6: Close the HTable Instance

After creating data in the HBase Table, close the **HTable** instance using the close() method as shown below.

```
hTable.close();
```

Given below is the complete program to create data in HBase Table.

```java
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;

public class InsertData{

   public static void main(String[] args) throws IOException {

      // Instantiating Configuration class
      Configuration config = HBaseConfiguration.create();

      // Instantiating HTable class
      HTable hTable = new HTable(config, "emp");

      // Instantiating Put class
      // accepts a row name.
      Put p = new Put(Bytes.toBytes("row1"));
```

```
     // adding values using add() method
     // accepts column family name, qualifier/row name ,value
     p.add(Bytes.toBytes("personal"),
     Bytes.toBytes("name"),Bytes.toBytes("raju"));


     p.add(Bytes.toBytes("personal"),
     Bytes.toBytes("city"),Bytes.toBytes("hyderabad"));


     p.add(Bytes.toBytes("professional"),Bytes.toBytes("designation"),
     Bytes.toBytes("manager"));


     p.add(Bytes.toBytes("professional"),Bytes.toBytes("salary"),
     Bytes.toBytes("50000"));


     // Saving the put Instance to the HTable.
     hTable.put(p);
     System.out.println("data inserted");


     // closing HTable
     hTable.close();


   }
}
```

Compile and execute the above program as shown below.

```
$javac InsertData.java
$java InsertData
```

The above compilation works only if you have set the classpath in " **.bashrc** ". If you haven't, follow the procedure given below to compile your .java file.

```
//if "/home/home/hadoop/hbase" is your Hbase home folder then.
$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
data inserted
```

Updating Data

You can update an existing cell value using the **put** command. To do so, just follow the same syntax and mention your new value as shown below.

```
put 'table name','row ','Column family:column name','new value'
```

The newly given value replaces the existing value, updating the row.

Example

Suppose there is a table in HBase called **emp** with the following data.

```
hbase(main):003:0> scan 'emp'


ROW                   COLUMN+CELL



 row1   column=personal:name, timestamp=1418051555,   value=raju


 row1   column=personal:city, timestamp=1418275907, value=Hyderabad


 row1   column=professional:designation, timestamp=14180555,value=manager


 row1   column=professional:salary, timestamp=1418035791555,value=50000

1 row(s) in 0.0100 seconds
```

The following command will update the city value of the employee named 'Raju' to Delhi.

```
hbase(main):002:0> put 'emp','row1','personal:city','Delhi'

0 row(s) in 0.0400 seconds
```

The updated table looks as follows where you can observe the city of Raju has been changed to 'Delhi'.

```
hbase(main):003:0> scan 'emp'


 ROW              COLUMN+CELL


row1    column=personal:name,  timestamp=1418035791555,  value=raju


row1    column=personal:city,  timestamp=1418274645907,  value=Delhi


row1    column=professional:designation, timestamp=141857555,value=manager


row1    column=professional:salary,  timestamp=1418039555, value=50000


1 row(s) in 0.0100 seconds
```

Updating Data Using Java API

You can update the data in a particular cell using the **put()** method. Follow the steps given below to update an existing cell value of a table.

Step 1: Instantiate the Configuration Class

Configuration class adds HBase configuration files to its object. You can create a configuration object using the **create()** method of the **HbaseConfiguration** class as shown below.

```
Configuration conf = HbaseConfiguration.create();
```

Step 2: Instantiate the HTable Class

You have a class called **HTable**, an implementation of Table in HBase. This class is used to communicate with a single HBase table. While instantiating this class, it accepts the configuration object and the table name as parameters. You can instantiate the HTable class as shown below.

```
HTable hTable = new HTable(conf, tableName);
```

Step 3: Instantiate the Put Class

To insert data into HBase Table, the **add()** method and its variants are used. This method belongs to **Put**, therefore instantiate the put class. This class requires the row name you want to insert the data into, in string format. You can instantiate the **Put** class as shown below.

```
Put p = new Put(Bytes.toBytes("row1"));
```

Step 4: Update an Existing Cell

The **add()** method of **Put** class is used to insert data. It requires 3 byte arrays representing column family, column qualifier (column name), and the value to be inserted, respectively. Insert data into HBase table using the add() method as shown below.

```
p.add(Bytes.toBytes("coloumn family "), Bytes.toBytes("column
name"),Bytes.toBytes("value"));


p.add(Bytes.toBytes("personal"),
Bytes.toBytes("city"),Bytes.toBytes("Delih"));
```

Step 5: Save the Data in Table

After inserting the required rows, save the changes by adding the put instance to the **put()** method of the HTable class as shown below.

```
hTable.put(p);
```

