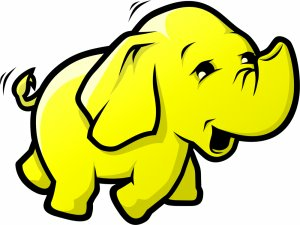
BIGDATA

HADOOP



Index

S.No Topic Page No

1. BigData 4

2. Hadoop Introduction 5

3. Requirements for New Approach 6

4. Hadoop’s History 6

5. HDFS 9

6. Local file System commands 13

7. Hdfs Commands 15

8. MapReduce 20

9. WordCount Job 37

Differences b/w Old and New Api 45

10. Streaming API 48

11. IDE (Eclipse) 50

12. IdentityReducer 54

13. IdentityMapper 57

14. Combiner 58

15. Partitioner 66

16. Accessing HDFS Programatically 71

17. Common MapReduce Algorithm 72

Sorting Algorithm 72

Searching Algorithm 74

Indexing Algorithm 77

WordCo-occurrence 80

18. Integrating Hadoop into the Workflow 84

19. Sqoop 85

20. Flume 93

21. Hive 95

22. Pig 112

23. HBase 130

24. Oozie 140

25. Software Installation Process 146

26. Single Node Cluster Configuration 160

27. Multi Node Cluster Configuration 169

28. Hadoop Interview Tips 179

BigData

BigData:

Big Data is a term that represents data sets whose size is beyond the capacity of commonly used software tools to manage and process the data within a tolerable elapsed time. Big data sizes are a constantly moving target, as of 2012 ranging from a few dozen terabytes to many petabytes of data in a single data set.

It is the term for a collection of data sets, so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications.

Bigdata is term which defines three characterstics

1. **Volume**
2. **Velocity**
3. **veriety**

Already we have RDBMS to store and process structured data. But oflate we have been getting data in form of videos, images and text. This data is called as unstructured data and semistructured data. It is difficult to efficiently store and process these data using RDBMS.

So definitely we have to find an alternative way to store and to process this type of unstructured and semistructured data.

**HADOOP** is one of the technologies to efficiently store and to process large set of data. This **HADOOP** is entirely different from Traditional distributed file system. It can overcome all the problems exits in the traditional distributed systems.

**HADOOP** is an **opensource** **framework** written in **java** for **stroing** data in a **distributed** file system and **processing** the data in **parallel** manner across cluster of commodity nodes.

HADOOP

**Introduction:**

**The Motivation for Hadoop**

* What problems exist with ‘traditional’ large-scale computing systems
* What requirements an alternative approach should have
* How Hadoop addresses those requirements

**Problems with Traditional Large-Scale Systems**

* Traditionally, computation has been processor-bound Relatively small amounts of data
* For decades, the primary push was to increase the computing power of a single machine
* Distributed systems evolved to allow developers to use multiple machines for a single job

**Distributed Systems: Data Storage**

* Typically, data for a distributed system is stored on a SAN
* At compute time, data is copied to the compute nodes
* Fine for relatively limited amounts of data

**Distributed Systems: Problems**

* Programming for traditional distributed systems is complex
* Data exchange requires synchronization
* Finite bandwidth is available
* Temporal dependencies are complicated
* It is difficult to deal with partial failures of the system

**The Data-Driven World**

**Modern systems have to deal with far more data than was the case in the past**

* Organizations are generating huge amounts of data
* That data has inherent value, and cannot be discarded
* **Examples:**

Facebook – over 70PB of data

EBay – over 5PB of data

**Many organizations are generating data at a rate of terabytes per day**

**Getting the data to the processors becomes the bottleneck**

**Requirements for a New Approach**

**Partial Failure Support**

**The system must support partial failure**

* Failure of a component should result in a graceful degradation of application performance. Not complete failure of the entire system.

**Data Recoverability**

* If a component of the system fails, its workload should be assumed by still-functioning units in the system
* Failure should not result in the loss of any data

**Component Recovery**

* If a component of the system fails and then recovers, it should be able to rejoin the system.
* Without requiring a full restart of the entire system

**Consistency**

* Component failures during execution of a job should not affect the outcome of the job

**Scalability**

* Adding load to the system should result in a graceful decline in performance of individual jobs Not failure of the system
* Increasing resources should support a proportional increase in load capacity.

**Hadoop’s History**

* Hadoop is based on work done by Google in the late 1990s/early 2000s.
* Specifically, on papers describing the Google File System (GFS) published in 2003, and MapReduce published in 2004.
* This work takes a radical new approach to the problems of Distributed computing so that it meets all the requirements of reliability and availability.
* **This core concept is distributing the data as it is initially stored in the system.**
* Individual nodes can work on data local to those nodes so data cannot be transmitted over the network.
* Developers need not to worry about network programming, temporal dependencies or low level infrastructure.
* Nodes can talk to each other as little as possible. Developers should not write code which communicates between nodes.
* Data spread among the machines in advance so that computation happens where the data is stored, wherever possible.
* Data is replicated multiple times on the system for increasing availability and reliability.
* When data is loaded into the system, it splits the input file into ‘blocks ‘, typically 64MB or 128MB.
* Map tasks generally work on relatively small portions of data that is typically a single block.
* A master program allocates work to nodes such that a map task will work on a block of data stored locally on that node whenever possible.
* Nodes work in parallel to each of their own part of the dataset.
* If a node fails, the master will detect that failure and re-assigns the work to some other node on the system.
* Restarting a task does not require communication with nodes working on other portions of the data.
* If failed node restarts, it is automatically add back to the system and will be assigned with new tasks.

**Q) What is speculative execution?**

If a node appears to be running slowly, the master node can redundantly execute another instance of the same task and first output will be taken. This process is called as **speculative execution.**

Hadoop consists of two core components

1. HDFS

2. MapReduce

There are many other projects based around core concepts of Hadoop. All these projects are called as **Hadoop Ecosystem**.

Hadoop Ecosystem has

Pig

Hive

Flume

Sqoop

Oozie

and so on…

A set of machines running HDFS and MapReduce is known as **hadoop cluster** and Individual machines are known as **nodes.**

A cluster can have as few as one node or as many as several thousands of nodes.

As the no of nodes are increased performance will be increased.

The other languages except java (C++, RubyOnRails, Python, Perl etc… ) that are supported by hadoop are called as **HADOOP Streaming.**

**HADOOP S/W AND H/W Requirements:**

* Hadoop useally runs on opensource os’s (like linux, ubantu etc)
  + Centos/RHEL is mostly used in production
* If we have Windows it require virtualization s/w for running other os on windows
  + Vm player/Vm workstation/Virtual box
* Java is prerequisite for hadoop installation

HDFS

* **HDFS** is a distributed file system designed for storing very large files with streaming data access patterns, running on cluster of commodity hardware.
* HDFS is a logical file system across all the nodes local file system; it provides special capabilities to handle the storage of bigdata efficiently.
* We are giving files to HDFS, and it will devide the file into no of blocks of managible size (either 64MB or 128MB based on configuration).
* These blocks will be replicated three times (default can be configurable) and stored in local file system as a separate file.
* Blocks are replicated across multiple machines, known as DataNodes. DataNode is a slave machine in hadoop cluster running the datanode deamon (a process continuously running).
* A master node(high end configurations like dual power supply,dual n/w cards etc..) called the NameNode (a node which is running namenode deamon) keeps track of which blocks make up a file, and where those blocks are located, known as the metadata.
* The NameNode keeps track of the file metadata—which files are in the system and how each file is broken down into blocks. The DataNodes provide backup store of the blocks and constantly report to the NameNode to keep the metadata current.
* HDFS - Blocks
* File Blocks

– 64MB (default), 128MB (recommended) – compare to 4KB in UNIX

– Behind the scenes, 1 HDFS block is supported by multiple operating system (OS) blocks

**HDFS is optimized for**

**1. Large files**: HDFS file system is a special filesystem specially designed to store large size files, generally 100MB or more.

**2. Streaming data access** : once we are writing data into HDFS , we can read the data for any number of time but we should not change the content of the file…

**3. Commodity Hardware:** HDFS is using cluster of commodity h/w nodes to store the data.

**HDFS is not optimized for**

**1. Lots of small files:** HDFS is not designed for storing the lots of small file, because for each file even it is a small file or large file it takes some amount of NameNode RAM.

Ex: 100MB file 5kb namenode ram

100 1MB files 100\*5kb namenode ram

2. **Write once and Read Many times** pattern.

**3. Low-latency data access pattern: Applications** that require low-latency access to data in that tens of milliseconds range but it will not work well with HDFS.

HDFS is optimized for delivering a high through put of data so the result will not come in seconds.

Data split into blocks and distributed across multiple nodes in the cluster.

Each block is typically 64MB or 128MB in size.

HDFS Block 64mb

64MB

Os blocks 4kb or 8kb

There is no memory wastage in HDFS for example for storing 100MB file, HDFS will take 2 blocks one is 64 MB and another is 36 MB.For storing 64 it takes how many os blocks are required it will take that many and for 36 mb how many os level blocks required it will take that many.

Each block is replicated three (configurable) times. Replicas are stored on different nodes this ensures both reliability and availability.

It provides redundant storage for massive amount of data using cheap and unreliable computers.

Files in HDFS are **‘write once’**.

Different blocks from same file will be stored on different machines. This provides for efficient MapReduce processing.

There are five daemons in HDFS

* + NameNode
  + SecondaryNameNode
  + JobTracker
  + DataNode
  + TaskTracker

**NameNode:**

NameNode keeps track of name of each file and its permissions and its blocks which make up a file and where those blocks are located. These details of data are known as **Metadata**.

**Example:**

**NameNode** holds metadata for the two files (Foo.txt, Bar.txt)

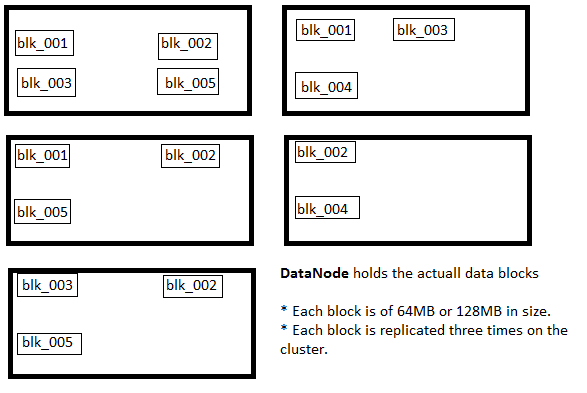
**NameNode:**



**NameNode default port no is 8020 and web ui address is 50070**

**HDFS Default location is user/<username>**

**DataNodes:**



The **NameNode** Daemon must be running at all times. If the NameNode gets stopped, the cluster becomes inaccessible.

System administrator will take care to ensure that the NameNode hardware is reliable.

**SecondaryNameNode**:

A separate daemon known as SecondaryNameNode takes care of some housekeeping task for the NameNode. But this SecondaryNameNode is not a backup NameNode. But it is a backup for metadata of NameNode.

Although files are splitted into 64MB or 128MB blocks, if a file is smaller than this the full 64MB or 128MB will not be used.

Blocks are stored as standard files on DataNodes, in a set of directories specified in Hadoop configuration file.

Without metadata on the NameNode, there is no way to access the files in the HDFS.

When client application wants to read a file,

It communicates with the NameNode to determine which blocks makeup the file and which DataNodes those blocks reside on.

It then communicates directly with the DataNodes to read the data.

**Accessing HDFS**:

Application can read and write HDFS files directly via the Java API.

Typically, files are created on the local file system and must be moved into HDFS.

Like wise files stored in HDFS may need to be moved to a machine’s local filesystem.

Access to HDFS from the command line is achieved with the “**hadoop fs**” command.

**Local File System commands:**

**pwd :** Present working directory.

**ls :** To list files and folders of the directory.

**ls -l** : to long list all files and directories

If any thing is starting with ‘d’ – it’s a directory

If any thing is starting with ‘-‘ – it’s s file

**ls -a** : to list all hidden files and directories

If any thing is starting with ‘. ’ – it’s a hidden file.

**mkdir :** To create a directory

syn: mkdir <directoryname>

**Cd :** To change the directory

**cd .. :** To go to parent directory

**cd ~ :** To go to home directory

**cd / :** To go root directory

**creating files:**

files can be created in three ways ,

1. cat
2. vi
3. touch

**1.cat:**

cat can be used in three ways

1. for creating file
2. for appending text
3. for displaying file content

$ cat > filename ---------- to create a file

After creating file we have to save and exit . To save and exit the file we have to use “ctrl+d” . If we want to exit the file without saving we have to use “ctrl+c”.

$ cat >> filename ---------- to add content to the existing file

$ cat filename ---------- to view the content of the file

**2.vi:**

Vi editor can be used in three ways

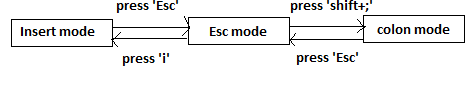
1. for creating the file
2. for appending text or for modifying the file
3. for displaying the file content

vi editor is working with three modes.

1. Esc mode (press ‘Esc’)
2. Insert mode (press ‘i’)
3. Colon mode (press ‘shift+;’)

By default “vi” editor will be in ‘esc mode’ . If we want to insert text to the file we have to enter into insert mode by pressing ‘i’ from keyboard. After giving text if we want save and exit the file we must enter into colon mode. But there is no way of moving from insert mode to colon mode, it must be through “Esc” mode only.

The following dig will be brief idea about using the “vi” editor.



**3.touch :**

This command can be used in only one way , for creating a file. After creating the file we can give text in three ways

1.either by using cat>>

2.by using vi editor

3. by using text editor ( manual creation through text editor)

**rm :** To remove the file

**rm –r :** for recursively removing all files and sub directories from a specified directory.

**rm –rf :** for recursively and forcibly removing files and sub directories from a specified directory.

**mv :** To move file/directory from one location to another location

**cp :** To copy the file/directory from one location to another location

**Hadoop distributed filesystem commands**

* To copy file foo.txt from local file system to the HDFS we can use either **“-put”**

Or “**-copyFromLocal”** command.

