Big Data Technologies

Aakash Ahuja
aakash@itmtb.com
+91 7709 009634

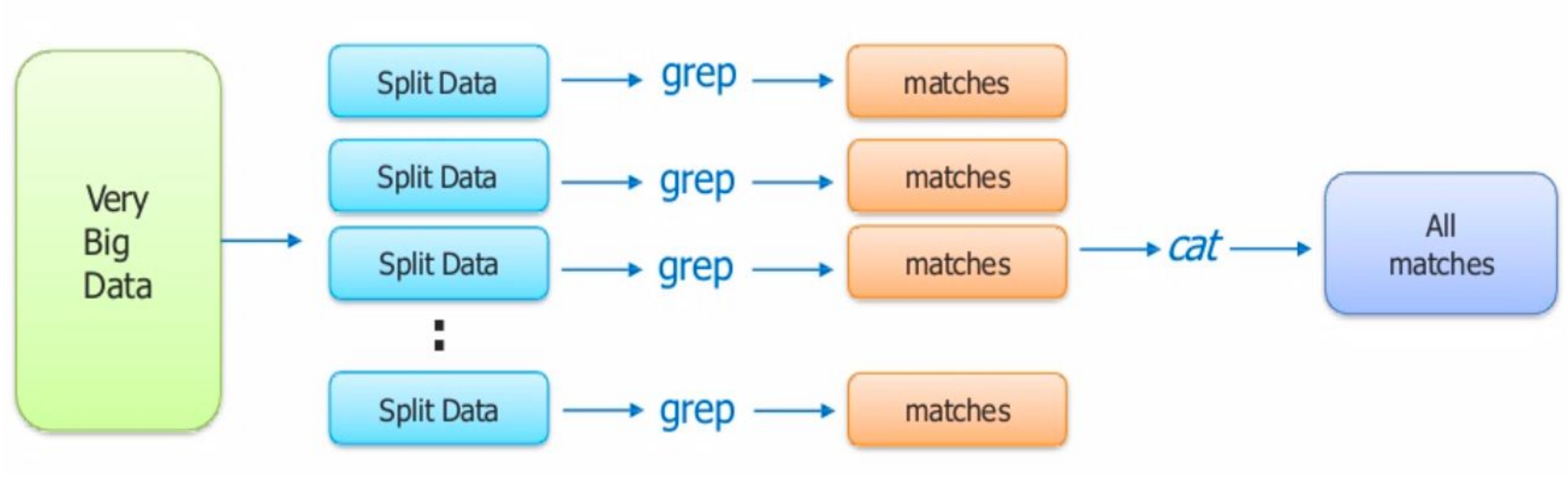
Basics of MapReduce programming

Basics of Mapreduce programming

In this session, you will learn about:

- Mapreduce input/output formats
- Data types and custom writables
- Combiners/partitioners

Traditional way



Challenges

Critical path problem

It is the amount of time taken to finish the job without delaying the next milestone or actual completion date. So, if, any of the machines delays the job, the whole work gets delayed.

Reliability problem

What if, any of the machines which is working with a part of data fails? The management of this failover becomes a challenge.

Equal split issue

How will I divide the data into smaller chunks so that each machine gets even part of data to work with. In other words, how to equally divide the data such that no individual machine is overloaded or under utilized.

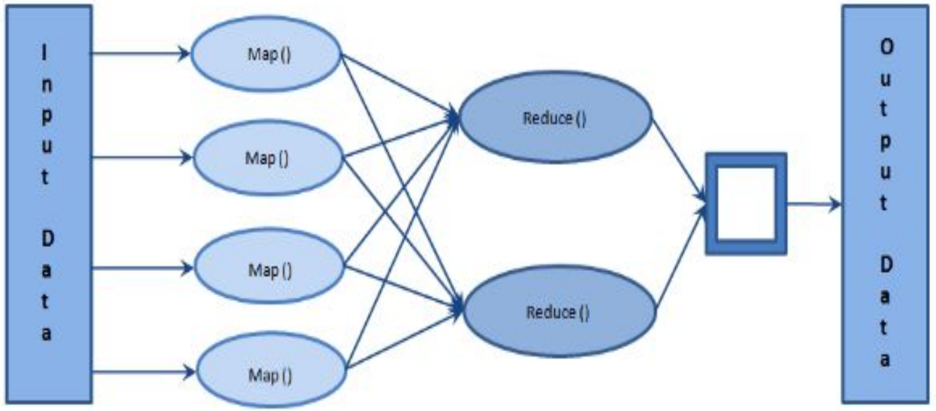
Single split may fail

If any of the machine fails to provide the output, I will not be able to calculate the result. So, there should be a mechanism to ensure this fault tolerance capability of the system.

Aggregation of result

There should be a mechanism to aggregate the result generated by each of the machines to produce the final output.

MapReduce way



- MapReduce consists of two distinct tasks – Map and Reduce.
- As the name MapReduce suggests, reducer phase takes place after mapper phase has been completed.
- So, the first is the map job, where a block of data is read and processed to produce key-value pairs as intermediate outputs.
- The output of a Mapper or map job (key-value pairs) is input to the Reducer.
- The reducer receives the key-value pair from multiple map jobs.
- Then, the reducer aggregates those intermediate data tuples (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.

Data Types - Need

Data types need to spread data across the entire cluster efficiently for Hadoop to work.

Data needs to be serialized - converted to byte form - while spreading across the cluster.

On the target node data needs to be deserialized - converted back to original format from the received byte format.

Java achieves this by implementing the Serializable interface. But it is too heavy to be implemented by Hadoop.

Data Types

Hadoop has its own equivalent data types called Writable data types.

These Writable data types are passed as parameters (input and output key-value pairs) for the mapper and reducer.

Writable data types are meant for writing the data to the local disk and it is a serialization format.

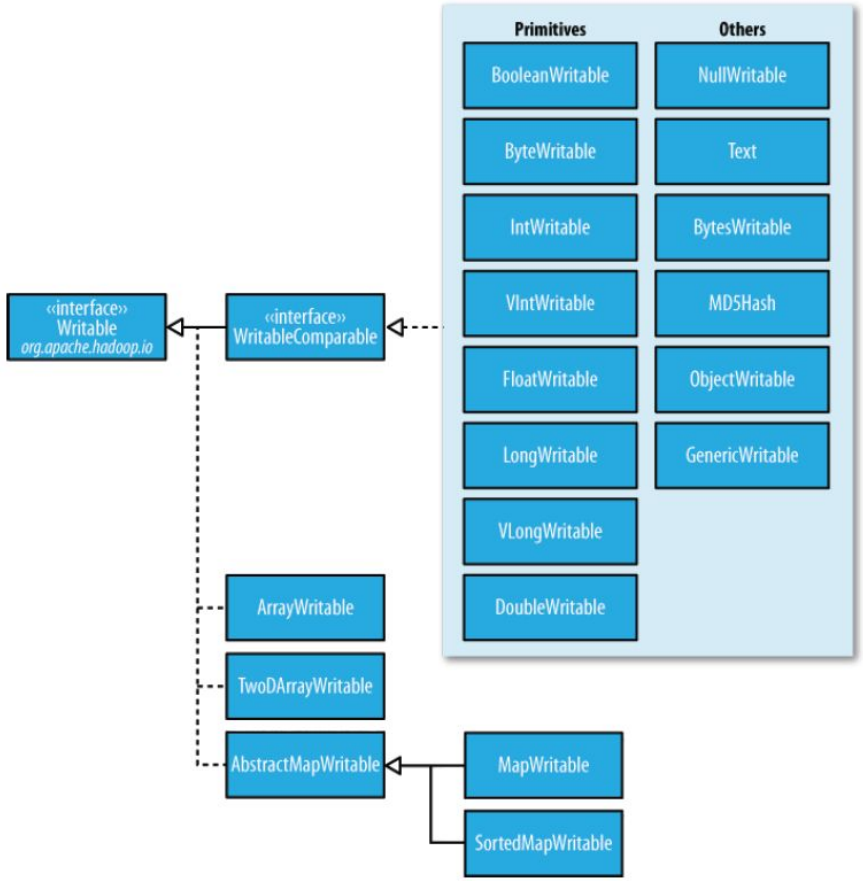
For keys, it also implements the Comparable interface.

