What is HBase?

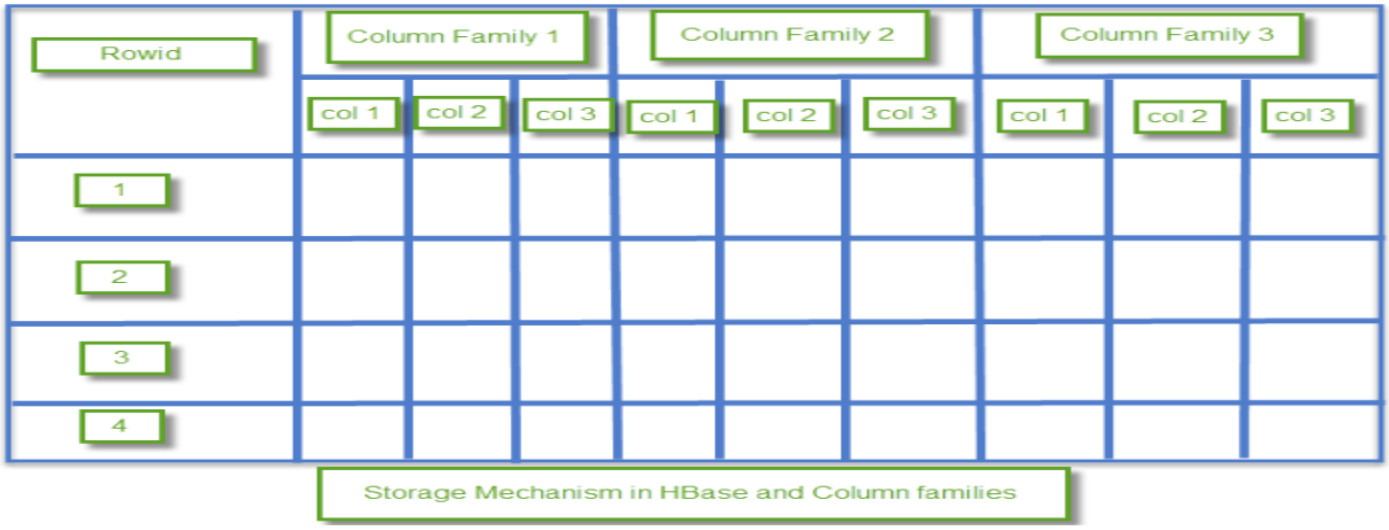
- HBase is an open-source, column-oriented distributed database system in Hadoop environment and is horizontally scalable.
- Initially it was Google Big Table, afterwards it was re-named as HBase and is primarily written in Java.
- It provides random real-time read/write access to data in Hadoop File System.
- HBase can store massive amount of data from terabytes to petabytes.
- HBase is built for low latency operations.

HBase	HDFS
HBase is a database built on top of HDFS	HDFS is a distributed file system suitable for storing large files
Low latency operations	High latency operations
Random reads and writes	Write Once Read Many times
Accessed through shell commands, client API in Java etc.	Primarily accessed through MapReduce jobs
Storage and Process both can be performed	It's only for Storage

Storage Mechanism in HBase:

HBase is a Column-Oriented database and data is stored in tables. The tables are sorted by **RowId**. As shown below HBase has RowId, which is a collection of several column families that are present in the table.

The column families that are present in the schema are key-value pairs. If we observe in detail each column family having multiple numbers of columns. Column values are stored into disk memory. Each cell of the table has its own Metadata like timestamp and other information.



Storage Mechanism in HBase and Column families

Coming to HBase the following are the key terms representing table schema:

- **Table:** Table is a collection of rows.

- **Row:** Row is a collection of column families.
- **Column Family:** Column Family is a collection of columns.
- **Column:** Column is a collection of Key Value pairs.
- **Namespace:** Logical grouping of tables.
- **Cell:** A {row, column, version} tuple exactly specifies cell definition in HBase.

Column Oriented and Row Oriented:

We all know traditional relational models store data in terms of row-based format like in terms of rows of data. Column-oriented storage store data tables in terms of columns and column families.

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

HBase Vs RDBMS:

HBase	RDBMS
HBase is schema-less, it doesn't have the concept of fixed column schema	RDBMS is governed by its schema, which describe the whole structure of tables
It is built for wide tables and 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 structured, semi-structured and unstructured data	It is good for structured data