**$ hadoop fs –copyFromLocal foo.txt foo**

* To get file from HDFS to local file system we can use either **“-get”** or **“-copyToLocal”**

**$ hadoop fs –copyToLocal foo/foo.txt file**

* To get a directory listing in the user’s home directory in the HDFS

**$ hadoop fs –ls**

* To get a directory listing of the HDFS root directory.

**$ hadoop fs –ls /**

* To get the contents of the file in HDFS /user/training/foo.txt

**$ hadoop fs –cat /user/training/foo.txt**

* To Create a directory in HDFS.

**$ hadoop fs –mkdir <directoryname>**

* To delete the directory from HDFS

**$ hadoop fs –rmr <directory path>**

* To remove file from HDFS

**$ hadoop fs –rm <file path>**

* To copy files from one location to another location

**$ hadoop fs –cp <src> <dest>**

* To move files from one location to another location

**$ hadoop fs –mv <src> <dest>**

* To find out disk free space

**$ hadoop fs –df**

* To find out disk usage

**$ hadoop fs –du <src>**

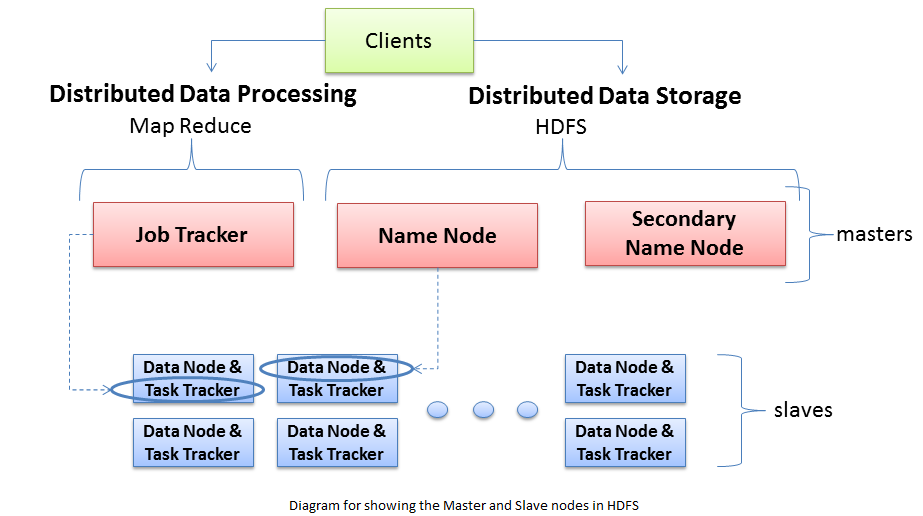
* To remove data from Trash

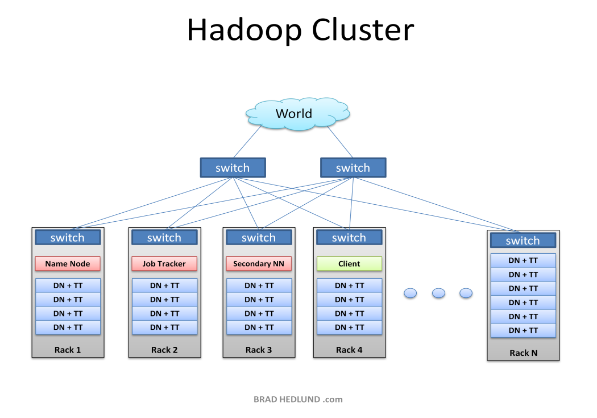
**$ hadoop fs –expunz**

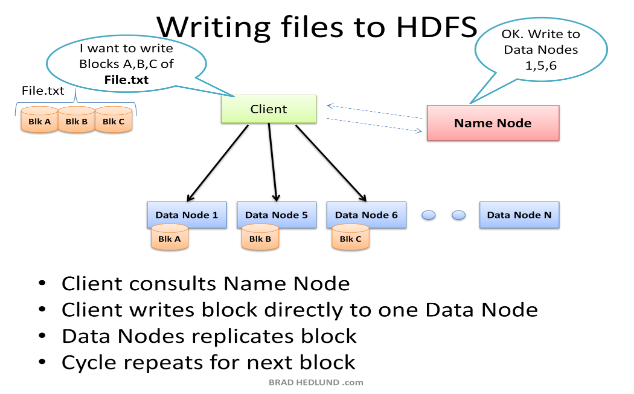
* To create empty file

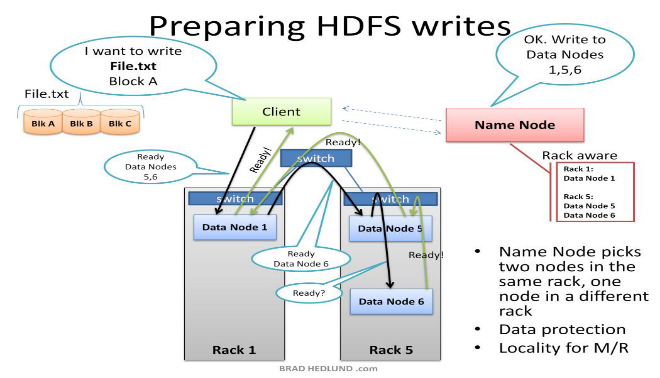
**$ hadoop fs –touchz**

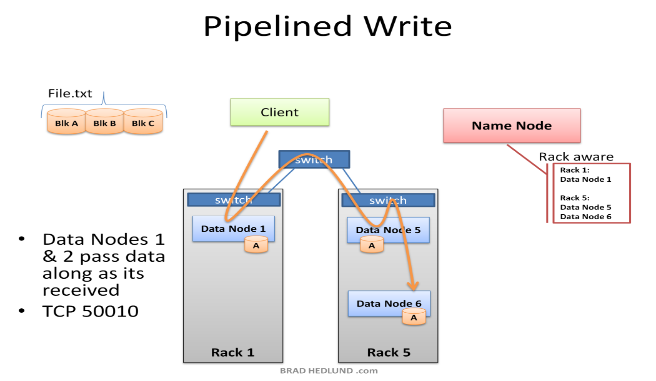
**HDFS Daemons:**

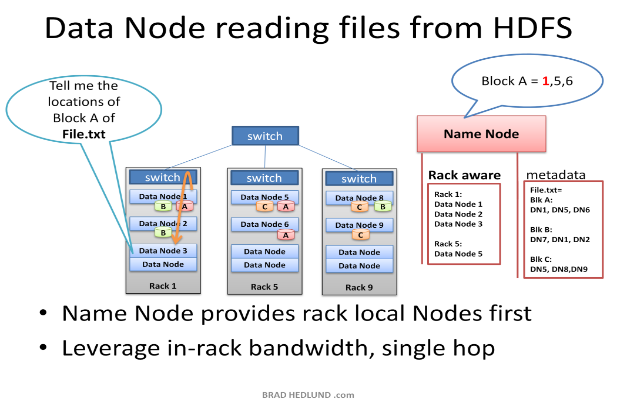












MapReduce

**MapReduce** is a massive parallel processing technique for processing data which is distributed on a commodity cluster. MapReduce is a method for distributing a task across multiple nodes.

Each node is processing data, stored on local to those nodes whenever possible.

Hadoop can run MapReduce programs written in various languages.Generally we are writing the mapreduce programs in java because whole hadoop has developed in java.The other languages which support hadoop like c++, ruby on rails, python etc are called as hadoop streaming.

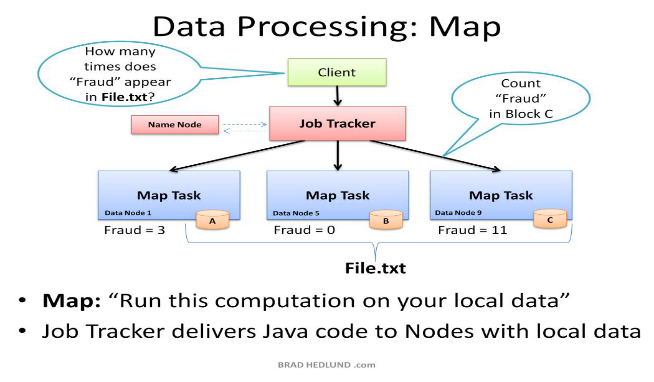
MapReduce consists of two phases.

* Map
* Reduce

**Features of MapReduce:**

* Automatic parallelization and distribution.
* Fault tolerance , Status and monitoring the tools
* It is clear abstraction for the programmers.
* MapReduce programs are usually written in java but it can be written in any other scripting language using Hadoop Streaming API.
* MapReduce abstracts all the “housekeeping” away from the developer.Developer can concentrate simply on writing the Map and Reduce functions.
* MapReduce default port number is 8021 and web ui is 50030

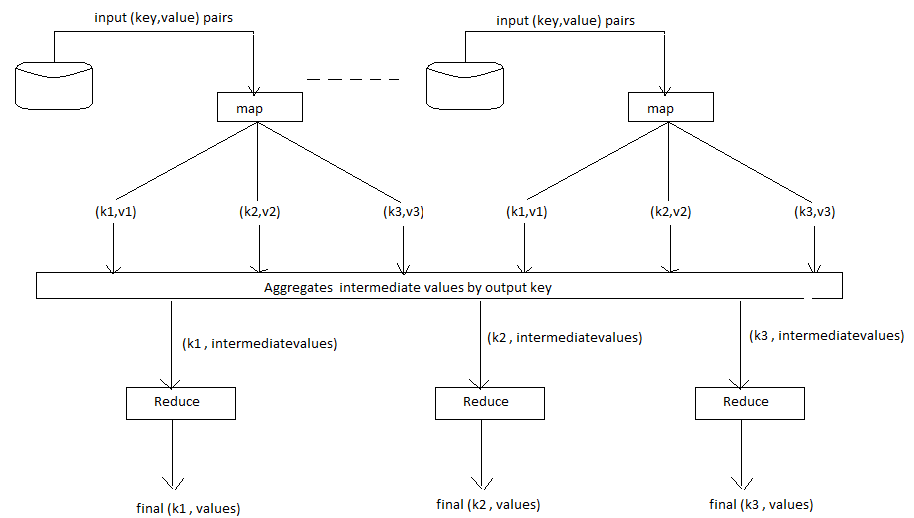
*JobTracker*

**

MapReduce jobs are controlled by a software daemon known as the ***JobTracker****.*

The JobTracker resides on a ‘masternode’.

* Client submits MapReduce jobs to the JobTracker.
* The JobTracker assigns Map and Reduce tasks to other nodes on the cluster.
* These nodes each run a software daemon known as the TaskTracker.
* The TaskTracker is resoponsible for actuall instantiating the Map or Reduce tasks and reporting progress back to the JobTracker.
* If a task attempt fails, another will be started by the JobTracker.



**Mapper:**

Hadoop attempts to ensure that mappers run on nodes which hold their portion of the data locally, to avoid network traffic.

One mapper can work on one individual split of the input file.

The mapper reads data in the form of (key, value) pairs.

It outputs either zero or more (key, value) pairs.

Map (input\_key, input\_value) ----------->(intermediate\_key,intermediate\_value) list

The Mapper may use or completely ignore the input key.

For example if a standard pattern is to read a line of a line starts.

The value is the contents of the line itself.

So typically the key is considered irrelevant.

If the mapper writes anything out, the output must be in the form (key, value) pairs.

Mappers output is called as intermediate data and this will be stored on local file system not on HDFS.

*Example Mapper:* **Uppercase Mapper**

* Change the output into uppercase letters

Let map(k,v) = emit(k.toUpper(),v.toUpper())

(‘foo’,’bar’) --------------->(‘FOO’,’BAR’)

(‘foo’,’other’)--------------> (‘FOO’,’OTHER’)

*Example Mapper:* **Explode Mapper**

* Output each input character separately.

Let map(k,v) = foreach character in v : emit(k,v)

(‘foo’,’bar’) ------------> (‘foo’ ,’b’) , (‘foo’, ‘a’) , (‘foo’, ‘r’)

(‘bar’,’other’) -----------> (‘bar’,’o’) , (‘bar’,’t’) , (‘bar’,’h’) , (‘bar’,’e’) , (‘bar’,’r’)

*Example Mapper:* **Changing keyspaces**

The key output by the mapper does not need to be the identical to the input key.

Output the word length as the key

Let map(k,v) = emit (v.length() , v)

(‘foo’,’bar’) ----------> (3, ‘bar’)

(‘baz’, ‘other’) ---------> (5, ‘other’)

**A job is a ‘full program’**

* A complete execution of Mappers and Reducers over a dataset
* A task is the execution of a single Mapper or Reducer over a slice of data
* A task attempt is a particular instance of an attempt to execute a task
* There will be at least as many task attempts as there are tasks
* If a task attempt fails, another will be started by the JobTracker is called as ***Speculative execution***.

**Reducer:**

After the Map phase is over, all the intermediate values for a given intermediate key are combined together into a list.This list is given to a Reducer.

There may be a single Reducer, or multiple Reducers. This is specified as part of the job configuration.

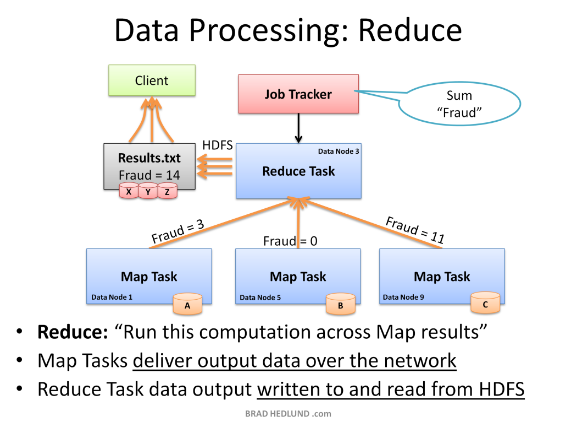
All values associated with a particular intermediate key are guaranteed to go to the same Reducer.

The intermediate keys, and their value lists, are passed to the Reducer in sorted key order .This step is known as the ‘shuffle and sort.

The Reducer outputs zero or more final (key, value) pairs and these will be written to HDFS.

The Reducer usually emits a single (key, value) pair for each input key.

The reducer output is final and it will be stored in HDFS and after that intermediate data will be deleted.