Comparable interface is used for comparing when the reducer sorts the keys, and Writable can write the result to the local disk.

WritableComparable is a combination of Writable and Comparable interfaces.

```
public interface Writable
{
   void readFields(DataInput in);
   void write(DataOutput out);
}
```

```
public interface WritableComparable extends Writable,
Comparable
{
   void readFields(DataInput in);
   void write(DataOutput out);
   int compareTo(WritableComparable o)
}
```



Integer –> IntWritable: It is the Hadoop variant of Integer. It is used to pass integer numbers as key or value.

Float –> FloatWritable: Hadoop variant of Float used to pass floating point numbers as key or value.

Long –> LongWritable: Hadoop variant of Long data type to store long values.

Short –> ShortWritable: Hadoop variant of Short data type to store short values.

Double –> DoubleWritable: Hadoop variant of Double to store double values.

String –> Text: Hadoop variant of String to pass string characters as key or value.

Byte –> ByteWritable: Hadoop variant of byte to store sequence of bytes.

null –> NullWritable: Hadoop variant of null to pass null as a key or value. Usually NullWritable is used as data type for output key of the reducer, when the output key is not important in the final result.

Custom writables

Some use cases might demand creation of a custom writable.

An example can be an extension of the word count example, where in you want to count occurence of two words together.

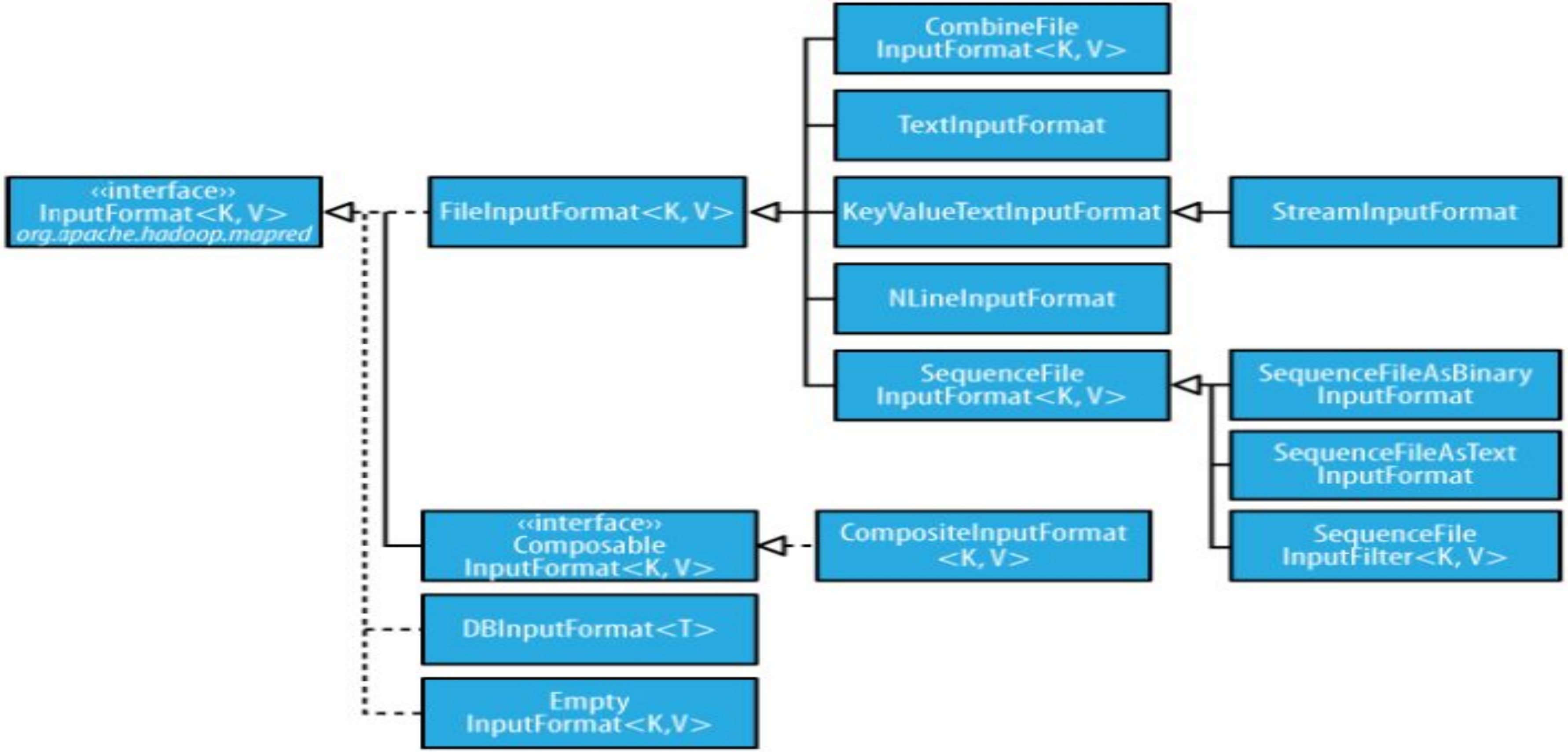
Code review - at the end of this topic.

InputFormats

How the input files are split up and read in Hadoop is defined by the InputFormat.

A Hadoop InputFormat is the first component in Map-Reduce.

It is responsible for creating the input splits and dividing them into records.


«interface» InputFormat<K, V> org.apache.hadoop.mapred
FileInputFormat<K, V>
CombineFile InputFormat<K, V>
TextInputFormat
KeyValueTextInputFormat
StreamInputFormat
NLineInputFormat
SequenceFile InputFormat<K, V>
SequenceFileAsBinary InputFormat
SequenceFileAsText InputFormat
SequenceFile InputFilter<K, V>
«interface» Composable InputFormat<K, V>
CompositeInputFormat <K, V>
DBInputFormat<T>
Empty InputFormat<K,V>

FileInputFormat

It is the base class for all file-based Input formats. Hadoop FileInputFormat specifies input directory where data files are located. FileInputFormat will read all files and divides these files into one or more InputSplits.

TextInputFormat

It is the default InputFormat of MapReduce. TextInputFormat treats each line of each input file as a separate record and performs no parsing. This is useful for unformatted data or line-based records like log files.

- Key – It is the byte offset of the beginning of the line within the file (not whole file just one split), so it will be unique if combined with the file name.
- Value – It is the contents of the line, excluding line terminators.

KeyValueTextInputFormat

Similar to TextInputFormat but breaks the line itself into key and value by a tab character ('/t').

SequenceFileInputFormat

Reads binary files that store sequences of binary key-value pairs. Here Key & Value both are user-defined.

SequenceFileAsTextInputFormat

Converts the sequence file key values to Text objects. This InputFormat makes sequence files suitable input for streaming.