Step 6: Close HTable Instance

After creating data in HBase Table, close the **HTable** instance using the close() method as shown below.

```
hTable.close();
```

Given below is the complete program to update data in a particular table.

```
import java.io.IOException;


import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.HTable;
```

```
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;


public class UpdateData{

   public static void main(String[] args) throws IOException {

     // Instantiating Configuration class
     Configuration config = HBaseConfiguration.create();


     // Instantiating HTable class
     HTable hTable = new HTable(config, "emp");


     // Instantiating Put class
     //accepts a row name
     Put p = new Put(Bytes.toBytes("row1"));


     // Updating a cell value
     p.add(Bytes.toBytes("personal"),
     Bytes.toBytes("city"),Bytes.toBytes("Delih"));


     // Saving the put Instance to the HTable.
     hTable.put(p);
     System.out.println("data Updated");


     // closing HTable
     hTable.close();


     }


}
```

Compile and execute the above program as shown below.

```
$javac UpdateData.java

$java UpdateData
```

The above compilation works only if you have set the classpath in "**.bashrc**". If you haven't, follow the procedure given below to compile your .java file.

```
//if "/home/home/hadoop/hbase" is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
data Updated
```

Reading Data

The **get** command and the **get()** method of **HTable** class are used to read data from a table in HBase. Using **get** command, you can get a single row of data at a time. Its syntax is as follows:

```
get '<table name>','row1'
```

The following example shows how to use the get command. Let us scan the first row of the **emp** table.

```
hbase(main):012:0> get 'emp', '1'

COLUMN                                    CELL

personal :   city          timestamp=1417521848375,    value=hyderabad

personal :   name          timestamp=1417521785385,    value=ramu

professional: designation  timestamp=1417521885277,    value=manager

professional: salary        timestamp=1417521903862,    value=50000

4 row(s) in 0.0270 seconds
```

Reading a Specific Column

Given below is the syntax to read a specific column using the **get** method.

```
hbase>get 'table name', 'rowid', {COLUMN => 'column family:column name '}
```

Given below is the example to read a specific column in HBase table.

```
hbase(main):015:0> get 'emp', 'row1', {COLUMN=>'personal:name'}


 COLUMN                    CELL


 personal:name    timestamp=1418035791555,    value=raju


1 row(s) in 0.0080 seconds
```

Reading Data Using Java API

To read data from an HBase table, use the **get()** method of the HTable class. This method requires an instance of the **Get** class. Follow the steps given below to retrieve data from the HBase table.

Step 1: Instantiate the Configuration Class

Configuration class adds HBase configuration files to its object. You can create a configuration object using the **create()** method of the **HbaseConfiguration** class as shown below.

```
Configuration conf = HbaseConfiguration.create();
```

Step 2: Instantiate the HTable Class

You have a class called **HTable**, an implementation of Table in HBase. This class is used to communicate with a single HBase table. While instantiating this class, it accepts the configuration object and the table name as parameters. You can instantiate the HTable class as shown below.

```
HTable hTable = new HTable(conf, tableName);
```

Step 3: Instantiate the Get Class

You can retrieve data from the HBase table using the **get()** method of the **HTable** class. This method extracts a cell from a given row. It requires a **Get** class object as parameter. Create it as shown below.

```
Get get = new Get(toBytes("row1"));
```

Step 4: Read the Data

While retrieving data, you can get a single row by id, or get a set of rows by a set of row ids, or scan an entire table or a subset of rows.

You can retrieve an HBase table data using the add method variants in **Get** class.

To get a specific column from a specific column family, use the following method.

```
get.addFamily(personal)
```

To get all the columns from a specific column family, use the following method.

```
get.addColumn(personal, name)
```

Step 5: Get the Result

Get the result by passing your **Get** class instance to the get method of the **HTable** class. This method returns the **Result** class object, which holds the requested result. Given below is the usage of **get()** method.

```
Result result = table.get(g);
```

Step 6: Reading Values from the Result Instance

The **Result** class provides the **getValue()** method to read the values from its instance. Use it as shown below to read the values from the **Result** instance.

```
byte [] value =
result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("name"));

byte [] value1 =
result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("city"));
```

Given below is the complete program to read values from an HBase table.

```
import java.io.IOException;


import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.Get;

import org.apache.hadoop.hbase.client.HTable;

import org.apache.hadoop.hbase.client.Result;
```

```
import org.apache.hadoop.hbase.util.Bytes;

public class RetriveData{

   public static void main(String[] args) throws IOException, Exception
   {
      // Instantiating Configuration class
      Configuration config = HBaseConfiguration.create();

      // Instantiating HTable class
      HTable table = new HTable(config, "emp");

      // Instantiating Get class
      Get g = new Get(Bytes.toBytes("row1"));

      // Reading the data
      Result result = table.get(g);

      // Reading values from Result class object
      byte [] value =
      result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("name"));

      byte [] value1 =
      result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("city"));

      // Printing the values
      String name = Bytes.toString(value);
      String city = Bytes.toString(value1);
      System.out.println("name: "+ name + "   city: "+city);

   }

}
```