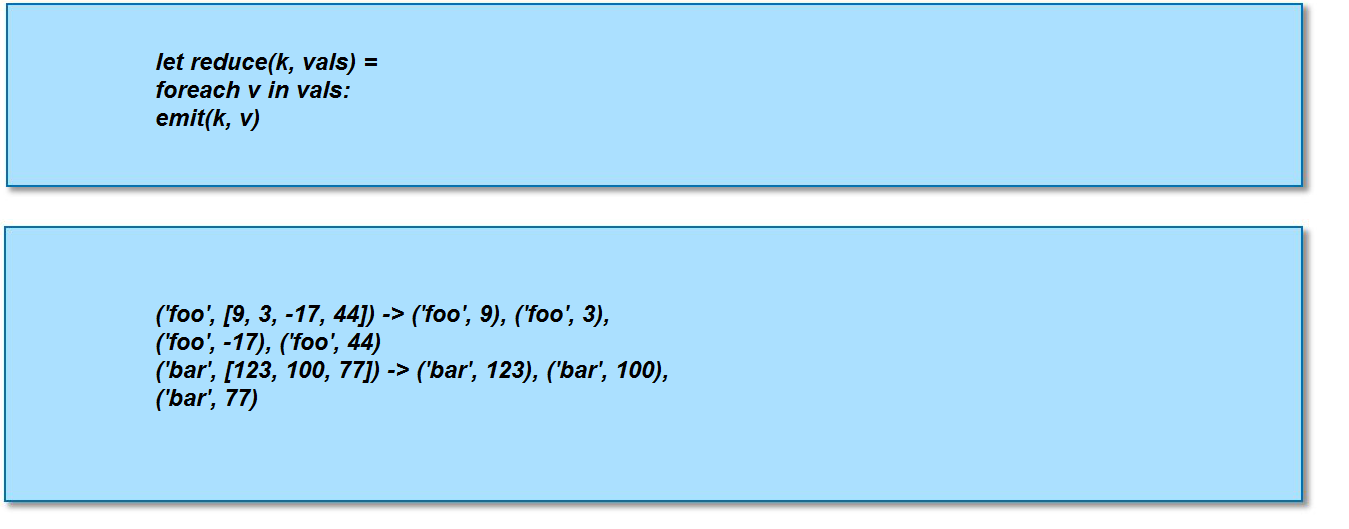


**Example Reducer**: Sum Reducer

Add up all the values associated with each intermediate key



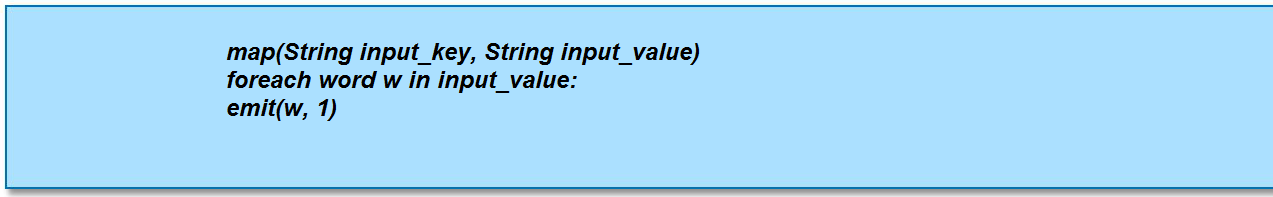
**Example Reducer: Identity Reducer**



**MapReduce Example: Word Count**

Count the number of occurrences of each word in a large amount of input data

– This is the ‘hello world’ of MapReduce programming

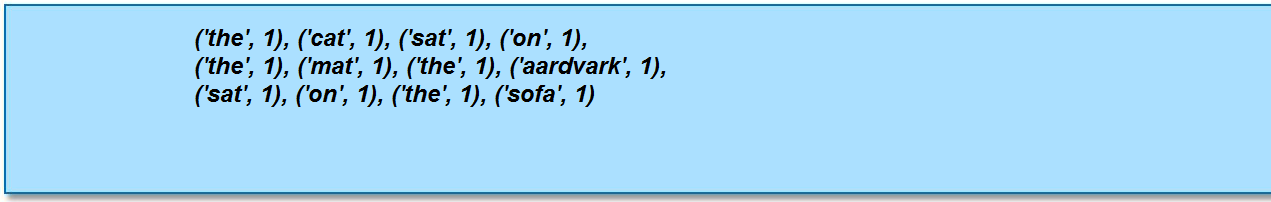




**Input to the Mapper:**



**Output from the Mapper:**



**Intermediate data sent to the Reducer:**

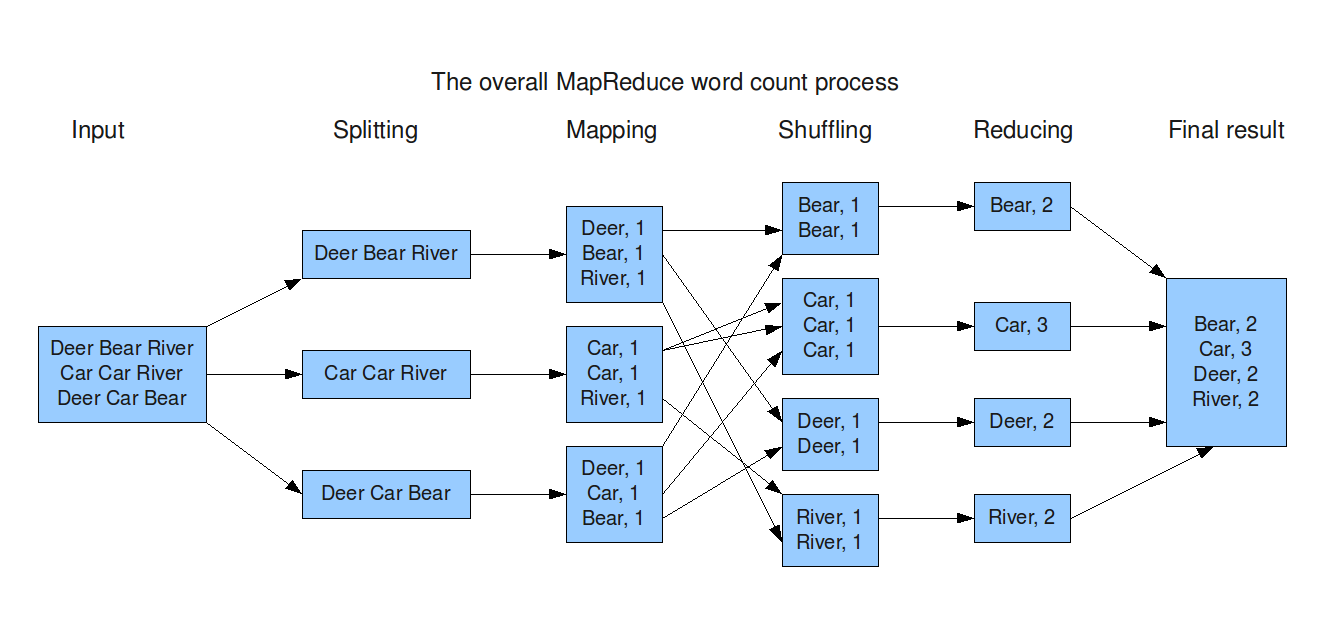


**Intermediate data sent to the Reducer:**



**Final Reducer output:** 

**MapReduce Flow Diagram:**

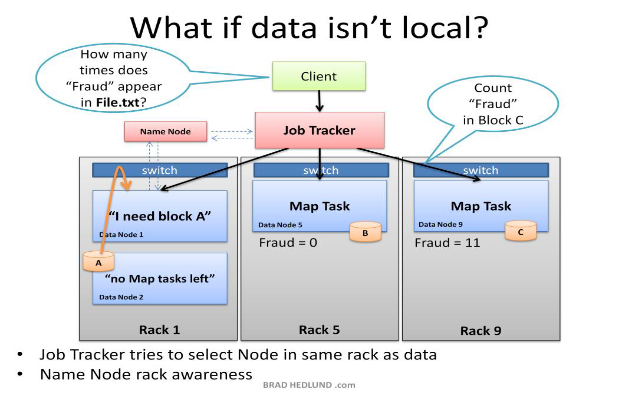
****

**MapReduce: Data Localization**

* Whenever possible, Hadoop will attempt to ensure that a Map task on a node is working on a block of data stored locally on that node via HDFS.
* If this is not possible, the Map task will have to transfer the data across the network as it processes that data.
* Once the Map tasks have finished, data is then transferred across the network to the Reducers Although the Reducers may run on the same physical machines as the Map tasks. there is no concept of data locality for the Reducers
* All Mappers will, in general, have to communicate with all Reducers

**MapReduce Internal Concept:**

* Client submits the job to job tracker.
* JobTracker contact namennode for the metadata related to the job i/p file or dir.
* Based on the information getting from the namenode JobTracker divides the file into logical splits called input splits, each input split is processed by one map task.
* Input split size is based on the various parameters like if job i/p is in the form of files it will be based on the size of file and where the file stored.
* Once map tasks are assigned to task trackers (each task tracker has fixed no of map and reduce slots to prform the tasks given by JobTracker), task tracker intiates a separate jvm for each task.
* Based on InputFormat, its respective RecordReader class will divide the split into no of records of each record is a one (k, v) pair.
* The RecordReader will give record by record to the Map task after processing each record.
* Based on the map logic written we may get zero or more (k, v) pairs for each i/p (k, v) pair to the map task.
* After all map tasks are completed the intermediate data(or intermediate o/p stored temporarly in local fs) will be copied to reduce task.
* As hadoop tries to process the data where ever it is stored there is data local concept for map tasks ,but reduce task need to copy the data from all the map tasks ,there is no data local concept for reduce task.
* Default no of reduce tasks in hadoop is one, we can make it as zero if there is no requirement of reduce phase or we can set variable no of reducers based on the requirement.
* Between map and reduce phase there is a default sort and shuffle phase, so the data to the reducer will be in sorting order and shuffle means hadoop framework will send all common keys as a single (k, list of vales associated with same key).
* Once reduce task is completed intermediate data will be deleted, and final o/p from reducer will be stored in HDFS.
* Based on no of reducere, no of output files will be genereated in the output directory.

****

**MapReduce: Is Shuffle and Sort a Bottleneck?**

* It appears that the shuffle and sort phase is a bottleneck
* The reduce method in the Reducers cannot start until all Mappers finishes their task.
* In practice, Hadoop will start to transfer data from Mappers to Reducers as the Mappers finish their work. This mitigates against a huge amount of data transfer starting assoon as the last Mapper finishes

**MapReduce: Is a Slow Mapper a Bottleneck?**

* It is possible for one Map task to run more slowly than the others due to faulty hardware, or just a very slow machine. It would appear that this would create a bottleneck.
* The reduce method in the Reducer cannot start until every Mapper has finished
* Hadoop uses ***speculative execution***to mitigate against this
* If a Mapper appears to be running significantly more slowly than the others, a new instance of the Mapper will be started on another machine, operating on the same data.
* The results of the first Mapper to finish will be used. Hadoop will kill the Mapper which is still running.

**HADOOP Configuration files:** There are three important configurations files in HADOOP

1. core-site.xml

2. hdfs-site.xml

3. mapred-site.xml

Hadoop-env.sh file is for placing path and classpaths of java, hadoop etc components.

**HADOOP DataTypes:**

In hadoop all datatypes are hadoop classes; those are called as HADOOP BOX classes

IntWritable

LongWritable

DoubleWritable

Text etc….

**HADOOP Input Formats:**

Based on how the data stored in file we have several input formats available in HADOOP

If input data is in the form of files **FileInputFormat** is the base class, and based on the how data stored in file we will choose the InputFormat

1. TextInputFormat

2. KeyValueTextInputFormat

3. SequenceFileInputFormat

4. SequenceFileAsInputFormat

**Installing a Hadoop Cluster**

* Cluster installation is usually performed by the hadoop administrator, and is outside the scope of this course.
* Hadoop Administrators specifically aimed at those responsible for maintaining Hadoop clusters.
* Three types of cluster configurations available
  + Local Job Runner(no HDFS,MapReduce only local file system and java)
  + Pseudo Distributed(All five deamons running in single machine)
  + Fully Distributed(separation of machines for different deamons)
* However, it’s very useful to understand how the component parts of the Hadoop cluster work together
* Typically, a developer will configure their machine to run in *pseudo-distributed mode*

– This effectively creates a single-machine cluster

– All five Hadoop daemons are running on the same machine

– Very useful for testing code before it is deployed to the real cluster

* Easiest way to download and install Hadoop, either for a full cluster or in pseudo-distributed mode, is by using Cloudera’s Distribution including Apache Hadoop (CDH)

Hadoop is comprised of five separate *daemons*

**NameNode**

* Holds the metadata for HDFS

**Secondary NameNode**

* Performs housekeeping functions for the NameNode
* Is not a backup or hot standby for the NameNode!

**DataNode**

* Stores actual HDFS data blocks

**JobTracker**

* Manages MapReduce jobs, distributes individual tasks to machines running the…

**TaskTracker**

* Instantiates and monitors individual Map and Reduce tasks

Each daemon runs in its own Java Virtual Machine (JVM)

No node on a real cluster will run all five daemons

– Although this is technically possible

We can consider nodes to be in two different categories:

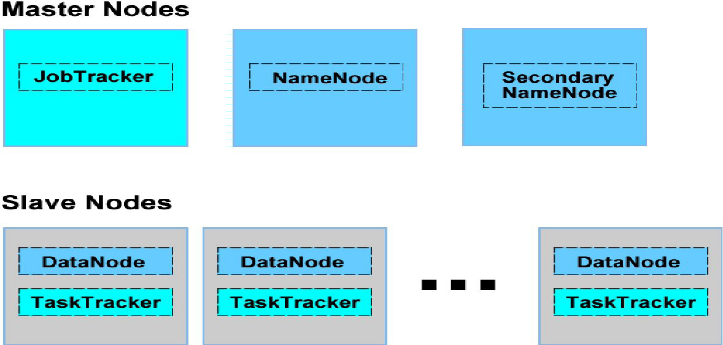
**Master Nodes**

* Run the NameNode, Secondary NameNode, JobTracker daemons
* Only one of each of these daemons runs on the cluster

**Slave Nodes**

* Run the DataNode and TaskTracker daemons
* A slave node will run both of these daemons

**Basic Cluster Configuration:**

****

* On very small clusters, the NameNode, JobTracker and Secondary NameNode can all reside on a single machine
* It is typical to put them on separate machines as the cluster grows beyond 20-30 nodes
* Each dotted box in the above diagram represents a separate Java Virtual Machine (JVM)

**Submitting a Job:**

* When a client submits a job, its configuration information is packaged into an XML file.
* This file, along with the .jar file containing the actual program code, is handed to the JobTracker.
* The JobTracker then parcels out individual tasks to TaskTracker nodes.
* When a TaskTracker receives a request to run a task, it instantiates a separate JVM for that task.
* TaskTracker nodes can be configured to run multiple tasks at the same time if the node has enough processing power and memory.
* The intermediate data is held on the TaskTracker’s local disk.
* As Reducers start up, the intermediate data is distributed across the network to the Reducers.
* Reducers write their final output to HDFS.
* Once the job has completed, the TaskTracker can delete the intermediate data from its local disk.