NLineInputFormat

Another form of TextInputFormat where the keys are byte offset of the line and values are contents of the line.

DBInputFormat

Reads data from a relational database, using JDBC. Best for loading relatively small datasets, perhaps for joining with large datasets from HDFS using MultipleInputs. Here Key is LongWritables while Value is DBWritables.

OutputFormats

The Hadoop Output Format checks the Output-Specification of the job. It determines how data from reducers is written to output files.

TextOutputFormat

MapReduce default Hadoop reducer Output Format is TextOutputFormat, which writes (key, value) pairs on individual lines of text files and its keys and values can be of any type since TextOutputFormat turns them to string by calling toString() on them.

SequenceFileOutputFormat

It is an Output Format which writes sequences files for its output and it is intermediate format use between MapReduce jobs, which rapidly serialize arbitrary data types to the file; and the corresponding SequenceFileInputFormat will deserialize the file into the same types and presents the data to the next mapper in the same manner as it was emitted by the previous reducer, since these are compact and readily compressible.

SequenceFileAsBinaryOutputFormat

It is another form of SequenceFileInputFormat which writes keys and values to sequence file in binary format.

MapFileOutputFormat

It is another form of FileOutputFormat in Hadoop Output Format, which is used to write output as map files. The key in a MapFile must be added in order, so we need to ensure that reducer emits keys in sorted order.

MultipleOutputs

It allows writing data to files whose names are derived from the output keys and values, or in fact from an arbitrary string.

DBOutputFormat

DBOutputFormat in Hadoop is an Output Format for writing to relational databases and HBase.

Mapper

Mapper maps input key/value pairs to a set of intermediate key/value pairs.

Maps are the individual tasks that transform input records into intermediate records.

The Hadoop MapReduce framework spawns one map task for each InputSplit generated by the InputFormat for the job.

Overall, Mapper implementations are passed to the JobConf for the job via the JobConfigurable.configure(JobConf) method.

The framework then calls map(WritableComparable, Writable, OutputCollector, Reporter) for each key/value pair in the InputSplit for that task.

Output pairs are collected with calls to OutputCollector.collect(WritableComparable,Writable).

Applications can use the Reporter to report progress, set application-level status messages and update Counters, or just indicate that they are alive.

All intermediate values associated with a given output key are subsequently grouped by the framework, and passed to the Reducer(s) to determine the final output.

The Mapper outputs are sorted and then partitioned per Reducer.

Reducer

Reducer implementations are passed the JobConf for the job via the JobConfigurable.configure(JobConf) method.

The framework then calls reduce(WritableComparable, Iterator, OutputCollector, Reporter) method for each <key, (list of values)> pair in the grouped inputs.

Reducer has 3 primary phases: shuffle, sort and reduce.

Shuffle

Input to the Reducer is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.

Sort

The framework groups Reducer inputs by keys (since different mappers may have output the same key) in this stage.

The shuffle and sort phases occur simultaneously; while map-outputs are being fetched they are merged.

Reduce

In this phase the reduce(WritableComparable, Iterator, OutputCollector, Reporter) method is called for each <key, (list of values)> pair in the grouped inputs.

The output of the reduce task is typically written to the FileSystem via OutputCollector.collect(WritableComparable, Writable).

Partitioner

Partitioner controls the partitioning of the keys of the intermediate map-outputs. The key is used to derive the partition, typically by a *hash function*. The total number of partitions is the same as the number of reduce tasks for the job. Hence this controls which of the m reduce tasks the intermediate key (and hence the record) is sent to for reduction.

HashPartitioner is the default Partitioner.

Reporter

Reporter is a facility for MapReduce applications to report progress, set application-level status messages and update Counters.

In scenarios where the application takes a significant amount of time to process individual key/value pairs, this is crucial since the framework might assume that the task has timed-out and kill that task. Another way to avoid this is to set the configuration parameter mapred.task.timeout to a high-enough value (or even set it to *zero* for no time-outs).

OutputCollector

This is used to collect data output by the Mapper or the Reducer (either the intermediate outputs or the output of the job).

Job Configuration

JobConf represents a MapReduce job configuration.

JobConf is the primary interface for a user to describe a MapReduce job to the Hadoop framework for execution. The framework tries to faithfully execute the job as described by JobConf, however:

JobConf is typically used to specify the Mapper, combiner (if any), Partitioner, Reducer, InputFormat, OutputFormat and OutputCommitter implementations.

JobConf also indicates the set of input files (setInputPaths(JobConf, Path...) /addInputPath(JobConf, Path)) and (setInputPaths(JobConf, String) /addInputPaths(JobConf, String)) and where the output files should be written (setOutputPath(Path)).

Optionally, JobConf is used to specify other advanced facets of the job such as files to be put in the DistributedCache, whether intermediate and/or job outputs are to be compressed (and how), debugging via user-provided scripts (setMapDebugScript(String)/setReduceDebugScript(String)) , whether job tasks can be executed in a *speculative* manner (setMapSpeculativeExecution(boolean))/(setReduceSpeculativeExecution(boolean)) , maximum number of attempts per task (setMaxMapAttempts(int)/setMaxReduceAttempts(int)) , percentage of tasks failure which can be tolerated by the job (setMaxMapTaskFailuresPercent(int)/setMaxReduceTaskFailuresPercent(int)) etc.

Sample code description - Map

- We have created a class Map that extends the class Mapper which is already defined in the MapReduce Framework.
- We define the data types of input and output key/value pair after the class declaration using angle brackets.
- Both the input and output of the Mapper is a key/value pair.
- Input:
  - The *key* is nothing but the offset of each line in the text file: *LongWritable*
  - The *value* is each individual line (as shown in the figure at the right): *Text*
- Output:
  - The *key* is the tokenized words: *Text*
  - We have the hardcoded *value* in our case which is 1: *IntWritable*
  - Example – Dear 1, Bear 1, etc.
- We have written a java code where we have tokenized each word and assigned them a hardcoded value equal to *1*.

Sample code description - Reduce

- We have created a class Reduce which extends class Reducer like that of Mapper.
- We define the data types of input and output key/value pair after the class declaration using angle brackets as done for Mapper.
- Both the input and the output of the Reducer is a key-value pair.
- Input:
    - The *key* nothing but those unique words which have been generated after the sorting and shuffling phase: *Text*
    - The *value* is a list of integers corresponding to each key: *IntWritable*
    - Example – Bear, [1, 1], etc.
- Output:
    - The *key* is all the unique words present in the input text file: *Text*
    - The *value* is the number of occurrences of each of the unique words: *IntWritable*
    - Example – Bear, 2; Car, 3, etc.
- We have aggregated the values present in each of the list corresponding to each key and produced the final answer.
- In general, a single reducer is created for each of the unique words, but, you can specify the number of reducer in mapred-site.xml.

Sample code description - Driver

- In the driver class, we set the configuration of our MapReduce job to run in Hadoop.
- We specify the name of the job , the data type of input/output of the mapper and reducer.
- We also specify the names of the mapper, combiner and reducer classes.
- The path of the input and output folder is also specified.
- Input and Output formats are set
- The main () method is the entry point for the driver. In this method, we instantiate a new Configuration object for the job.