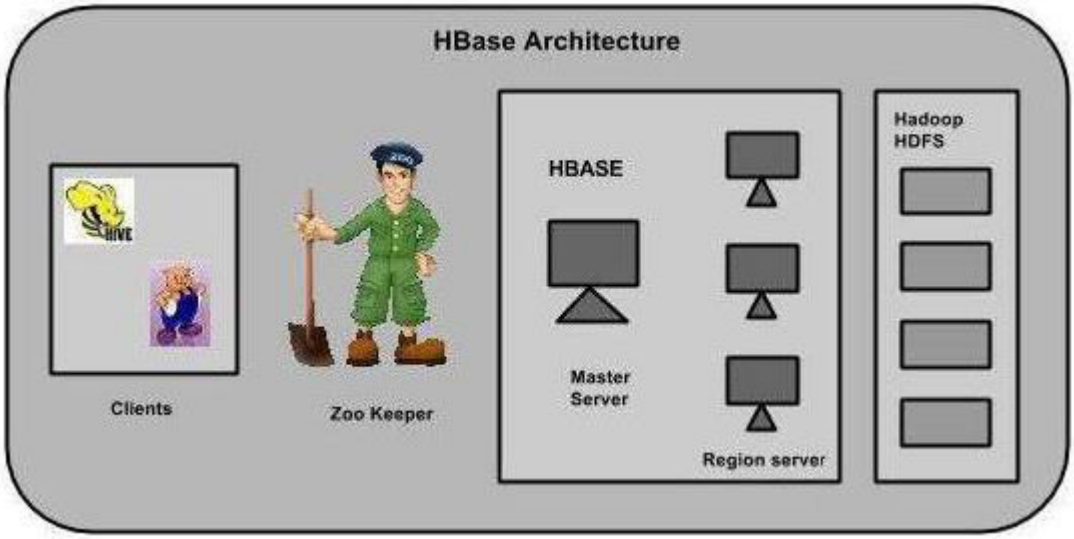
Features of HBase:

- HBase is linearly scalable and has automatic failure support.
- It provides consistent read and writes.
- It integrates with Hadoop, both as a source and destination.
- It has easy java API for client.
- It provides data replication across clusters.

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: **Client Library, Master Server,** and **Region Servers**. Region Servers can be added or removed as per requirement.

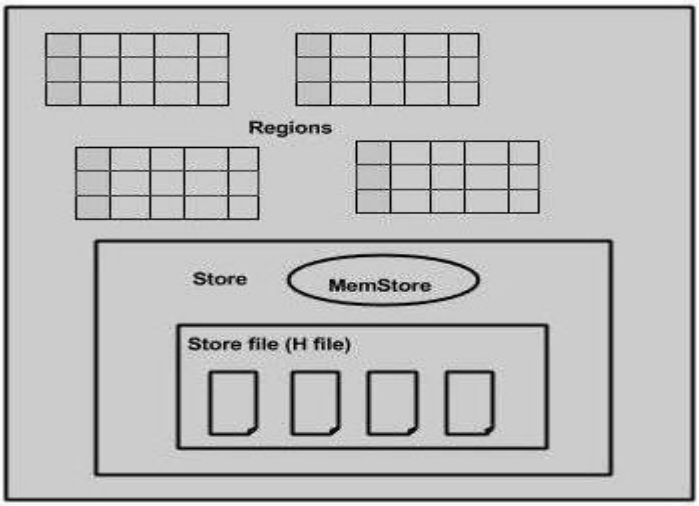
MasterServer:

- Assigns regions to the region servers and takes the help of 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 load balancing.
- 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.
- Communicate with client and handle data-related operations.
- Handle read and write requests for all the regions under it.
- Decide the size of the regions by following 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 HBase is stored here initially. Later, the data is transferred and saved in HFiles as blocks and the Memstore is flushed.

ZooKeeper:
Zookeeper is a centralized monitoring server which maintains configuration information and provides distributed synchronization. Distributed synchronization is to access the distributed applications running across the cluster with the responsibility of providing coordination services between nodes. If the client wants to communicate with regions, the server's client has to approach ZooKeeper first.

Services provided by ZooKeeper:

- Maintains Configuration information & provides distributed synchronization.
- Client Communication establishment with region servers.
- Master Server's usability of ephemeral nodes for discovering available servers in the cluster.
- To track server failure and network partitions.

Master and HBase slave nodes (region servers) registered themselves with ZooKeeper. The client needs access to zookeeper ZKquoram configuration to connect with master and region servers.
During a failure of nodes that present in HBase cluster, ZKquoram will trigger error messages and it starts to repair the failed nodes.