**Note**:

The intermediate data will not be deleted until entire job completes.

**A Sample MapReduce Program**

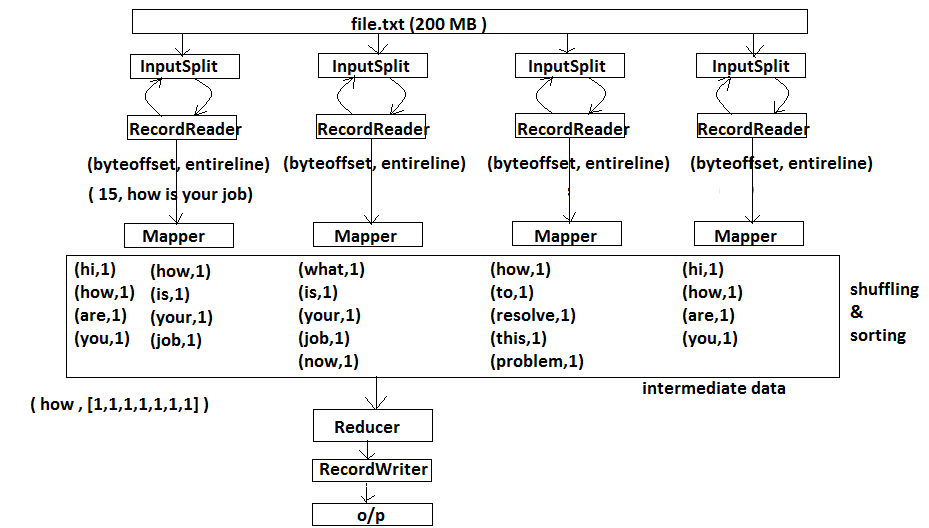
**Job Name: Wordcount**

**Job Description:** Counting the no of occurrences of each word in a input(input may be a file or list of files or a directory or list of dirs etc)

* we will examine the code for WordCount.This will demonstrate the Hadoop API
* We will also investigate Hadoop Streaming, allow you to write MapReduce programs in (virtually) any language.
* You will always create at least a **Driver, Mapper, and Reducer**

**The MapReduce Flow:**

****

****

* Here hdfs file will be splitted into number of input splits. For each input split there will be given one mapper.
* How Map interface is dealing with (key,value) pair in Collection Framework, in the same way ,hadoop is also dealing mapper and reducer with (key,value) pairs.
* So each mapper should get (key, value) pair from its corresponding input split.
* Here input is file but map (key, value) accepts only the data in the form of (key, value) pair. So to present the data in the split in the (key, value) pairs we have for each input there is an InputForamt and based on the InputFormat for each InputFormat ther is a RecordReader. This will divide split into no of records of each record is a one (key , value) pair.The RecordReader will submit one record at a time to mapper once mapper process that record it will submit another next (key , value) pair (next record).
* Depending on the logic what we are writing in the Mapper code , this mapper input (key,value) pair will be processed and will be given as mapper output (key,value) pair.
* This mapper output is called as an intermediate data.
* This intermediate data will be shuffled and sorted and will be given as an input (key, value) pair to the reducer.

**Shuffle and Sort:**

**Shuffling** is a phase on intermediate data, to group individual objects into single entity associated to single identical key.

**Sorting** is also a phase on intermediate data to decide its order depending on the key in (key, value) pair.

**Our MapReduce Program: WordCount**

To investigate the API, we will dissect the WordCount program as we have seen in the page no:

This consists of three portions

* The driver code

Code that runs on the client to configure and submit the job

* The Mapper code

Code that runs Mapper code

* The Reducer code

Code that runs Reducer code

Driver Code: In drivercode we need to specify the following things

* Input for the job(InputFormat and HDFS location of input)
* Where to store final o/p
* JobConf class for specifiying job specific configurations
* Mapper class name
* Map o/p k,v type
* Reducer class name
* Reduce o/p key value types
* JobClient to submit job

Before we look at the code, we need to cover some basic Hadoop API concepts.

**Getting Data to the Mapper:**

The data passed to the Mapper is specified by an ***InputFormat.***

* Specified in the driver code
* Defines the location of the input data, a file or directory, for example.
* Determines how to split the input data into *input splits*
* Each Mapper deals with a single input split
* InputFormat is a factory for RecordReader objects to extract (key, value) pairs from the input file.

**Some Standard InputFormats**

**FileInputFormat**

* This is the base class used for all file-based InputFormats

**TextInputFormat**

* This is the default InputFormat class for any text related file.
* Key is the byte offset within the file of that line
* Treats each \n-terminated line of a file as a value

**KeyValueTextInputFormat**

* Maps \n-terminated lines as ‘key SEP value’
* By default, separator is a tab

**SequenceFileInputFormat**

* Binary file of (key, value) pairs with some additional metadata

**SequenceFileAsTextInputFormat**

* Similar to **SequenceFileInputFormat**, but maps (key.toString(), value.toString())

**Keys and Values are Objects**

* Keys and values in Hadoop are Objects
* Values are objects which implements Writable interface
* Keys are objects which implements WritableComparable interface

In Java, for every primitive data type we have corresponding Wrapper classes. For example,

int --------> Integer

float --------> Float

double --------> Double

in the same way we have “box classes” for all data types in hadoop

**Hadoop defines its own ‘box classes’ for all data types as shown below**

– IntWritable for int

– LongWritable for long

– FloatWritable for float

– DoubleWritable for double

– Text for string

– Etc.

**What is Writable?**

* Writable is an interface which makes serialization quick and easy for Hadoop.
* Any value’s type must implement Writable interface.

**What is WritableComparable?**

* A WritableComaparable is an interface, which is Writable as well as Comparable.
* Two WritableComparables can be compared against each other to determine their order.
* Keys must be WritableComparables because they are passed to the Reducer in sorting order.

**Note:**

Despite their names, all Hadoop box classes implement both Writable and WritableComparable interfaces. For example Intwritable is actually a WritableComparable.

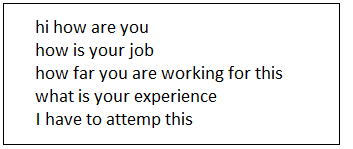
Hello World Job

WordCountJob

**Step 1:** Prepare some input for the wordcount job

Create a file

**$cat > file**

**Ctrl + d**

**Step 2:**

Put this file to Hdfs

**$hadoop fs –copyFromLocal file File**

**Step 3:**

Write an mpreduce application programs to process the above file

**The Driver Code:**

**Introduction:**

* The driver code runs on the client machine
* It configures the job, then submits it to the cluster

import org.apache.hadoop.fs.**Path**;

import org.apache.hadoop.io.**IntWritable**;

import org.apache.hadoop.io.**Text**;

import org.apache.hadoop.mapred.**FileInputFormat**;

import org.apache.hadoop.mapred.**FileOutputFormat**;

import org.apache.hadoop.mapred.**JobClient**;

import org.apache.hadoop.mapred.**JobConf**;

import org.apache.hadoop.conf.**Configured**;

import org.apache.hadoop.util.**Tool**;

import org.apache.hadoop.util.**ToolRunner**;

public class ***WordCount*** extends **Configured** implements **Tool** {

public int run(String[] args) throws **Exception** {

if (args.length != 2) {

System.out.printf("please give input and output directories”);

return -1;

}

//JobConf class is used for job specific configurations.

**JobConf** conf = new **JobConf**(***WordCount***.class);

//to set the name for the job

conf.setJobName(this.getClass().getName());

//to set intput path

**FileInputFormat**.setInputPaths(conf, new **Path**(args[0]));

//to set output path

**FileOutputFormat**.setOutputPath(conf, new **Path**(args[1]));

//to set Mapper class

conf.setMapperClass(***WordMapper***.class);

//to set Reducer class

conf.setReducerClass(***SumReducer***.class);

//to set Mapper output key type

conf.setMapOutputKeyClass(**Text**.class);

//to set Mapper output value type

conf.setMapOutputValueClass(**IntWritable**.class);

//to set output key from reducer

conf.setOutputKeyClass(**Text**.class);

//to set output value from reducer

conf.setOutputValueClass(**IntWritable**.class);

//JobClient class for submitting job to JobTracker

**JobClient**.runJob(conf);

return 0;

}

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

//ToolRunner class will take the implementation class of Tool Interface to read command line argument along with the input and output directories.

int exitCode = **ToolRunner**.run(new ***WordCount***(), args);

System.exit(exitCode);

}

}

**Explaination about Driver Code:**

**The Driver: Import Statements**

import org.apache.hadoop.fs.**Path**;

import org.apache.hadoop.io.**IntWritable**;

import org.apache.hadoop.io.**Text**;

import org.apache.hadoop.mapred.**FileInputFormat**;

import org.apache.hadoop.mapred.**FileOutputFormat**;

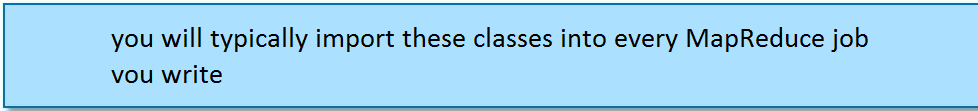
import org.apache.hadoop.mapred.**JobClient**;

import org.apache.hadoop.mapred.**JobConf**;

import org.apache.hadoop.conf.**Configured**;

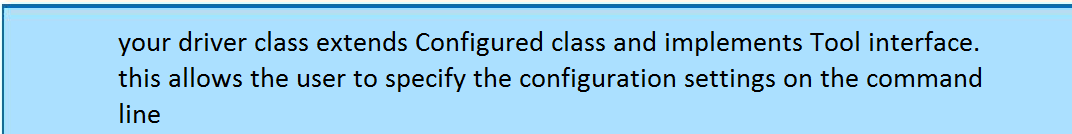
import org.apache.hadoop.util.**Tool**;

import org.apache.hadoop.util.**ToolRunner**;

****

**The Driver Class: Tool and ToolRunnner**

Public class ***WordCount*** extends **Configured** implements **Tool** {

****

**}**

Public static void main(String args[])throws Exception{

int exitCode = **ToolRunner**.run(new WordCount(),args);

****

**}**

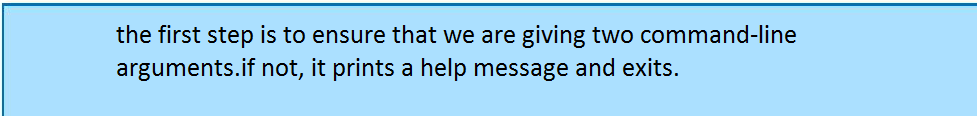
**The Job’s Invocation**

public int run(String[] args) throws **Exception** {

if (args.length != 2) {

System.out.printf("please give input and output directories”);

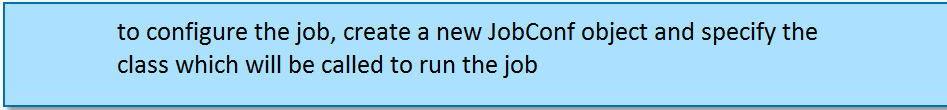
return -1;



}

**Configuring the Job With JobConf**

JobConf conf = new JobConf(WordCount.class)



**Creating a New JobConf Object**

**The JobConf class allows you to set configuration options for your MapReduce job**

– The classes to be used for your Mapper and Reducer

– The input and output directories

– Many other options

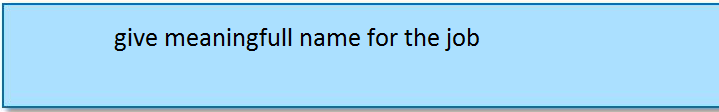
**Any options not explicitly set in your driver code will be read from your Hadoop configuration files**

– Usually located in /etc/hadoop/conf

**Any options not specified in your configuration files will receive Hadoop’s default values**

**Naming The Job**

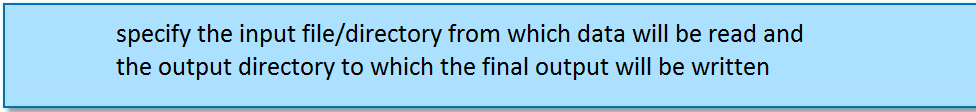
conf.setJobName(this.getClass().getName());



**Specifying Input and Output Directories:**

FileInputFormat.setIntputPaths(conf,new Path(args[0]));

FileOutputFormat.setOutputPath(conf,new Path(args[1]));

****

**Specifying the InputFormat**

The default **InputForma**t is the **TextInputFormat** and it will be used unless you specify the **InputFormat** class explicitly.