Sample code description - Execution

Assuming HADOOP_HOME is the root of the installation and HADOOP_VERSION is the Hadoop version installed, compile WordCount.java and create a jar:

```
$ mkdir wordcount_classes

$ javac -classpath ${HADOOP_HOME}/hadoop-${HADOOP_VERSION}-core.jar -d wordcount_classes WordCount.java

$ jar -cvf /usr/aa/wordcount.jar -C wordcount_classes/ .
```

Assuming that:

- /usr/aa/wordcount/input - input directory in HDFS
- /usr/aa/wordcount/output - output directory in HDFS

Run the application:

```
$ bin/hadoop jar /usr/aa/wordcount.jar org.myorg.WordCount /usr/aa/wordcount/input /usr/aa/wordcount/output
```

Mapreduce streaming

Both the mapper and the reducer are executables that read the input from stdin (line by line) and emit the output to stdout.

The utility will create a Map/Reduce job, submit the job to an appropriate cluster, and monitor the progress of the job.

When an executable is specified for mappers, each mapper task will launch the executable as a separate process when the mapper is initialized.

As the mapper task runs, it converts its inputs into lines and feed the lines to the stdin of the process.

In the meantime, the mapper collects the line oriented outputs from the stdout of the process and converts each line into a key/value pair, which is collected as the output of the mapper.

By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) will be the value.

If there is no tab character in the line, then entire line is considered as key and the value is null. However, this can be customized by setting -inputformat command option, as discussed later.

Mapreduce streaming

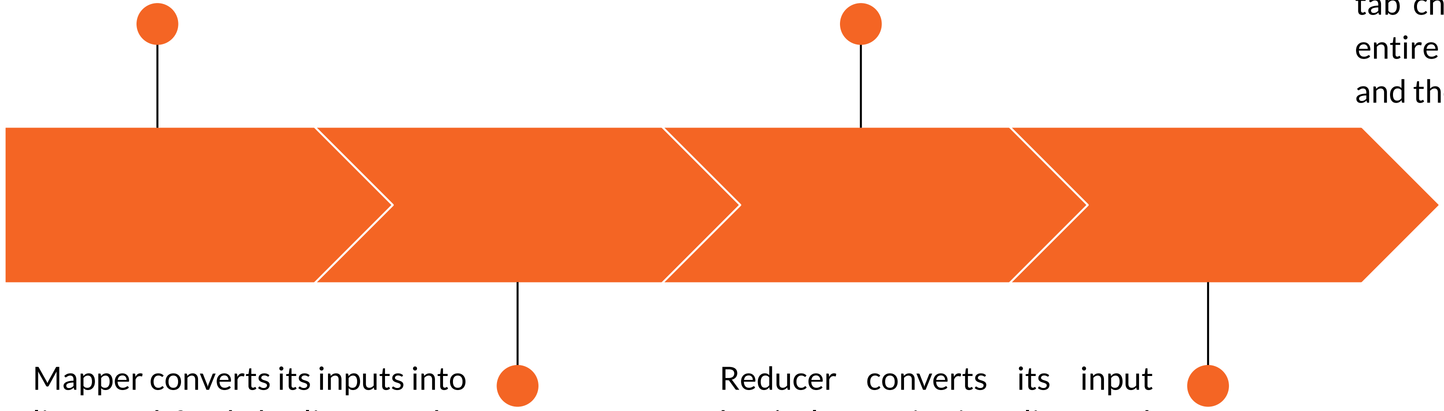
When an executable is specified for reducers, each reducer task will launch the executable as a separate process then the reducer is initialized.

As the reducer task runs, it converts its input key/values pairs into lines and feeds the lines to the stdin of the process.

In the meantime, the reducer collects the line oriented outputs from the stdout of the process, converts each line into a key/value pair, which is collected as the output of the reducer.

By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) is the value.

This can be customized by setting -outputformat command option.



Mapper task will launch the executable as a separate process when the mapper is initialized.

Mapper converts its inputs into lines and feed the lines to the stdin of the process. In the meantime, the mapper collects the line oriented outputs from the stdout of the process and converts each line into a key/value pair.

Each reducer task will launch the executable as a separate process then the reducer is initialized.

Reducer converts its input key/values pairs into lines and feeds the lines to the stdin of the process. In the meantime, the reducer collects the line oriented outputs from the stdout of the process, converts each line into a key/value pair.

By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) will be the value. If there is no tab character in the line, then entire line is considered as key and the value is null.

Streaming options

Parameter	Optional/Required	Description
-input directory name or filename	Required	Input location for mapper
-output directory name	Required	Output location for reducer
-mapper executable or JavaClassName	Required	Mapper executable
-reducer executable or JavaClassName	Required	Reducer executable
-file filename	Optional	Make the mapper, reducer, or combiner executable available locally on the compute nodes
-inputformat JavaClassName	Optional	Class you supply should return key/value pairs of Text class. If not specified, TextInputFormat is used as the default
-outputformat JavaClassName	Optional	Class you supply should take key/value pairs of Text class. If not specified, TextOutputformat is used as the default
-cmdenv name=value	Optional	Pass environment variable to streaming commands
-inputreader	Optional	For backwards-compatibility: specifies a record reader class (instead of an input format class)

Streaming options continued

Parameter	Optional/Required	Description
-verbose	Optional	Verbose output
-lazyOutput	Optional	Create output lazily. For example, if the output format is based on FileOutputFormat, the output file is created only on the first call to output.collect (or Context.write)
-numReduceTasks	Optional	Specify the number of reducers
-mapdebug	Optional	Script to call when map task fails
-reducedebug	Optional	Script to call when reduce task fails

Generic command options

Parameter	Optional/Required	Description
-conf configuration_file	Optional	Specify an application configuration file
-D property=value	Optional	Use value for given property
-fs host:port or local	Optional	Specify a namenode
-jt host:port or local	Optional	Specify a job tracker
-files	Optional	Specify comma-separated files to be copied to the Map/Reduce cluster
-libjars	Optional	Specify comma-separated jar files to include in the classpath
-archives	Optional	Specify comma-separated archives to be unarchived on the compute machines

Generic command options must always be supplied before streaming command options.

HBase

In this session, you will learn about:

- Introduction to HBase
- Architecture
- Installation
- HBase shell
- Common operations
- Java API
- Security

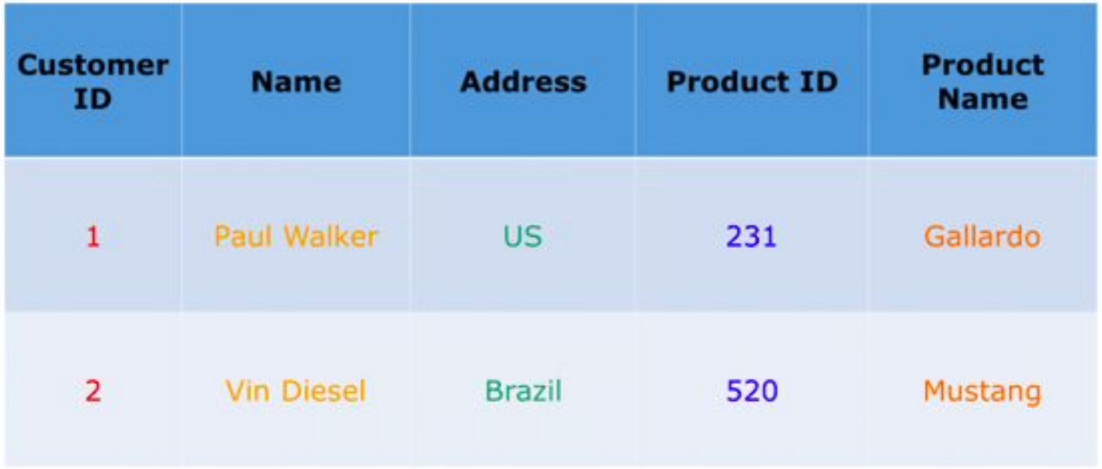
HBase

- HBase is an open source, multidimensional, distributed, scalable and a NoSQL database written in Java.
- HBase is not a relational database.
- HBase runs on top of HDFS (Hadoop Distributed File System) and provides BigTable like capabilities to Hadoop.
- It is designed to provide a fault tolerant way of storing large collection of sparse data sets.
- HBase achieves high throughput and low latency by providing faster Read/Write Access on huge data sets.
- HBase is suitable for random access of data.
- It is column oriented.
- Provides compression, in-memory operations and Bloom filters (data structure which tells whether a value is present in a set or not).
- Apache HBase is modelled after Google's BigTable, which is used to collect data and serve request for various Google services like Maps, Finance, Earth etc.