→ Start HBase Shell:
start-hbase.sh
hbase shell

→ **Stop HBase Shell:**
stop-hbase.sh

→ **Restart HBase Shell:**
sudo service hbase-master restart
sudo service hbase-regionserver restart

1) Create Syntax: create '<table name>','<column family>'
create 'emp', 'personal data', 'professional data'

2) List Tables:
list

3) Describe Table:
describe 'emp'

4) Insert Syntax: put '<table name>','row_key','<column_family:column_name>','<value>'
put 'emp','1','personal data:name','Saif'
put 'emp','1','personal data:city','Mumbai'
put 'emp','1','professional data:designation','Trainer'
put 'emp','1','professional data:salary','5000'

put 'emp','2','personal data:name','Ram'
put 'emp','2','personal data:city','Balewadi'
put 'emp','2','professional data:designation','Developer'
put 'emp','2','professional data:salary','10000'

5) Scan Syntax:
scan 'emp'

6) Limit Data:
scan 'emp', {LIMIT => 1}

7) Use Columns to restrict data:
scan 'emp', {LIMIT => 2, COLUMN => ['personal data:city']}
scan 'emp', {LIMIT => 2, COLUMN => ['personal data:city','professional data:designation']}

8) We could also specify STARTROW from where the results should return:
scan 'emp', {LIMIT => 2, COLUMN => ['personal data:city','personal data:name'], STARTROW =>

'2'}

9) STOPROW specification will specify where to stop:
scan 'emp', {LIMIT => 2, COLUMN => ['personal data:city','personal data:name'], STARTROW => '1', STOPROW => '2'}

put 'emp','3','personal data:name','Tausif'
put 'emp','3','personal data:city','NIBM'
put 'emp','3','professional data:designation','Architect'
put 'emp','3','professional data:salary','50000'

scan 'emp', {LIMIT => 2, COLUMN => ['personal data:city','personal data:name'], STARTROW => '1', STOPROW => '3'}

10) Exists Syntax:
exists 'emp'

11) Count: Use count command to get the total records in a table.
Syntax: count 'table_name>'
count 'emp'

12) Filter:
In this tutorial, we will learn how to fetch the data by filtering records from HBase table using predicate conditions. In order to use filters, you need to import certain java classes into Shell.
SingleColumnValueFilter:
import org.apache.hadoop.hbase.filter.SingleColumnValueFilter
import org.apache.hadoop.hbase.filter.CompareFilter
import org.apache.hadoop.hbase.filter.BinaryComparator

a) Filter All Rows:
scan 'emp', {FILTER=>SingleColumnValueFilter.new(Bytes.toBytes('personal data'),Bytes.toBytes('city'),CompareFilter::CompareOp.valueOf('EQUAL'),BinaryComparator.new(Bytes.toBytes('Mumbai')))}

b) Filter Data for Specific Row:
scan 'emp', {FILTER => "ValueFilter (=,'binaryprefix:50000')"}

13) get: Use get to retrieve the data from a single row and its columns.
Syntax: get 'table name','<row_key>','<column_key>'
a) Below command, returns column_key 2 with all columns.
get 'emp','2'
b) This again returns column_key 2 but with few column names.
get 'emp','2',{COLUMNS => ['personal data:name','professional data:salary']}

14) Disable and Enable Table: Use 'disable' to disable a table. Prior to delete a table or change its setting, first, you need to disable the table.
Syntax: disable '<table_name>'
disable 'emp'

a) Checking Disabled Table: Use is_disabled to check if the table is disabled. When it disabled it returns 'true'.
is_disabled 'emp'
describe 'emp'

Once the table is disabled, you cannot perform regular manipulation commands. For example running a scan on the disable table results below error.
scan 'emp'

15) Enable Table: Use 'enable' to enable a disabled table. You need to enable a disabled table first to perform any regular commands.
Syntax: enable '<table_name>'
enable 'emp'

a) Checking Disabled Table: Use is_disabled to check if the table is enabled. When it enabled it returns 'false'.
is_disabled 'emp'
describe 'emp'
16) Copy existing tables to a new table:
Make sure you enable snapshot in hbase-site.xml

```
 <property>
   <name>hbase.snapshot.enabled</name>
   <value>true</value>
  </property>
```

Syntax:
a) snapshot 'x' ,'snapshot_x'
b) clone_snapshot 'snapshot_x' ,'another_x'

snapshot 'emp', 'emp-snapshot'
clone_snapshot 'emp-snapshot', 'emp_new'

List & Delete Snapshots:
list_snapshots
delete_snapshot 'emp-snapshot'

17) Delete: Let's see how to remove an entire row and a specific column cell of a row from an HBase table using delete and deleteall commands.

a) deleteall: Use deleteall to remove a specified row from an HBase table. This takes table name and row as a mandatory argument; optionally column and timestamp. It also supports deleting a row range using a row key prefix.
Syntax: deleteall '<table_name>', 'row_key', '<column_name>'
==> This removes row 1 and all its cells from a table 'emp'.
deleteall 'emp_new', '1'

b) delete: Using the delete command, you can delete a specific cell in a table.
Syntax: delete '<table name>', '<row>', '<column name >', '<time stamp>'
delete 'emp_new', '3', 'professional data:salary', 1616488939427

18) DROP Table: Using the drop command, you can delete a table. Before dropping a table, you have to disable it.
drop 'emp_new'
disable 'emp_new'
is_disabled 'emp_new'
drop 'emp_new'
exists 'emp_new'