To use an InputFormat other than the default, use e.g.

conf.setInputFormat(**KeyValueTextInputFormat**.class)

or

conf.setInputFormat(**SequenceFileInputFormat**.class)

or

conf.setInputFormat(**SequenceFileAsTextInputFormat**.class)

**Determining Which Files To Read**

**By default, FileInputFormat.setInputPaths () will read all files from a specified directory and send them to Mappers**

– Exceptions: items whose names begin with a period (.) or underscore (\_)

– Globs can be specified to restrict input

– For example, /2010/\*/01/\*

**Alternatively, FileInputFormat.addInputPath () can be called multiple times, specifying a single file or directory each time**

**Specifying Final Output with OutputFormat**

**FileOutputFormat.setOutputPath () specifies the directory to which the Reducers will write their final output**

**The driver can also specify the format of the output data**

– Default is a plain text file

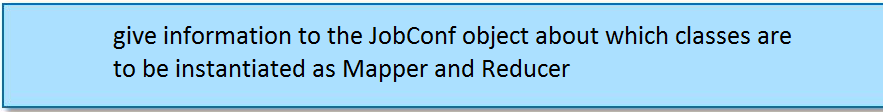
– Could be explicitly written as

conf.setOutputFormat (TextOutputFormat.class);

**Specify the Classes for Mapper and Reducer**

conf.setMapperClass (WordMapper.class);

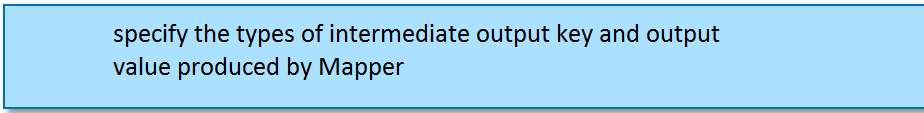
conf.setReducerClass (WordReducer.class);

****

**Specify the Intermediate Data Types**

conf.setMapOutputKeyClass (Text.class);

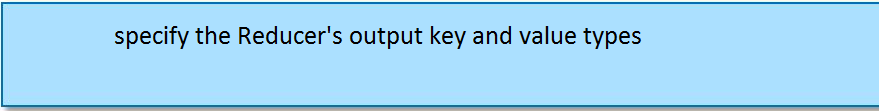
conf.setMapOutputValueClass (IntWritable.class);

****

**Specify the Final Output Data Types**

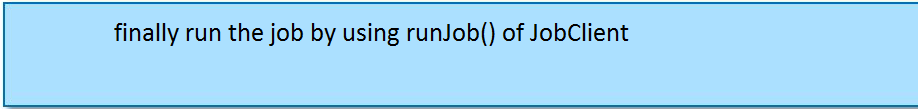
conf.setOutputKeyClass (Text.class);

conf.setOutputValueClass (IntWritable.class);

****

**Running the Job:**

JobClient.runJob (conf);

****

**There are two ways to run your MapReduce job:**

– **JobClient**.runJob (conf)

Blocks (waits for the job to complete before continuing)

– **JobClient**.submitJob (conf)

Does not block (driver code continues as the job is running)

**JobClient determines the proper division InputSplits of input data into.**

**JobClient then sends the job information to the JobTracker daemon on the cluster.**

**We didn’t specify any InputFormat so default is TextInputFormat.**

**We didn’t specify any no of Reducers so default no of reducers is one.Based of no of reducers we will get no of o/p file.**

**We didn’t specify any OutputFormat so default is KeyValueTextOutputFormat.**

**Default mapper is IdentityMaper**

**Default Reducer is IdentityReducer**

**Default Partitoner is HAshPartitioner etc..**

**Note:** Revise DriverCode once again so that you can understand it clearly.

**The Mapper Code:**

import org.apache.hadoop.io.**IntWritable**;

import org.apache.hadoop.io.**LongWritable**;

import org.apache.hadoop.io.**Text**;

import org.apache.hadoop.mapred.**MapReduceBase**;

import org.apache.hadoop.mapred.**Mapper**;

import org.apache.hadoop.mapred.**OutputCollector**;

import org.apache.hadoop.mapred.**Reporter**;

import java.io.**IOException**;

public class ***WordMapper*** extends **MapReduceBase** implements **Mapper**<**LongWritable, Text, Text, IntWritable**> {

//implementing the map() method of Mapper interface

public void map(**LongWritable** key, **Text** value,**OutputCollector**<**Text**, **IntWritable**> output, **Reporter** reporter)throws **IOException** {

//converting key from Text type to String type

**String** s = value.toString();

for (**String** word : s.split("\\W+")) {

if (word.length() > 0) {

output.collect(**new Text(**word**)**, **new IntWritable(**1**)**);

}

}

}

**The Reducer: Complete Code**

import org.apache.hadoop.io.**IntWritable**;

import org.apache.hadoop.io.**Text**;

import org.apache.hadoop.mapred.**OutputCollector**;

import org.apache.hadoop.mapred.**MapReduceBase**;

import org.apache.hadoop.mapred.**Reducer**;

import org.apache.hadoop.mapred.**Reporter**;

import java.io.**IOException**;

import java.util.**Iterator**;

public class ***SumReducer*** extends **MapReduceBase** implements **Reducer**<**Text, IntWritable, Text, IntWritable**> {

//implementing the reduce() method of Reducer

public void reduce(**Text** key, **Iterator**<**IntWritable**> values,**OutputCollector**<**Text**, **IntWritable**> output, **Reporter** reporter)throws **IOException** {

int wordCount = 0;

//retrieving the values from Iterator

while (values.hasNext()) {

**IntWritable** value = values.next();

//converting value from IntWritable type to int type

wordCount += value.get();

}

output.collect(key, new **IntWritable**(wordCount));

}

}

**Step 4:** Compile above programs

$ javac –classpath $HADOOP\_HOME/hadoop-core.jar \*.java

**Step 5:** Create a jar file for all .class files generated in the above step

$ jar wc.jar \*.class

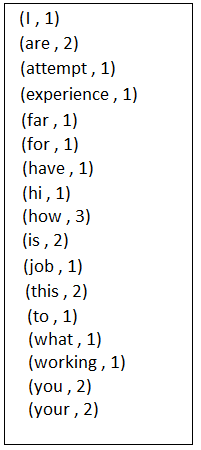
**Step 6**:Run the above create jar file on the file which is in stored hdfs

$ hadoop jar wc.jar WordCount file File

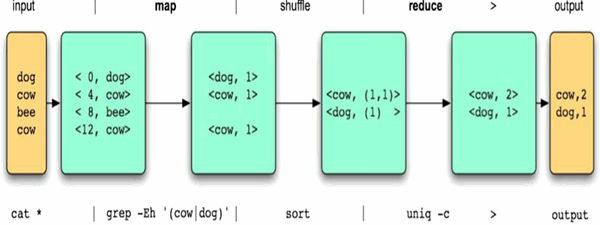
**Step 7:**

To see the content in the output file

$ hadoop fs -cat File/part-00000



Example Flow of Wordcount program:



**What Is The New API?**

**When Hadoop 0.20 was released, a ‘New API’ was introduced**

* It is designed to make the API easier to evolve in the future
* It favors abstract classes over interfaces

**The ‘Old API’ was deprecated**

**However, the New API is still not absolutely feature-complete in Hadoop 0.20.x**

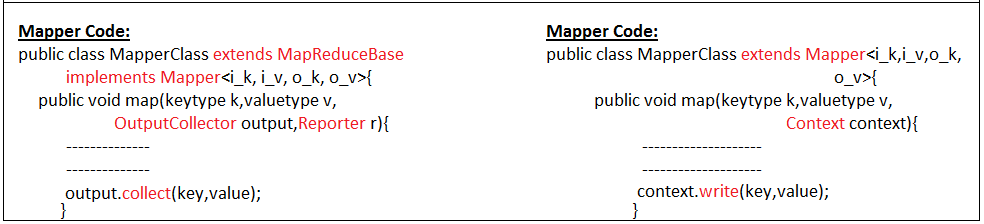
* The Old API should not have been deprecated as quickly as it was

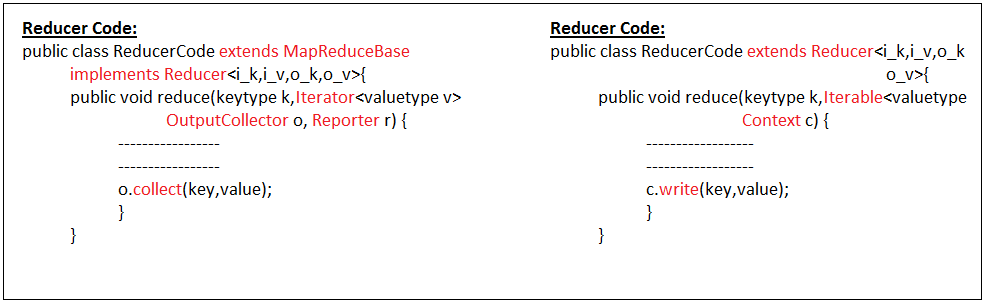
**Most developers still use the Old API**

**All the code examples in this course use the Old API as well as New API.**

Difference between old API and new API







**New API For Word Count**

**Driver Code:**

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class NewWordCount extends Configured{

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

if (args.length != 2) {

System.out.printf("plz give input and output directories");

System.exit(-1);

}

Job job = new Job();

job.setJarByClass(NewWordCount.class);

FileInputFormat.setInputPaths(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

job.setMapperClass(NewWordMapper.class);

job.setReducerClass(NewWordReducer.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

System.exit(job.waitForCompletion(true)?0:-1);

}

}

**Mapper Code:**

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Mapper.Context;

public class NewWordMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

@Override

public void map(LongWritable key, Text value,Context context)

throws IOException,InterruptedException {

String s = value.toString();

for (String word : s.split(" ")) {

if (word.length() > 0) {

context.write(new Text(word), new IntWritable(1));

}

}

}

}

**Reducer Code:**

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.Reducer.Context;

public class NewWordReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

@Override

public void reduce(Text key, Iterable<IntWritable> values, Context context)throws IOException,InterruptedException{

int count = 0;

while(values.iterator().hasNext()) {

IntWritable i= values.iterator().next();

count+=i.get();

}

context.write(key, new IntWritable(count));

}

}

**The Streaming API: Motivation**

**Many organizations have developers skilled in languages other than Java, such as**

– Ruby

– Python

– Perl

**The Streaming API allows developers to use any language they wish to write Mappers and Reducers**

– As long as the language can read from standard input and write to standard output.

**The Streaming API: Advantages**

**Advantages of the Streaming API:**

– No need for non-Java coders to learn Java

– Fast development time

– Ability to use existing code libraries

**How Streaming Works**

**To implement streaming, write separate Mapper and Reducer programs in the language of your choice**

– They will receive input via stdin

– They should write their output to stdout

**If TextInputFormat (the default) is used, the streaming Mapper just receives each line from the file on stdin**

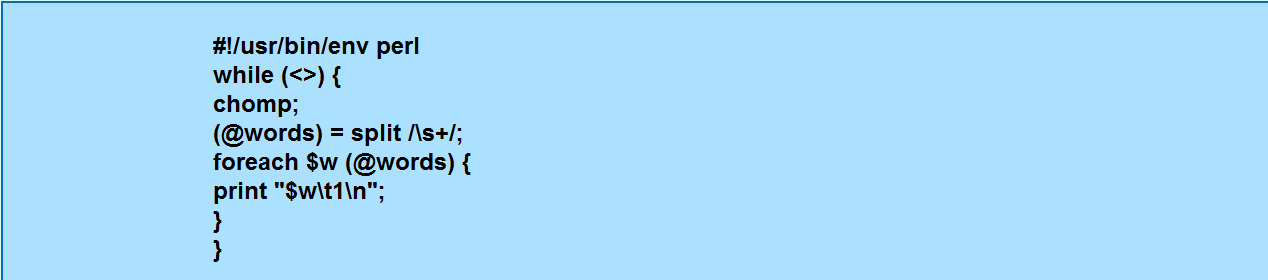
– No key is passed

**Streaming Mapper and streaming Reducer’s output should be sent to stdout as key (tab) value (newline)**

**Separators other than tab can be specified**

**Streaming: Example Mapper**

**Example streaming wordcount Mapper:**



**Streaming Reducers: Caution**

**Recall that in Java, all the values associated with a key are passed to the Reducer as an Iterator**

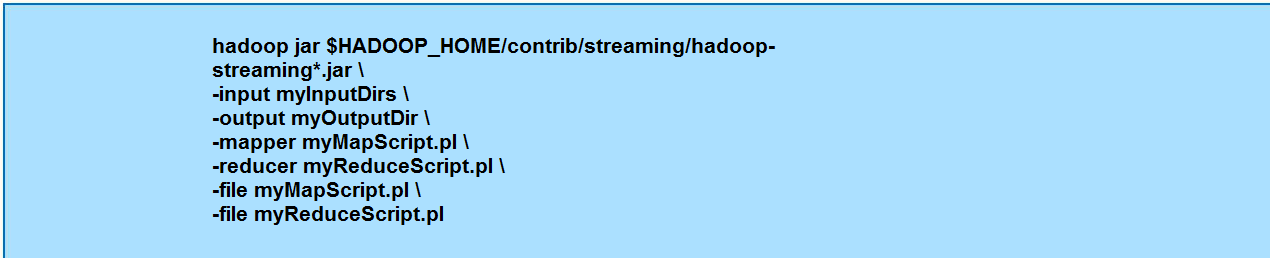
**Using Hadoop Streaming, the Reducer receives its input as (key,value) pairs**