Row-oriented vs column-oriented Databases:

Row-oriented databases store table records in a sequence of rows. Whereas column-oriented databases store table records in a sequence of columns, i.e. the entries in a column are stored in contiguous locations on disks.

To better understand it, let us take an example and consider the table below.



Customer ID	Name	Address	Product ID	Product Name
1	Paul Walker	US	231	Gallardo
2	Vin Diesel	Brazil	520	Mustang

In row-oriented databases data is stored on the basis of rows or tuples.

While the column-oriented databases store this data as:

1,2, Paul Walker, Vin Diesel, US, Brazil, 231, 520, Gallardo, Mustang

In a column-oriented databases, all the column values are stored together like first column values will be stored together, then the second column values will be stored together and data in other columns are stored in a similar manner. Columns are logically grouped into column families which can be either created during schema definition or at runtime.

Column based storage provides faster access of data.

Suitable for OLTP workloads.

Architecture

HBase data model

Architecture components

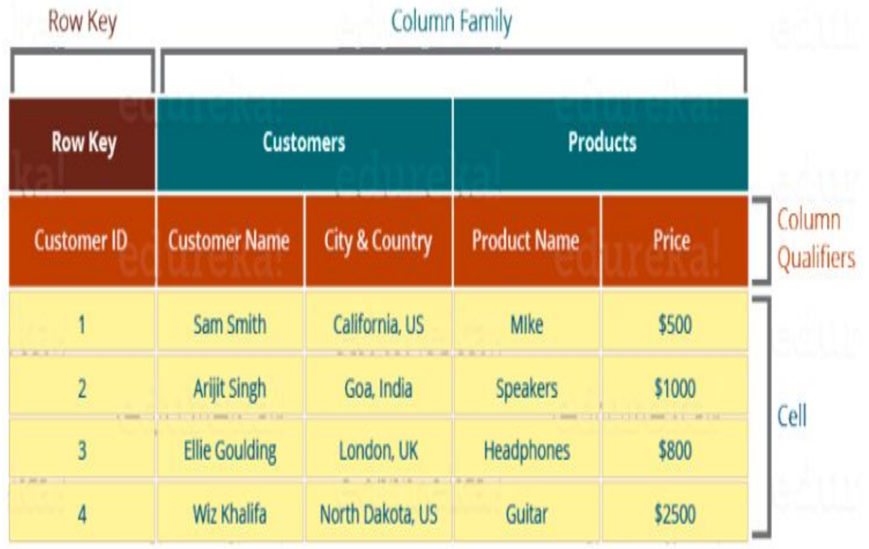
Write mechanism

Read mechanism

Performance optimization

Data model

- Tables: Data is stored in a table format in HBase. But here tables are in column-oriented format.
- Row Key: Row keys are used to search records which make searches fast.
- Column Families: Various columns are combined in a column family. These column families are stored together which makes the searching process faster because data belonging to same column family can be accessed together in a single seek.


Row Key
Column Family
Column Qualifiers
Cell

Row Key	Customers		Products	
Customer ID	Customer Name	City & Country	Product Name	Price
1	Sam Smith	California, US	Mike	$500
2	Arijit Singh	Goa, India	Speakers	$1000
3	Ellie Goulding	London, UK	Headphones	$800
4	Wiz Khalifa	North Dakota, US	Guitar	$2500

- Column Qualifiers: Each column's name is known as its column qualifier.
- Cell: Data is stored in cells. The data is dumped into cells which are specifically identified by rowkey and column qualifiers.
- Timestamp: Timestamp is a combination of date and time. Whenever data is stored, it is stored with its timestamp. This makes easy to search for a particular version of data.