Compile and execute the above program as shown below.

```
$javac RetriveData.java

$java RetriveData
```

The above compilation works only if you have set the class path in " **.bashrc** ". If you haven't, follow the procedure given below to compile your .java file.

```
//if "/home/home/hadoop/hbase" is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
name: Raju     city: Delhi
```

19. DELETE DATA

Deleting a Specific Cell in a Table

Using the **delete** command, you can delete a specific cell in a table. The syntax of **delete** command is as follows:

```
delete '<table name>', '<row>', '<column name >', '<time stamp>'
```

Here is an example to delete a specific cell. Here we are deleting the salary.

```
hbase(main):006:0> delete 'emp', '1', 'personal data:city',
1417521848375

0 row(s) in 0.0060 seconds
```

Deleting All Cells in a Table

Using the "deleteall" command, you can delete all the cells in a row. Given below is the syntax of deleteall command.

```
deleteall '<table name>','<row>'
```

Here is an example of "deleteall" command, where we are deleting all the cells of row1 of emp table.

```
hbase(main):007:0> deleteall 'emp','1'
0 row(s) in 0.0240 seconds
```

Verify the table using the **scan** command. A snapshot of the table after deleting the table is given below.

```
hbase(main):022:0> scan 'emp'


ROW                    COLUMN+CELL


 2     column=personal data:city, timestamp=1417524574905, value=chennai

```

```
2    column=personal data:name, timestamp=1417524556125, value=ravi

2    column=professional data:designation, timestamp=1417524204, value=sr:engg

2    column=professional data:salary, timestamp=1417524604221, value=30000

3    column=personal data:city, timestamp=1417524681780, value=delhi

3    column=personal data:name, timestamp=1417524672067, value=rajesh

3    column=professional data:designation, timestamp=1417523187, value=jr:engg

3    column=professional data:salary, timestamp=1417524702514, value=25000
```

Deleting Data Using Java API

You can delete data from an HBase table using the **delete()** method of the **HTable** class. Follow the steps given below to delete data from a table.

Step 1: Instantiate the Configuration Class

Configuration class adds HBase configuration files to its object. You can create a configuration object using the **create()** method of the the **HbaseConfiguration** class as shown below.

```
Configuration conf = HbaseConfiguration.create();
```

Step 2: Instantiate the HTable Class

You have a class called **HTable**, an implementation of Table in HBase. This class is used to communicate with a single HBase table. While instantiating this class, it accepts the configuration object and the table name as parameters. You can instantiate the HTable class as shown below.

```
HTable hTable = new HTable(conf, tableName);
```

Step 3: Instantiate the Delete Class

Instantiate the **Delete** class by passing the rowid of the row that is to be deleted, in byte array format. You can also pass timestamp and Rowlock to this constructor.

```
Delete delete = new Delete(toBytes("row1"));
```

Step 4: Select the Data to be Deleted

You can delete the data using the delete methods of the **Delete** class. This class has various delete methods. Choose the columns or column families to be deleted using those methods. Take a look at the following examples that show the usage of Delete class methods.

```
delete.deleteColumn(Bytes.toBytes("personal"), Bytes.toBytes("name"));

delete.deleteFamily(Bytes.toBytes("professional"));
```

Step 5: Delete the Data

Delete the selected data by passing the **delete** instance to the **delete()** method of the **HTable** class as shown below.

```
table.delete(delete);
```

Step 6: Close the HTable Instance

After deleting the data, close the **HTable** Instance.

```
table.close();
```

Given below is the complete program to delete data from the HBase table.

```java
import java.io.IOException;


import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.Delete;

import org.apache.hadoop.hbase.client.HTable;

import org.apache.hadoop.hbase.util.Bytes;


public class DeleteData {


    public static void main(String[] args) throws IOException {

```

```
        // Instantiating Configuration class
        Configuration conf = HBaseConfiguration.create();


        // Instantiating HTable class
        HTable table = new HTable(conf, "employee");


        // Instantiating Delete class
        Delete delete = new Delete(Bytes.toBytes("row1"));
        delete.deleteColumn(Bytes.toBytes("personal"), Bytes.toBytes("name"));
        delete.deleteFamily(Bytes.toBytes("professional"));


        // deleting the data
        table.delete(delete);


        // closing the HTable object
        table.close();
        System.out.println("data deleted.....");


    }


}
```