– One per line of standard input

**Your code will have to keep track of the key so that it can detect when values from a new key start appearing.**

**Launching a Streaming Job**

**To launch a Streaming job, use e.g.,:**

****

**Many other command-line options are available**

– See the documentation for full details

**Note that system commands can be used as a Streaming Mapper or Reducer**

– For example: awk, grep, sed, or wc

**Integrated Development Environments**

**There are many Integrated Development Environments (IDEs) available**

**Eclipse is one such IDE**

– Open source

– Very popular among Java developers

– Has plug-ins to speed development in several different languages

**Launch Eclipse by double-clicking on the Eclipse icon on the**

**desktop**

– If you are asked whether you want to send usage data, hit **Cancel**

**Create java project in eclipse:**

**Step 1:**

****

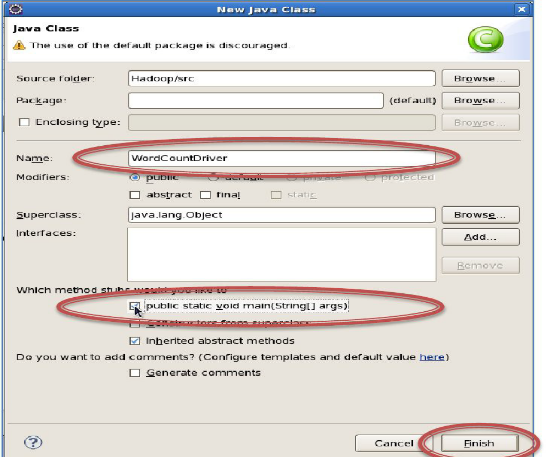
**Step 2:**

**Expand the ‘Hadoop’ project**

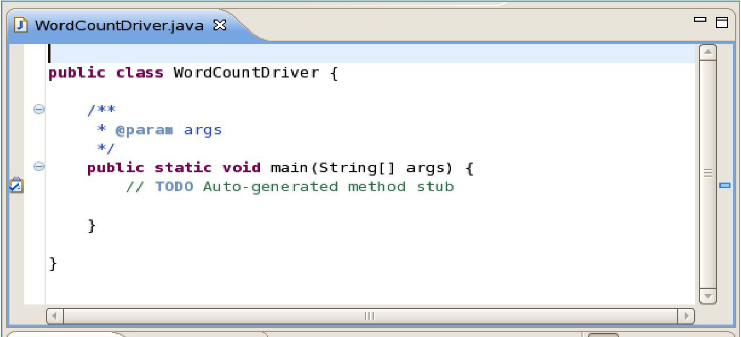
**Step 3:**

**Right-click on ‘src’, and choose New** ------->**Class**

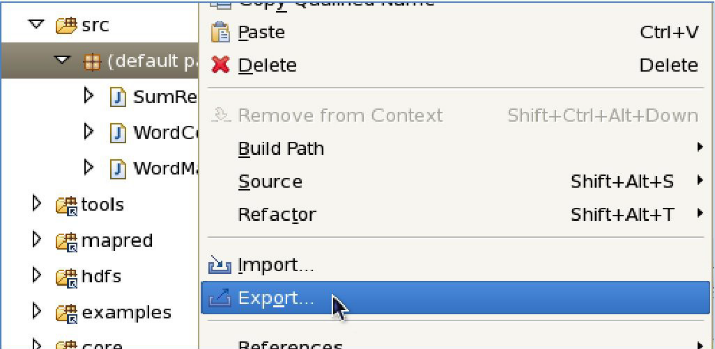
**Enter a class name, check the box to generate the main method stub, and then click Finish.**

****

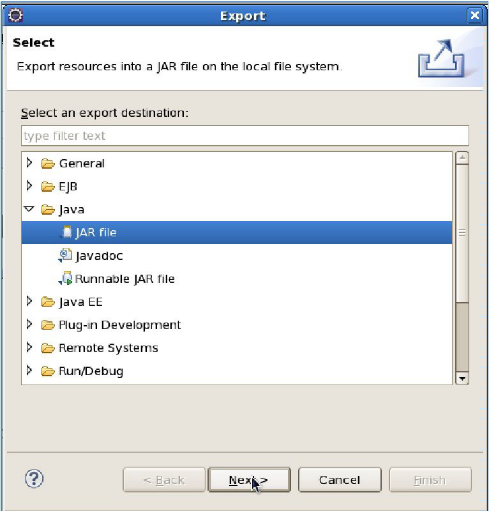
**You can now edit your class**

****

**Add other classes in the same way. When you are ready to test your code, right-click on the default package and choose Export**

****

**Expand ‘Java’, select the ‘JAR file’ item, and then click Next.**

****

Enter a path and filename inside /home/training (your home directory), and click Finish

Your JAR file will be saved; you can now run it from the command line with the standard

hadoop jar... command

**IdentityReducer:**

* An important consideration when creating your job is to determine the number of Reducers specified
* Default is a single Reducer
* With a single Reducer, one task receives *all* keys in sorted order

– This is sometimes advantageous if the output must be in completely sorted order

– Can cause significant problems if there is a large amount of intermediate data

– Node on which the Reducer is running may not have enough disk space to hold all intermediate data

– The Reducer will take a long time to run

**Jobs Which Require a Single Reducer**

* If a job needs to output a file where all keys are listed in sorted order, a single Reducer must be used
* Alternatively, the TotalOrderPartitioner can be used
* Uses an externally generated file which contains information about intermediate key distribution
* Partitions data such that all keys which go to the first reducer are smaller than any which go to the second, etc
* In this way, multiple Reducers can be used Concatenating the Reducers’ output files results in a totally ordered list.

**Jobs Which Require a Fixed Number of Reducers**

* Some jobs will require a specific number of Reducers
* Example: a job must output one file per day of the week
* Key will be the weekday
* Seven Reducers will be specified
* A Partitioner will be written which sends one key to each Reducer

**Jobs with a Variable Number of Reducers**

**Many jobs can be run with a variable number of Reducers**

**Developer must decide how many to specify**

– Each Reducer should get a reasonable amount of intermediate data, but not too much

– Chicken-and-egg problem

**Typical way to determine how many Reducers to specify:**

– Test the job with a relatively small test data set

– Extrapolate to calculate the amount of intermediate data expected from the ‘real’ input data

– Use that to calculate the number of Reducers which should be specified

**Creating Map-Only Jobs**

* To create a Map-only job, set the number of Reducers to 0 in your Driver code
* Anything written using the OutputCollector.collect method will be written to HDFS

**Conf.setReducerClass(IdentityReducer.class);**

– Rather than written as intermediate data

– One file per Mapper will be written

**IdentityReducer:**

**Driver Code:**

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat;

import org.apache.hadoop.mapred.FileOutputFormat;

import org.apache.hadoop.mapred.JobClient;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.mapred.lib.IdentityMapper;

import org.apache.hadoop.mapred.lib.IdentityReducer;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class DriverCode extends Configured implements Tool {

public int run(String[] args) throws Exception {

if (args.length != 2) {

System.out.printf("plz give input and output directories");

return -1;

}

JobConf conf = new JobConf(DriverCode.class);

conf.setJobName(this.getClass().getName());

FileInputFormat.setInputPaths(conf, new Path(args[0]));

FileOutputFormat.setOutputPath(conf, new Path(args[1]));

conf.setMapperClass(MapperCode.class);

**conf.setReducerClass(IdentityReducer.class);**

conf.setMapOutputKeyClass(Text.class);

conf.setMapOutputValueClass(IntWritable.class);

//conf.setOutputKeyClass(Text.class);

//conf.setOutputValueClass(IntWritable.class);

JobClient.runJob(conf);

return 0;

}

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

int exitCode = ToolRunner.run(new DriverCode(), args);

System.exit(exitCode);

}

}

**Mappe Code:**

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Mapper;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reporter;

public class MapperCode extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {

@Override

public void map(LongWritable key, Text value,OutputCollector<Text,IntWritable> output,Reporter r)

throws IOException {

String s = value.toString();

for (String word : s.split(" ")) {

if (word.length() > 0) {

output.collect(new Text(word), new IntWritable(1));

}

}

}

}

**IdentityMapper:**

* If we want to get RecordReader generated (key,value) pairs as the output to the output file then we can make our mapper as IdentityMapper.
* We have to configure this IdentityMapperin DriverCode class as below,

**Conf.SetMapperClass(IdentityMapper.class)**

**Driver Code:**

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat;

import org.apache.hadoop.mapred.FileOutputFormat;

import org.apache.hadoop.mapred.JobClient;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.mapred.lib.IdentityMapper;

import org.apache.hadoop.mapred.lib.IdentityReducer;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class DriverCode extends Configured implements Tool {

public int run(String[] args) throws Exception {

if (args.length != 2) {

System.out.printf("plz give input and output directories");

return -1;

}

JobConf conf = new JobConf(DriverCode.class);

conf.setJobName(this.getClass().getName());

FileInputFormat.setInputPaths(conf, new Path(args[0]));

FileOutputFormat.setOutputPath(conf, new Path(args[1]));

conf.setMapperClass(IdentityMapper.class);

conf.setReducerClass(IdentityReducer.class);

//conf.setMapOutputKeyClass(Text.class);

//conf.setMapOutputValueClass(IntWritable.class);

//conf.setOutputKeyClass(Text.class);

//conf.setOutputValueClass(IntWritable.class);

JobClient.runJob(conf);

return 0;

}

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

int exitCode = ToolRunner.run(new DriverCode(), args);

System.exit(exitCode);

}

}

**Combiner**

* Hadoop is there for Storing and Processing large amount of files , not recommended for small files.
* So as it can store large files into the HDFS, each file will be splited into number of input splits either with 64MB or 128MB block.
* Here each split will be given to one mapper at a time.Mapper will give output in the form of (key,value) paris. The outcome of Mapper is called as an intermediate data.
* If we process on very large file with more number of input splits, each split with one mapper, then lot of intermediate data will be generated.
* This intermediate data will be sent to Reducer as an input.
* But as we have discussed in the earlier of our course there is data locality for Mapper. Because Mapper will work on the data, local to that node, where input split is stored.
* So we can say that there is data locality for Mapper but not for Reducer.
* So all intermediate data must be transferred to the nodes where Reducers are working. So there it can result with lot of network traffic.

Here combiner can reduce the intermediate data which will be sent to Reducer.

Combiner always works on the data where intermediate data is generated.

Later this intermediate data will be passed to Reducer by combiner.

**Often, Mappers produce large amounts of intermediate data** – That data must be passed to the Reducers – This can result in a lot of network traffic

**It is often possible to specify a *Combiner***

– Like a ‘mini-Reducer’

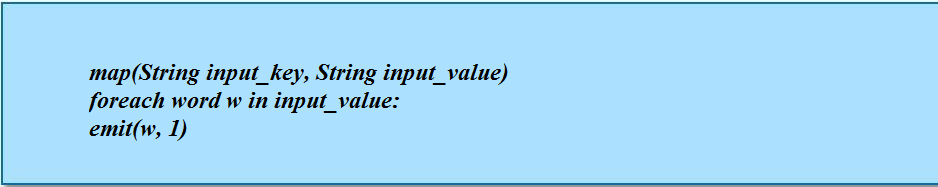
– Runs locally on a single Mapper’s output – Output from the Combiner is sent to the Reducers

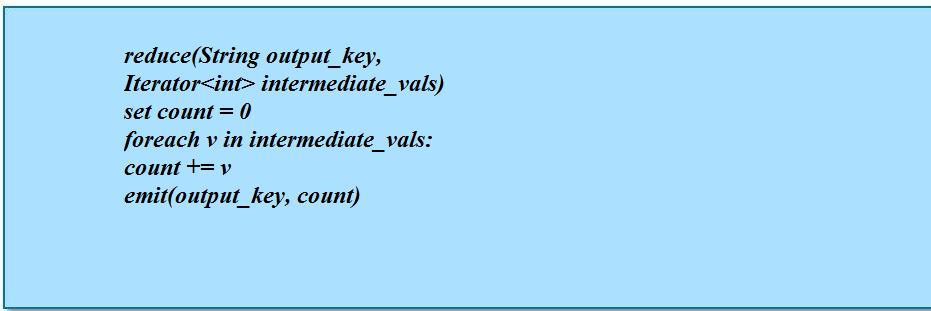
**Combiner and Reducer code are often identical**

* Technically, this is possible if the operation performed is commutative and associative
* In this case, input and output data types for the Combiner/Reducer must be identical.

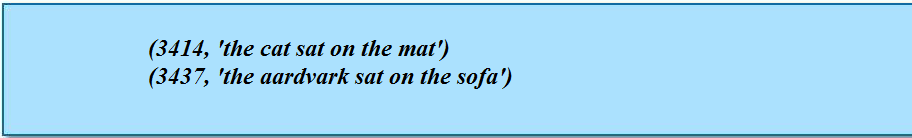
**MapReduce Example: Word Count**

**To see how a Combiner works, let’s revisit the WordCount example we covered earlier**

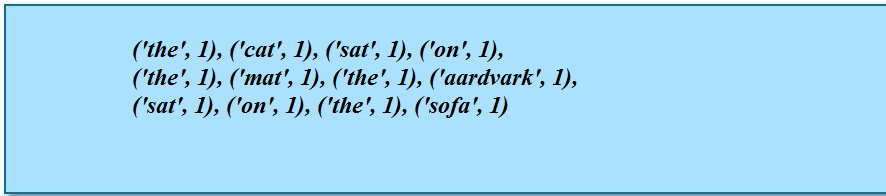
****

****

**Input to the Mapper:**

****

**Output from the Mapper:**

****

**Intermediate data sent to the Reducer:**

****

**Final Reducer output:**

****

**A Combiner would reduce the amount of data sent to the Reducer**

– Intermediate data sent to the Reducer after a Combiner using the

same code as the Reducer:

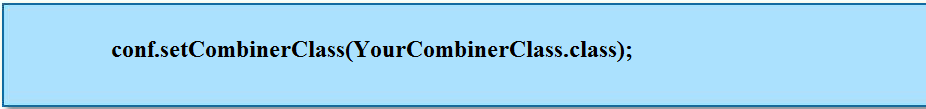
****

**Combiners decrease the amount of network traffic required during the shuffle and sort phase**

* Often also decrease the amount of work needed to be done by the Reducer

**Specifying a Combiner**

To specify the Combiner class in your MapReduce code, put the following line in your Driver:

****

**The Combiner uses the same interface as the Reducer**

– Takes in a key and a list of values

– Outputs zero or more (key, value) pairs

– The actual method called is the reduce method in the class

**VERY IMPORTANT: *Never* put code in the Combiner that must be run as part of your MapReduce job**

– The Combiner may not be run on the output from some or all of the Mappers

**Combiner Code:**

**Driver Code:**

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat;

import org.apache.hadoop.mapred.FileOutputFormat;

import org.apache.hadoop.mapred.JobClient;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class WordCountCombiner extends Configured implements Tool {

@Override

public int run(String[] args) throws Exception {

String input, output;

if(args.length == 2) {

System.out.println(“give input and output directories”);

Return -1;

}

JobConf conf = new JobConf(getConf(), WordCountCombiner.class);

conf.setJobName(this.getClass().getName());

FileInputFormat.setInputPaths(conf, new Path(args[0]));