Image courtesy: Edureka

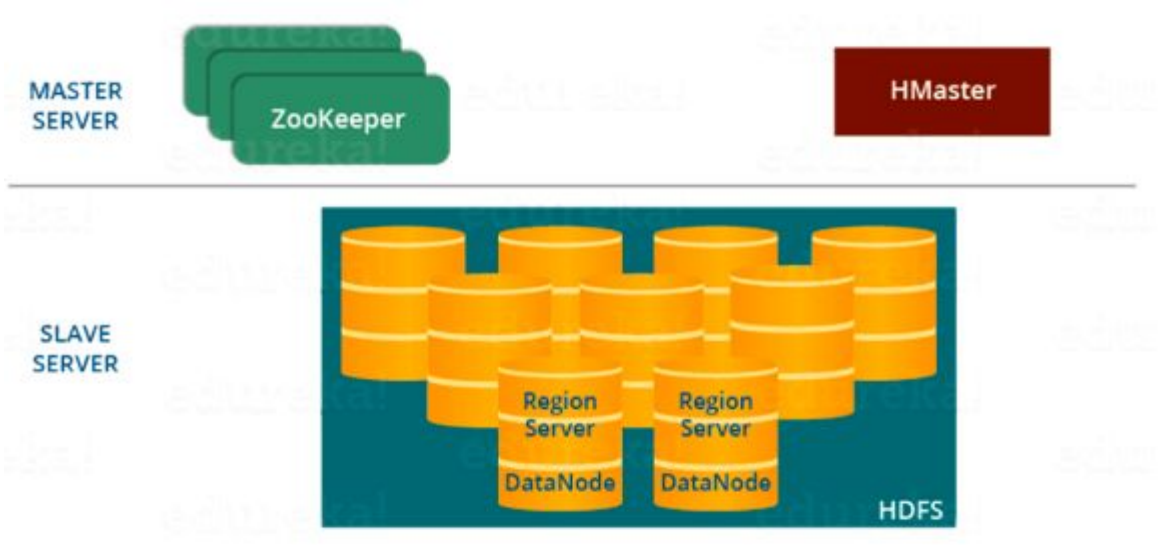
Architecture components

HMaster Server

HBase Region Server

Regions

Zookeeper

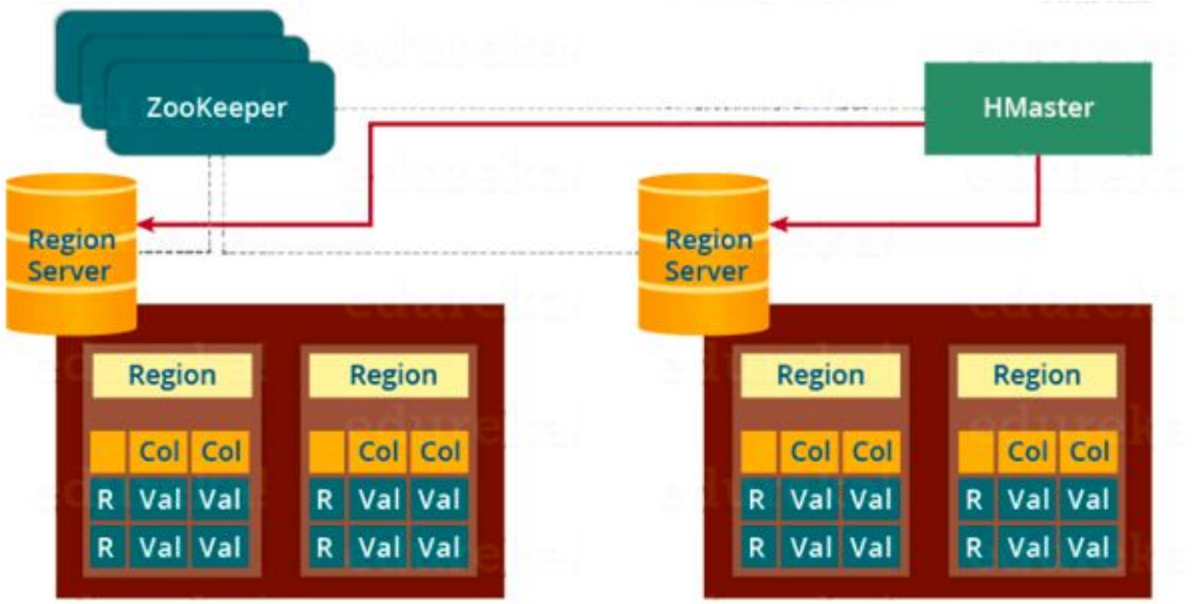


Architecture components - Region

- HBase tables can be divided into a number of regions in such a way that all the columns of a column family is stored in one region.
- A region contains all the rows between the start key and the end key assigned to that region.
- Each region contains the rows in a sorted order.
- Many regions are assigned to a Region Server, which is responsible for handling, managing, executing reads and write operations on that set of regions.
- A Region has a default size of 256MB which can be configured according to the need.
- A Group of regions is served to the clients by a Region Server.
- A Region Server can serve approximately 1000 regions to the client.

Architecture components - Master

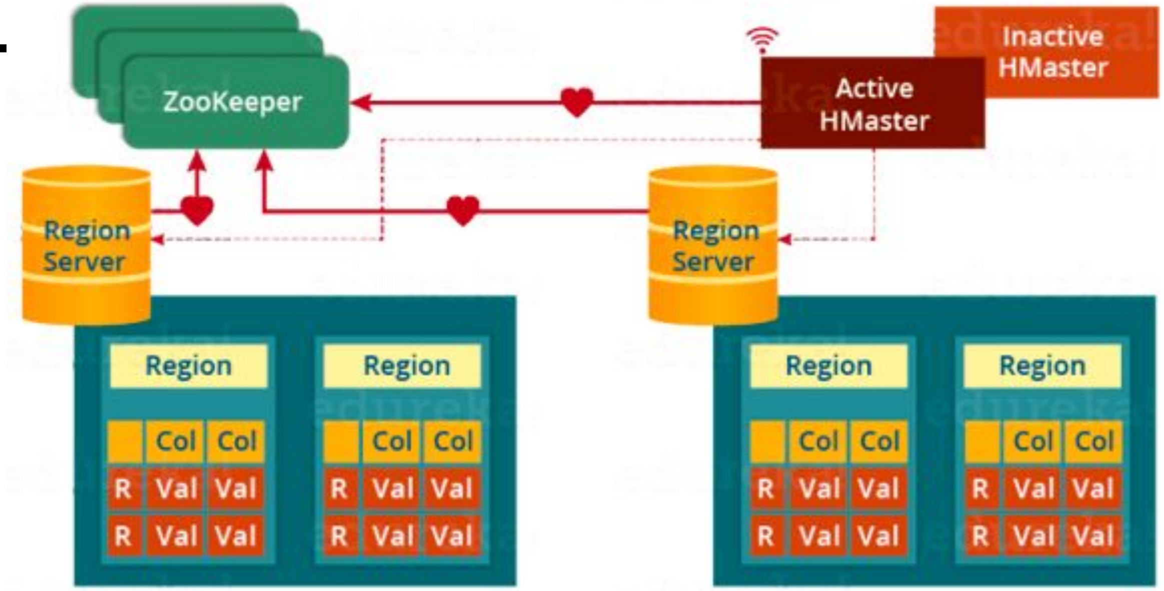
- HBase HMaster performs DDL operations (create and delete tables) and assigns regions to the Region servers
- It coordinates and manages the Region Server (similar as NameNode manages DataNode in HDFS



- It assigns regions to the Region Servers on startup and re-assigns regions to Region Servers during recovery and load balancing.
- It monitors all the Region Server's instances in the cluster (with the help of Zookeeper) and performs recovery activities whenever any Region Server is down.
- It provides an interface for creating, deleting and updating tables.

Architecture components - Zookeeper

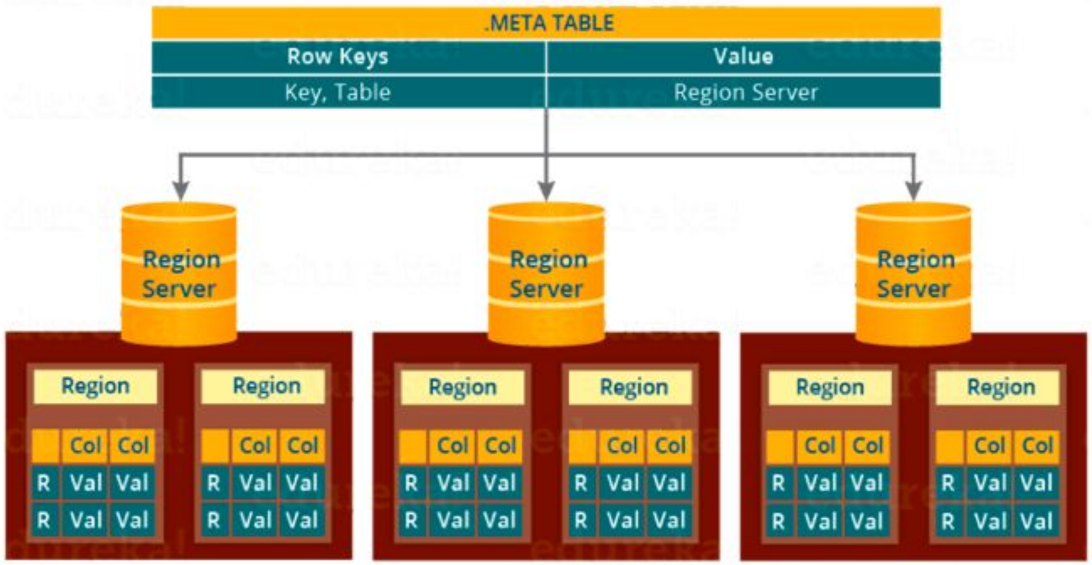
- Zookeeper acts like a coordinator inside HBase distributed environment. It helps in maintaining server state inside the cluster by communicating through sessions.
- Changes standby to active server after failure.



- Every Region Server along with HMaster Server sends continuous heartbeat at regular interval to Zookeeper and it checks which server is alive. It also provides server failure notifications so that, recovery measures can be executed
- The active HMaster sends heartbeats to the Zookeeper while the inactive HMaster listens for the notification send by active HMaster. If the active HMaster fails to send a heartbeat the session is deleted and the inactive HMaster becomes active.
- If a Region Server fails to send a heartbeat, the session is expired and all listeners are notified about it. Then HMaster performs suitable recovery actions.