Compile and execute the above program as shown below.

```
$javac Deletedata.java
$java DeleteData
```

The above compilation works only if you have set the classpath in " **.bashrc** ". If you haven't, follow the procedure given below to compile your .java file.

```
//if "/home/home/hadoop/hbase " is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
data deleted
```

20. HBASE SCAN

scan

The **scan** command is used to view the data in HTable. Using the scan command, you can get the table data. Its syntax is as follows:

```
scan '<table name>'
```

The following example shows how to read data from a table using the scan command. Here we are reading the **emp** table.

```
hbase(main):010:0> scan 'emp'

ROW                                    COLUMN+CELL

 1    column=personal data:city, timestamp=1417521848375, value=hyderabad

 1    column=personal data:name, timestamp=1417521785385, value=ramu

 1    column=professional data:designation, timestamp=1417585277,value=manager

 1    column=professional data:salary, timestamp=1417521903862, value=50000

1 row(s) in 0.0370 seconds
```

Scanning Using Java API

The complete program to scan the entire table data using java API is as follows.

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
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;


public class ScanTable{

   public static void main(String args[]) throws IOException{


     // Instantiating Configuration class
     Configuration config = HBaseConfiguration.create();


     // Instantiating HTable class
     HTable table = new HTable(config, "emp");


     // Instantiating the Scan class
     Scan scan = new Scan();


     // Scanning the required columns
     scan.addColumn(Bytes.toBytes("personal"), Bytes.toBytes("name"));


     scan.addColumn(Bytes.toBytes("personal"), Bytes.toBytes("city"));


     // Getting the scan result
     ResultScanner scanner = table.getScanner(scan);


     // Reading values from scan result
     for (Result result = scanner.next(); result != null; result =
     Scanner.next())


         System.out.println("Found row : " + result);


     //closing the scanner
     scanner.close();
```

```
    }


}
```

Compile and execute the above program as shown below.

```
$javac ScanTable.java

$java ScanTable
```

The above compilation works only if you have set the classpath in " **.bashrc** ". If you haven't, follow the procedure given below to compile your .java file.

```
//if "/home/home/hadoop/hbase" is your Hbase home folder then.


$javac -cp /home/hadoop/hbase/lib/*: Demo.java
```

If everything goes well, it will produce the following output:

```
Found row :
keyvalues={row1/personal:city/1418275612888/Put/vlen=5/mvcc=0,
row1/personal:name/1418035791555/Put/vlen=4/mvcc=0}
```

21. COUNT AND TRUNCATE

count

You can count the number of rows of a table using the **count** command. Its syntax is as follows:

```
count '<table name>'
```

After deleting the first row, emp table will have two rows. Verify it as shown below.

```
hbase(main):023:0> count 'emp'
2 row(s) in 0.090 seconds
=> 2
```

truncate

This command disables drops and recreates a table. The syntax of **truncate** is as follows:

```
hbase> truncate 'table name'
```

Given below is the example of truncate command. Here we have truncated the **emp** table.

```
hbase(main):011:0> truncate 'emp'
Truncating 'one' table (it may take a while):
  - Disabling table...
  - Truncating table...
0 row(s) in 1.5950 seconds
```

After truncating the table, use the scan command to verify. You will get a table with zero rows.

```
hbase(main):017:0> scan 'emp''
ROW                           COLUMN+CELL
0 row(s) in 0.3110 seconds
```

22. HBASE SECURITY

We can grant and revoke permissions to users in HBase. There are three commands for security purpose: grant, revoke, and user_permission.

grant

The **grant** command grants specific rights such as read, write, execute, and admin on a table to a certain user. The syntax of grant command is as follows:

```
hbase> grant <user> <permissions> [<table> [<column family> [<column qualifier>]]
```

We can grant zero or more privileges to a user from the set of RWXCA, where

- R - represents read privilege.

- W - represents write privilege.

- X - represents execute privilege.

- C - represents create privilege.

- A - represents admin privilege.

Given below is an example that grants all privileges to a user named 'Tutorialspoint'.

```
hbase(main):018:0> grant 'Tutorialspoint', 'RWXCA'
```

revoke

The **revoke** command is used to revoke a user's access rights of a table. Its syntax is as follows:

```
hbase> revoke <user>
```

The following code revokes all the permissions from the user named 'Tutorialspoint'.

```
hbase(main):006:0> revoke 'Tutorialspoint'
```

user_permission

This command is used to list all the permissions for a particular table. The syntax of **user_permission** is as follows:

```
hbase>user_permission 'tablename'
```

The following code lists all the user permissions of 'emp' table.

```
hbase(main):013:0> user_permission 'emp'
```