FileOutputFormat.setOutputPath(conf, new Path(args[1]));

conf.setMapperClass(WordMapper.class);

conf.setReducerClass(SumReducer.class);

conf.setCombinerClass(SumCombiner.class);

conf.setMapOutputKeyClass(Text.class);

conf.setMapOutputValueClass(IntWritable.class);

conf.setOutputKeyClass(Text.class);

conf.setOutputValueClass(IntWritable.class);

if (conf.getCombinerClass() == null) {

throw new Exception("Combiner not set");

}

JobClient.runJob(conf);

return 0;

}

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

int exitCode = ToolRunner.run(new WordCountCombiner(), args);

System.exit(exitCode);

}

}

**Mapper Code:**

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Mapper;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reporter;

public class WordMapper extends MapReduceBase implements

Mapper<LongWritable, Text, Text, IntWritable> {

@Override

public void map(LongWritable key, Text value,

OutputCollector<Text, IntWritable> output, Reporter reporter)

throws IOException {

String s = value.toString();

for (String word : s.split("\\W+")) {

if (word.length() > 0) {

output.collect(new Text(word), new IntWritable(1));

}

}

}

}

**Reducer Code;**

import java.io.IOException;

import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Reducer;

import org.apache.hadoop.mapred.Reporter;

public class SumReducer extends MapReduceBase implements

Reducer<Text, IntWritable, Text, IntWritable> {

@Override

public void reduce(Text key, Iterator<IntWritable> values,

OutputCollector<Text, IntWritable> output, Reporter reporter)

throws IOException {

int wordCount = 0;

while (values.hasNext()) {

IntWritable value = values.next();

wordCount += value.get();

}

output.collect(key, new IntWritable(wordCount));

}

}

**Combiner Code;**

import java.io.IOException;

import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Reducer;

import org.apache.hadoop.mapred.Reporter;

public class SumCombiner extends MapReduceBase implements

Reducer<Text, IntWritable, Text, IntWritable> {

@Override

public void reduce(Text key, Iterator<IntWritable> values,

OutputCollector<Text, IntWritable> output, Reporter reporter)

throws IOException {

int wordCount = 0;

while (values.hasNext()) {

IntWritable value = values.next();

wordCount += value.get();

}

output.collect(key, new IntWritable(wordCount));

}

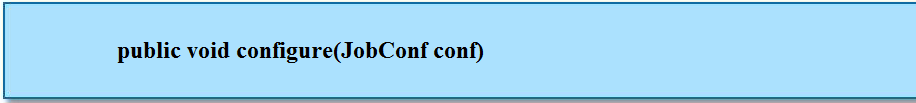
}

**The configure Method**

If you want your Mapper or Reducer to execute some code before the map or reduce method is called then we can use configure(-) method

* Initialize data structures
* Read data from an external file
* Set parameters

**The configure method is run before the map or reduce method is called for the first time**

****

**The close Method**

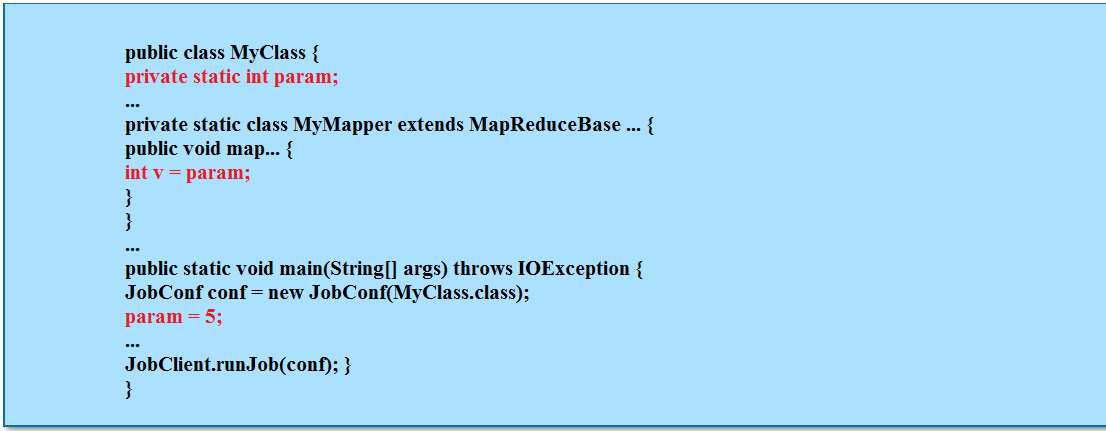
* Similarly, you may wish to perform some action(s) after all the records have been processed by -your Mapper or Reducer
* The close method is called before the Mapper or Reducer terminates

****

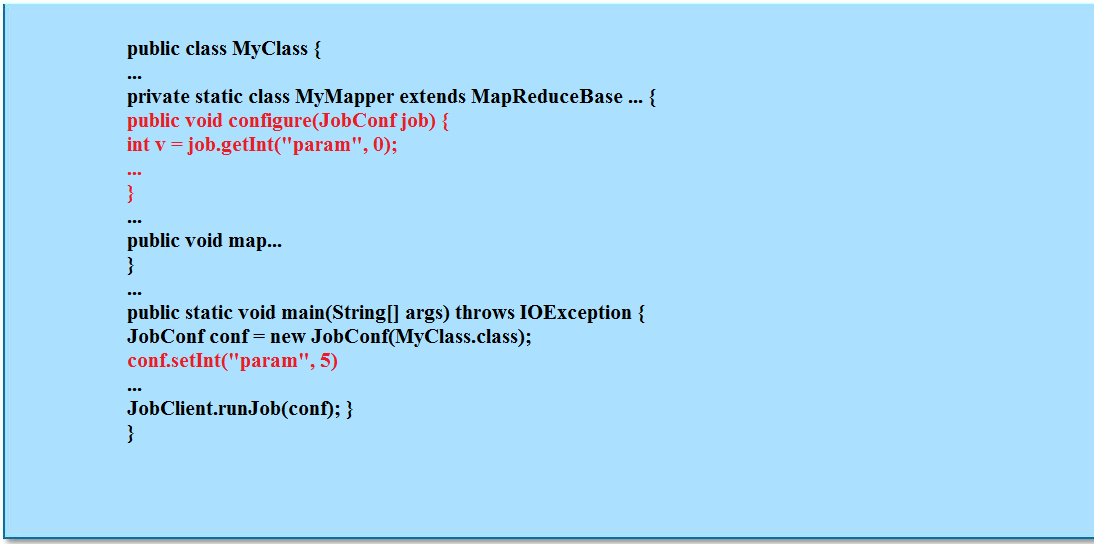
**Note that the close() method does not receive the JobConf object**

* You could save a reference to the JobConf object in the configure method and use it in the close method if necessary

**Passing Parameters: The Wrong Way!**

****

**Passing Parameters: The Right Way**

****

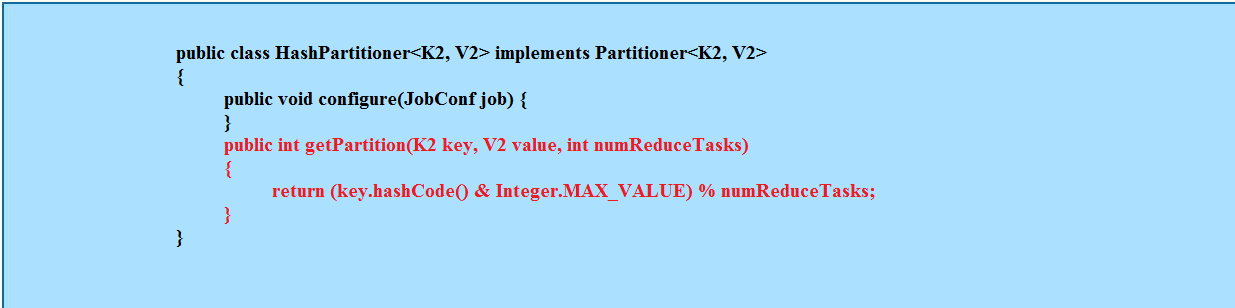
**Partitioner:**

**The Partitioner divides up the keyspace**

– Controls which Reducer each intermediate key and its associated values goes to

**Often, the default behavior is fine**

– Default is the HashPartitioner

****

**Custom Partitioners**

**Sometimes you will need to write your own Partitioner**

**Example: your key is a custom WritableComparable which contains a pair of values (a, b)**

– You may decide that all keys with the same value for **a** need to go to the same Reducer

– The default Partitioner is not sufficient in this case

**Custom Partitioners are needed when performing a secondary sort**

**Custom Partitioners are also useful to avoid potential performance issues**

– To avoid one Reducer having to deal with many very large lists of values

– Example: in our word count job, we wouldn't want a single reduce dealing with all the three- and four-letter words, while another only had to handle 10- and 11-letter words.

**Creating a Custom Partitioner**

**To create a custom partitioner:**

**1. Create a class for the partitioner**

– Should implement the Partitioner interface

**2. Create a method in the class called getPartition**

– Receives the key, the value, and the number of Reducers – Should return an int between 0 and one less than the number of Reducers

– e.g., if it is told there are 10 Reducers, it should return an int between 0 and 9

**3. Specify the custom partitioner in your driver code**

****

**Partitioner Code:**

**Driver Code:**

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat;

import org.apache.hadoop.mapred.FileOutputFormat;

import org.apache.hadoop.mapred.JobClient;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class WordCount extends Configured implements Tool {

public int run(String[] args) throws Exception {

if (args.length != 2) {

System.out.printf("plz give input and output directories");

return -1;

}

JobConf conf = new JobConf(WordCount.class);

conf.setJobName(this.getClass().getName());

FileInputFormat.setInputPaths(conf, new Path(args[0]));

FileOutputFormat.setOutputPath(conf, new Path(args[1]));

conf.setMapperClass(WordMapper.class);

conf.setReducerClass(WordReducer.class);

conf.setPartitionerClass(MyPartitioner.class);

conf.setNumReduceTasks(5);

conf.setMapOutputKeyClass(Text.class);

conf.setMapOutputValueClass(IntWritable.class);

conf.setOutputKeyClass(Text.class);

conf.setOutputValueClass(IntWritable.class);

JobClient.runJob(conf);

return 0;

}

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

int exitCode = ToolRunner.run(new WordCount(), args);

System.exit(exitCode);

}

}

**Mapper Code:**

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Mapper;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reporter;

public class WordMapper extends MapReduceBase implements Mapper<LongWritable,Text,Text,IntWritable> {

@Override

public void map(LongWritable key, Text value,OutputCollector<Text,IntWritable> output,Reporter r)

throws IOException {

String s = value.toString();

for (String word : s.split(" ")) {

if (word.length() > 0) {

output.collect(new Text(word), new IntWritable(1));

}

}

}

}

**Reducer Code :**

import java.io.IOException;

import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reducer;

import org.apache.hadoop.mapred.Reporter;

public class WordReducer extends MapReduceBase implements Reducer<Text,IntWritable,Text,IntWritable> {

@Override

public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,IntWritable> output,Reporter r)

throws IOException {

int count = 0;

while(values.hasNext()) {

IntWritable i= values.next();

count+=i.get();

}

output.collect(key, new IntWritable(count));

}

}

**Partitioner Code:**

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.mapred.Partitioner;

public class MyPartitioner implements Partitioner<Text,IntWritable>{

@Override

public void configure(JobConf arg0) {

// TODO Auto-generated method stub

}

@Override

public int getPartition(Text key, IntWritable value, int setNumReducers) {

String s = key.toString();

if(s.length()<2){

return 0;

}

if(s.length()<3){

return 1;

}

if(s.length()<4){

return 2;

}

if(s.length()>3){

return 3;

}else

return 4;

}

}

**Accessing HDFS Programmatically:**

**In addition to using the command-line shell, you can access HDFS programmatically**

– Useful if your code needs to read or write ‘side data’ in addition to the standard MapReduce inputs and outputs

**Beware: HDFS is not a general-purpose filesystem!**

– Files cannot be modified once they have been written, for example

**Hadoop provides the FileSystem abstract base class**

– Provides an API to generic file systems

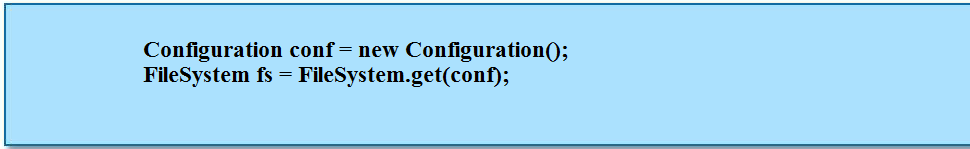
– Could be HDFS

– Could be your local file system

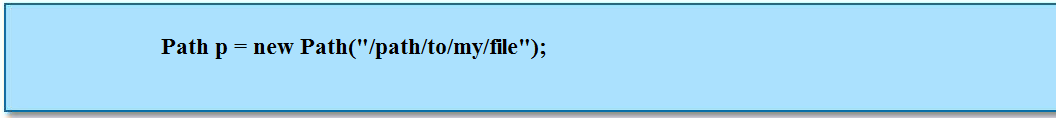
– Could even be, for example, Amazon S3

**The FileSystem API**

* In order to use the FileSystem API, retrieve an instance of it

****

* The conf object has read in the Hadoop configuration files, and therefore knows the address of the NameNode etc.
* A file in HDFS is represented by a Path object

****

**Some useful API methods:**

– FSDataOutputStream create(...) – Extends java.io.DataOutputStream

– Provides methods for writing primitives, raw bytes etc

– FSDataInputStream open(...) – Extends java.io.DataInputStream

– Provides methods for reading primitives, raw bytes etc

– boolean delete(...)

– boolean mkdirs(...)

– void copyFromLocalFile(...)

– void copyToLocalFile(...)

– FileStatus[] listStatus(...)