Architecture components - .META table

- The META table is a special HBase catalog table. It maintains a list of all the Regions Servers in the HBase storage system.
- .META file maintains the table in form of keys and values. Key represents the start key of the region and its id whereas the value contains the path of the Region Server.



Architecture components - Region server components

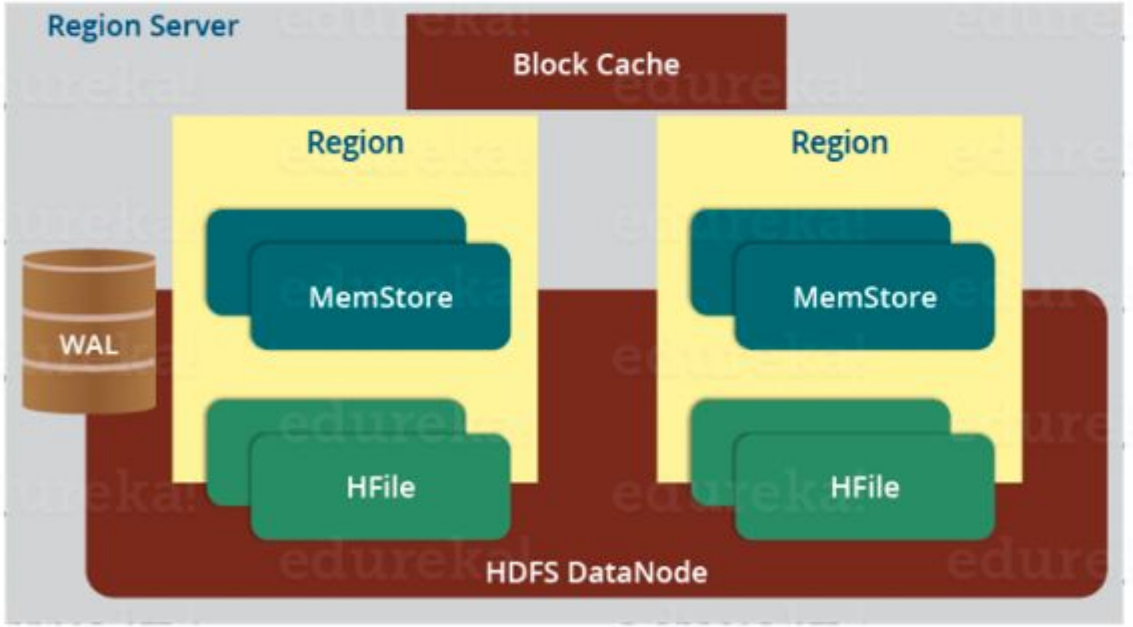


Image courtesy: Edureka

Architecture components - Region server components

A Region Server maintains various regions running on the top of HDFS. Components of a Region Server are:

- **WAL**: Write Ahead Log (WAL) is a file attached to every Region Server inside the distributed environment. The WAL stores the new data that hasn’t been persisted or committed to the permanent storage. It is used in case of failure to recover the data sets.
- **Block Cache**: Block Cache resides in the top of Region Server. It stores the frequently read data in the memory. If the data in Block Cache is least recently used, then that data is removed from BlockCache.
- **MemStore**: It is the write cache. It stores all the incoming data before committing it to the disk or permanent memory. There is one MemStore for each column family in a region. there are multiple MemStores for a region because each region contains multiple column families. The data is sorted in lexicographical order before committing it to the disk.
- **HFile**: HFile is stored on HDFS. It stores the actual cells on the disk. MemStore commits the data to HFile when the size of MemStore exceeds.

Architecture components - Search in HBase

Whenever a client approaches with a read or writes requests to HBase following operation occurs:

1. The client retrieves the location of the .META table from the ZooKeeper.
2. The client then requests for the location of the Region Server of corresponding row key from the .META table to access it. The client caches this information with the location of the .META Table.
3. Then it will get the row location by requesting from the corresponding Region Server.
4. For future references, the client uses its cache to retrieve the location of .META table and previously read row key's Region Server.
5. Then the client will not refer to the META table, until and unless there is a miss because the region is shifted or moved. Then it will again request to the META server and update the cache.

Architecture components - Writes in HBase

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

- The edits are then appended at the end of the WAL file.
- This WAL file is maintained in every Region Server and Region Server uses it to recover data which is not committed to the disk.

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

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

Step 4: When the MemStore reaches the threshold, it dumps or commits the data into a HFile.

Architecture components - Writes - Memstore

- The MemStore always updates the data stored in it, in a lexicographical order (sequentially in a dictionary manner) as sorted KeyValues.
- When the MemStore reaches the threshold, it dumps all the data into a new HFile in a sorted manner.
- This HFile is stored in HDFS. HBase contains multiple HFiles for each Column Family.
- Over time, the number of HFile grows as MemStore dumps the data.

Architecture components - Writes - HFile

- The writes are placed sequentially on the disk. This makes write and search mechanism very fast.
- The HFile indexes are loaded in memory whenever an HFile is opened. This helps in finding a record in a single seek.
- The trailer is a pointer which points to the HFile’s meta block . It is written at the end of the committed file. It contains information about timestamp and bloom filters.

Architecture - Reads

As discussed in our search mechanism, first the client retrieves the location of the Region Server from .META Server if the client does not have it in its cache memory. Then it goes through the sequential steps as follows:

- For reading the data, the scanner first looks for the Row cell in Block cache. Here all the recently read key value pairs are stored.
- If Scanner fails to find the required result, it moves to the MemStore, as we know this is the write cache memory. There, it searches for the most recently written files, which has not been dumped yet in HFile.
- At last, it will use bloom filters and block cache to load the data from HFile.

Compaction & Region split

HBase combines HFiles to reduce the storage and reduce the number of disk seeks needed for a read. This process is called compaction. There are two types of compaction:

1. Minor Compaction: HBase automatically picks smaller HFiles and recommits them to bigger HFiles. This is called Minor Compaction.
2. Major Compaction: In Major compaction, HBase merges and recommits the smaller HFiles of a region to a new HFile. In this process, the same column families are placed together in the new HFile. It drops deleted and expired cell in this process. It increases read performance.

During this process, input-output disks and network traffic might get congested.

Whenever a region becomes large, it is divided into two child regions. Each region represents exactly a half of the parent region.

Crash and recovery

- Whenever a Region Server fails, ZooKeeper notifies to the HMaster about the failure.
- Then HMaster distributes and allocates the regions of crashed Region Server to many active Region Servers.
- The HMaster then distributes the WAL to all the Region Servers.
- Each Region Server re-executes the WAL to build the MemStore for that failed region’s column family.

Hbase shell commands

hbase shell

Information about logged in user:

whoami

Information about hbase version:

version

system status variants:

status 'simple'

status 'detailed'

status 'summary'

Hbase shell commands

Create table:

Syntax: create <tablename>, <columnfamilyname>

create 'testtab', 'cf1'

In order to check whether the table 'education' is created or not, we have to use the "list" command as mentioned below.

list

Syntax:list

Describe

Syntax:describe <table name>

describe 'testtab'

Hbase shell commands

Before a table can be dropped, it needs to be disabled:

Syntax: disable <tablename>

Enable a table

Syntax: enable <tablename>

Show filters available

Syntax: show_filters

This command displays all the filters present in HBase like ColumnPrefix Filter, TimestampsFilter, PageFilter, FamilyFilter, etc.

Hbase shell commands

drop a table

Syntax:drop <table name>

alter table

Syntax: alter <tablename>, NAME=><column familyname>, ...

Syntax: alter <tablename>, {NAME=><column familyname>, ...}, {NAME=><column familyname>, ...}

You cannot rename a column family.

Hbase shell commands

show count

Syntax: count <'tablename'>, CACHE =>1000

The command will retrieve the count of a number of rows in a table. The value returned by this one is the number of rows.

Current count is shown per every 1000 rows by default. Count interval may be optionally specified.

Hbase shell commands

Put data in a table

Syntax: put <'tablename'>,<'rowname'>,<'columnvalue'>,<'value'>

This command is used for following things:

1. It will put a cell 'value' at defined or specified table or row or column.

2. It will optionally coordinate time stamp.

Hbase shell commands

Get data from table

Syntax: get <'tablename'>, <'rowname'>, {< Additional parameters>}

Delete from a table

Syntax:delete <'tablename'>,<'row name'>,<'column name'>

This command will delete cell value at defined table of row or column. Delete must and should match the deleted cells coordinates exactly. When scanning, delete cell suppresses older versions of values.

Hbase shell commands

Truncate

Syntax: truncate <tablename>

After truncate of an hbase table, the schema will present but not the records. This command performs 3 functions; those are listed below:

Disables table if it already presents

Drops table if it already presents

Recreates the mentioned table

Hbase shell commands

Scan a table

Syntax: scan <'tablename'>, {Optional parameters}

This command scans entire table and displays the table contents.

Hbase shell commands

Examples:-

scan '.META.', {COLUMNS => 'info:regioninfo'}

It display all the meta data information related to columns that are present in the tables in HBase

scan 'testtab', {COLUMNS => ['cf1', 'cf2'], LIMIT => 10, STARTROW => 'xyz'}

It display contents of table testtab with their column families c1 and c2 limiting the values to 10

scan 'testtab', {COLUMNS => 'c1', TIMERANGE => [1704978426, 1704978626**]}**

It display contents of testtab with its column name c1 with the values present in between the mentioned time range attribute value.

Hbase Java API access

Sample code discussion

Refer to: More Java API references doc provided

Refer to https://hbase.apache.org/apidocs/index.html

Hbase Delete

When a Delete command is issued through the HBase client, no data is actually deleted. Instead a tombstone marker is set. User Scans and Gets automatically filter deleted cells until they get removed. HBase periodically removes deleted cells during compactions.

There are three types of tombstone markers:

1.	version delete marker:	Marks a single version of a column for deletion
2.	column delete marker:	Marks all versions of a column for deletion
3.	family delete marker:	Marks all versions of all columns for a column family for deletion

Hbase security - basics

With the default Apache HBase configuration, everyone is allowed to read from and write to all tables available in the system.

HBase can be configured to provide *User Authentication*, which ensures that only authorized users can communicate with HBase.

The authorization system is implemented at the RPC level, and is based on the Simple Authentication and Security Layer (SASL), which supports (among other authentication mechanisms) Kerberos.

SASL

Kerberos

Hbase security - kerberos

- Kerberos is a networked authentication protocol.
- It is designed to provide strong authentication for client/server applications by using key cryptography.
- The Kerberos protocol uses strong cryptography (AES, 3DES, …) so that a client can prove its identity to a server (and vice versa) across an insecure network connection.
- After a client and server have used Kerberos to prove their identities, they can also encrypt all of their communications to assure privacy and data integrity as they go about their business.
- At a high level, to access a service using Kerberos, each client must follow three steps:
    - Kerberos Authentication: The client authenticates itself to the Kerberos Authentication Server and receive a Ticket Granting Ticket (TGT).
    - Kerberos Authorization: The client request a service ticket from the Ticket Granting Server, which issues a ticket and a session key if the client TGT sent with the request is valid.
    - Service Request: The client uses the service ticket to authenticate itself to the server that is providing the service the client is using (e.g. HDFS, HBase, …)

Hbase security - https

A default HBase install uses insecure HTTP connections for Web UIs for the master and region servers.

To enable secure HTTPS connections, set hbase.ssl.enabled to true in hbase-site.xml.

This does not change the port used by the Web UI. To change the port for the web UI for a given HBase component, configure that port's setting in hbase-site.xml. These settings are:

hbase.master.info.port

hbase.regionserver.info.port

If you enable HTTPS, clients should:

- avoid using the non-secure HTTP connection
- connect to HBase using the https:// URL. Clients using the http:// URL will receive an HTTP response of 200, but will not receive any data

Hbase security - simple mode

Simple user access is not a secure method of operating HBase.

It can be used to mimic the Access Control using on a development system without having to set up Kerberos. This is done using SASL.

This method is not used to prevent malicious or hacking attempts.

Refer document Additional Details.txt for configuration details.

Hbase security - securing ZooKeeper access

- Secure HBase requires secure ZooKeeper and HDFS so that users cannot access and/or modify the metadata and data from under HBase.
- ZooKeeper has a pluggable authentication mechanism to enable access from clients using different methods.
- ZooKeeper allows authenticated and unauthenticated clients at the same time.
- HBase daemons authenticate to ZooKeeper via SASL and kerberos.
- HBase sets up the znode ACLs so that only the HBase user and the configured hbase superuser (hbase.superuser) can access and modify the data.
- In cases where ZooKeeper is used for service discovery or sharing state with the client, the znodes created by HBase will also allow anyone (regardless of authentication) to read these znodes (clusterId, master address, meta location, etc), but only the HBase user can modify them.

Hbase security - securing File system/HDFS access

- All of the data under management is kept under the root directory in the file system (hbase.rootdir).
- Access to the data and WAL files in the filesystem should be restricted.
- HBase assumes the filesystem used (HDFS or other) enforces permissions hierarchically. If sufficient protection from the file system (both authorization and authentication) is not provided, HBase level authorization control (ACLs, visibility labels, etc) is meaningless since the user can always access the data from the file system.
- HBase enforces the posix-like permissions 700 (rwx------) to its root directory. The default setting can be changed by configuring hbase.rootdir.perms in hbase-site.xml. You can, also, manually set the permissions for the root directory if needed. Using HDFS. What would be the command?

Hbase security - Tag and Visibility labels

- A tag is a piece of metadata which is part of a cell, separate from the key, value, and version. Tags are an implementation detail which provides a foundation for other security-related features such as cell-level ACLs and visibility labels.
- Every cell can have zero or more tags. Every tag has a type and the actual tag byte array.
- Visibility labels control can be used to only permit users associated with a given label to access cells with that label.
- For instance, you might label a cell top-secret, and only grant access to that label to the managers group.
- Visibility labels are implemented using Tags.
- If a user’s labels do not match a cell’s label or expression, the user is denied access to the cell.
- A user adds visibility expressions to a cell during a Put operation.

Hbase security - Access Control Labels

HBase has a simpler security model than relational databases, especially in terms of client operations. No distinction is made between an insert (new record) and update (of existing record), for example, as both collapse down into a Put.

HBase access levels are granted independently of each other and allow for different types of operations at a given scope.

- *Read (R)* - can read data at the given scope
- *Write (W)* - can write data at the given scope
- *Execute (X)* - can execute coprocessor endpoints at the given scope
- *Create (C)* - can create tables or drop tables (even those they did not create) at the given scope
- *Admin (A)* - can perform cluster operations such as balancing the cluster or assigning regions at the given scope

Example usage: grant 'user', 'RWXCA', 'TABLE', 'CF', 'CQ'