**Common MapReduce Algorithms**

**Some typical MapReduce algorithms, including**

– Sorting

– Searching

– Indexing

– Term Frequency – Inverse Document Frequency

– Word Co-Occurrence

**Introduction**

* MapReduce jobs tend to be relatively short in terms of lines of code
* It is typical to combine multiple small MapReduce jobs together in a single workflow
* You are likely to find that many of your MapReduce jobs use very similar code
* In this chapter we present some very common MapReduce algorithms
* These algorithms are frequently the basis for more complex MapReduce jobs

**Sorting:**

* MapReduce is very well suited for sorting large data sets
* Recall: keys are passed to the reducer in sorted order.
* Assuming the file to be sorted contains lines with a single value:

– Mapper is merely the identity function for the value

(k, v) -> (v, \_)

– Reducer is the identity function

(k, \_) -> (k, '')

* Trivial with a single reducer
* For multiple reducers, need to choose a partitioning function such that if k1 < k2, partition(k1) <= partition(k2)

**Sorting Algorithm:**

**DriverCode:**

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat;

import org.apache.hadoop.mapred.FileOutputFormat;

import org.apache.hadoop.mapred.JobClient;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.mapred.lib.IdentityReducer;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class DriverSorting extends Configured implements Tool {

public int run(String[] args) throws Exception {

if (args.length != 2) {

System.out.printf("plz give input and output directories");

return -1;

}

JobConf conf = new JobConf(DriverSorting.class);

conf.setJobName(this.getClass().getName());

FileInputFormat.setInputPaths(conf, new Path(args[0]));

FileOutputFormat.setOutputPath(conf, new Path(args[1]));

conf.setMapperClass(MapperSorting.class);

conf.setReducerClass(IdentityReducer.class);

conf.setMapOutputKeyClass(Text.class);

conf.setMapOutputValueClass(LongWritable.class);

JobClient.runJob(conf);

return 0;

}

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

int exitCode = ToolRunner.run(new DriverSorting(), args);

System.exit(exitCode);

}

}

**Mapper Code:**

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Mapper;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reporter;

public class MapperSorting extends MapReduceBase implements Mapper<LongWritable,Text,Text,LongWritable>{

public void map(LongWritable key,Text value,OutputCollector<Text,LongWritable> output,Reporter r)throws IOException{

output.collect(value,key);

}

}

**Searching:**

**Assume the input is a set of files containing lines of text**

**Assume the Mapper has been passed the pattern for which to search as a special parameter**

– We saw how to pass parameters to your Mapper in the previous

chapter

**Algorithm:**

– Mapper compares the line against the pattern

– If the pattern matches, Mapper outputs (line, \_)

– Or (filename+line, \_), or …

– If the pattern does not match, Mapper outputs nothing

– Reducer is the Identity Reducer

– Just outputs each intermediate key

**Searching Algorithm:**

**Driver Code:**

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat;

import org.apache.hadoop.mapred.FileOutputFormat;

import org.apache.hadoop.mapred.JobClient;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.mapred.lib.IdentityReducer;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class DriverSearching extends Configured implements Tool {

public int run(String[] args) throws Exception {

if (args.length != 2) {

System.out.printf("plz give input and output directories");

return -1;

}

JobConf conf = new JobConf(DriverSearching.class);

conf.setJobName(this.getClass().getName());

FileInputFormat.setInputPaths(conf, new Path(args[0]));

FileOutputFormat.setOutputPath(conf, new Path(args[1]));

conf.setMapperClass(MapperSearching.class);

conf.setReducerClass(IdentityReducer.class);

conf.setMapOutputKeyClass(Text.class);

conf.setMapOutputValueClass(Text.class);

JobClient.runJob(conf);

return 0;

}

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

int exitCode = ToolRunner.run(new DriverSearching(), args);

System.exit(exitCode);

}

}

**Mapper Code:**

import java.io.IOException;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileSplit;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Mapper;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reporter;

public class MapperSearching extends MapReduceBase implements Mapper<LongWritable,Text,Text,Text>{

String s;

public void configure(JobConf conf){

s = "the";

}

public void map(LongWritable key,Text value,OutputCollector<Text,Text> output,Reporter r)throws IOException{

FileSplit fs =(FileSplit) r.getInputSplit();

Path p = fs.getPath();

String s1 = p.getName();

String s2 = value.toString();

for(String word:s2.split(" ")){

if(word.equals(s)){

output.collect(new Text(s),new Text(s1));

}

}

}

}

**Indexing**

* Assume the input is a set of files containing lines of text
* Key is the byte offset of the line, value is the line itself
* We can retrieve the name of the file using the Reporter object

– More details on how to do this later

**Inverted Index Algorithm:**

**Mapper:**

– For each word in the line, emit (word, filename)

**Reducer:**

– Identity function

– Collect together all values for a given key (i.e., all filenames

for a particular word)

– Emit (word, filename\_list)

**Inverted Index: Dataflow**

**Aside: Word Count**

**Recall the WordCount example we used earlier in the course**

– For each word, Mapper emitted (word, 1)

– Very similar to the inverted index

**This is a common theme: reuse of existing Mappers, with minor**

**Modifications**

**InvertedIndex:**

**Driver Code:**

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat;

import org.apache.hadoop.mapred.FileOutputFormat;

import org.apache.hadoop.mapred.JobClient;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.mapred.KeyValueTextInputFormat;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class DriverIndex extends Configured implements Tool {

public int run(String[] args) throws Exception {

if (args.length != 2) {

System.out.printf("plz give input and output directories");

return -1;

}

JobConf conf = new JobConf(DriverIndex.class);

conf.setJobName(this.getClass().getName());

conf.setInputFormat(KeyValueTextInputFormat.class);

FileInputFormat.setInputPaths(conf, new Path(args[0]));

FileOutputFormat.setOutputPath(conf, new Path(args[1]));

conf.setMapperClass(MapperIndex.class);

conf.setReducerClass(ReducerIndex.class);

conf.setMapOutputKeyClass(Text.class);

conf.setMapOutputValueClass(Text.class);

conf.setOutputKeyClass(Text.class);

conf.setOutputValueClass(Text.class);

JobClient.runJob(conf);

return 0;

}

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

int exitCode = ToolRunner.run(new DriverIndex(), args);

System.exit(exitCode);

}

}

**Mapper Code:**

import java.io.IOException;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileSplit;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Mapper;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reporter;

public class MapperIndex extends MapReduceBase implements Mapper<Text,Text,Text,Text>{

@Override

public void map(Text key, Text value, OutputCollector<Text, Text> output,

Reporter r) throws IOException {

FileSplit fs = (FileSplit)r.getInputSplit();

Path p = fs.getPath();

String s = value.toString();

for(String s1:s.split(" ")){

if(s1.length()>0){

String word = p.getName().toString()+"@"+key.toString();

output.collect(new Text(s1),new Text(word));

}

}

}

}

**Reducer Code:**

import java.io.IOException;

import java.util.Iterator;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reducer;

import org.apache.hadoop.mapred.Reporter;

public class ReducerIndex extends MapReduceBase implements Reducer<Text,Text,Text,Text>{

public static String sep = ",";

public void reduce(Text key, Iterator<Text> values,

OutputCollector<Text, Text> output, Reporter r) throws IOException {

StringBuilder sb =new StringBuilder();

boolean firstValue = true;

while(values.hasNext()){

if(!firstValue){

sb.append(sep);

}

sb.append(values.next().toString());

firstValue = false;

}

output.collect(key,new Text(sb.toString()));

}

**Word Co-Occurrence:**

**Word Co-Occurrence measures the frequency with which two words appear close to each other in a corpus of documents**

– For some definition of ‘close’

**This is at the heart of many data-mining techniques**

– Provides results for “people who did this, also do that”

– Examples:

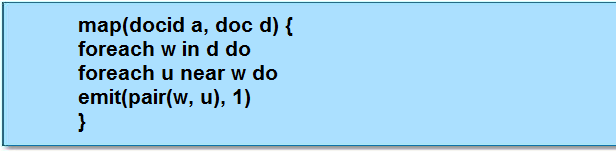
– Shopping recommendations

– Credit risk analysis

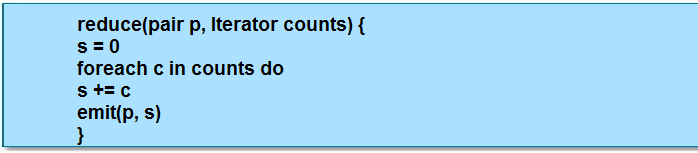
– Identifying ‘people of interest’

**Word Co-Occurrence: Algorithm**

**Mapper**

****

**Reducer**

****

**WordCo-Occurrence**

**Driver Code:**

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat;

import org.apache.hadoop.mapred.FileOutputFormat;

import org.apache.hadoop.mapred.JobClient;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class DriverWordCo extends Configured implements Tool {

public int run(String[] args) throws Exception {

if (args.length != 2) {

System.out.printf("plz give input and output directories");

return -1;

}

JobConf conf = new JobConf(DriverWordCo.class);

conf.setJobName(this.getClass().getName());

FileInputFormat.setInputPaths(conf, new Path(args[0]));

FileOutputFormat.setOutputPath(conf, new Path(args[1]));

conf.setMapperClass(MapperWordCo.class);

conf.setReducerClass(ReducerWordCo.class);

conf.setMapOutputKeyClass(Text.class);

conf.setMapOutputValueClass(IntWritable.class);

conf.setOutputKeyClass(Text.class);

conf.setOutputValueClass(IntWritable.class);

JobClient.runJob(conf);

return 0;

}

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

int exitCode = ToolRunner.run(new DriverWordCo(), args);

System.exit(exitCode);

}

}

**Mapper Code:**

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Mapper;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reporter;

public class MapperWordCo extends MapReduceBase implements Mapper<LongWritable,Text,Text,IntWritable>{

@Override

public void map(LongWritable key, Text value,

OutputCollector<Text, IntWritable> output, Reporter r)

throws IOException {

String s = value.toString();

String s1[] = s.split(" ");

for(int i=0;i<s1.length-1;i++){

String word = s1[i]+","+s1[i+1];

output.collect(new Text(word),new IntWritable(1));

}

}

}

**Reducer Code:**

import java.io.IOException;

import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reducer;

import org.apache.hadoop.mapred.Reporter;

public class ReducerWordCo extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {

@Override

public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,IntWritable> output,Reporter r) throws IOException {

int count = 0;

while(values.hasNext()) {

IntWritable i= values.next();

count+=i.get();

}

output.collect(key, new IntWritable(count));

}

}

**Integrating Hadoop into the Workflow**

**RDBMS Strengths:**

**Relational Database Management Systems (RDBMSs) have many**

**strengths**

* Ability to handle complex transactions
* Ability to process hundreds or thousands of queries per second
* Real-time delivery of results
* Simple but powerful query language

**RDBMS Weaknesses:**

**There are some areas where RDBMSs are less idea**

* Data schema is determined before data is ingested
* Can make ad-hoc data collection difficult
* Upper bound on data storage of 100s of terabytes
* Practical upper bound on data in a single query of 10s of terabytes

**Typical RDBMS Scenario**

**Using an interactive RDBMS to serve queries from a Web site etc**

**Data is later extracted and loaded into a data warehouse for future processing and archiving**

* Usually denormalized into an OLAP cube

**Hadoop Strenghts**

**Processing power scales with data storage**

* As you add more nodes for storage, you get more processing power ‘for free’

**Views do not need prematerialization**

* Ad-hoc full or partial dataset queries are possible

**Total query size can be multiple petabytes**

**Hadoop Weaknesses**

**Cannot serve interactive queries**

* The fastest Hadoop job will still take several seconds to run

**Less powerful updates**

* No transactions
* No modification of existing records

Sqoop

**It is difficult to use data stored in an RDBMS (such as Oracle database, MySQL or Teradata) in a MapReduce job**

* Lookup tables
* Legacy data

Possible to read directly from an RDBMS in your Mapper but it can lead to the equivalent of a distributed denial of service (DDoS) attack on your RDBMS.

**DDoS** attack is an attack to make other machines or network resources to be unavailable for the inteded users.

Better scenario is to import the data into HDFS to processs the data with MapReduce.

**Sqoop** is an open source tool written at Cloudera

It imports tables from an RDBMS into HDFS

* Just one table
* All tables in a database
* Just portions of a table
* Sqoop supports a WHERE clause

It can actually use MapReduce directly to import the data from RDBMS but it can make the number of Mappers to avoid DDoS scenarios

* Uses four Mappers by default and its value is configurable

**Uses a JDBC interface**

* Should work with any JDBC-compatible database.

**Imports data to HDFS as delimited text files or SequenceFiles**

* Default is a comma-delimited text file

**It can be used for incremental data imports**

* First import retrieves all rows in a table
* Subsequent imports retrieve just rows created since the last import

It g**enerates a class file which can encapsulate a row of the imported data**

* Useful for serializing and deserializing data in subsequent MapReduce jobs

**Custom Sqoop Connectors**

**Cloudera has partnered with other organizations to create custom Sqoop *connectors***

* Use a system’s native protocols to access data rather than JDBC
* It can provide much faster performance

**Current systems supported by custom connectors include:**

* Netezza
* Teradata
* MicroStrategy
* Oracle Database (connector developed with Quest Software)

**Others are in development**

**Custom connectors are not open source, but are free**

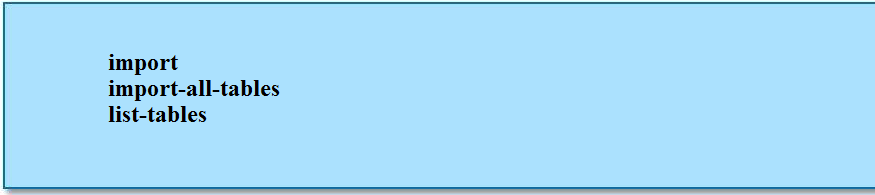
* Available from the Cloudera Web site

**Sqoop:**

**Standard syntax:**

****

**Tools include:**

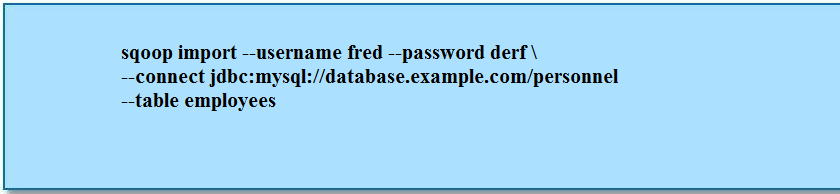
****

**Options include:**

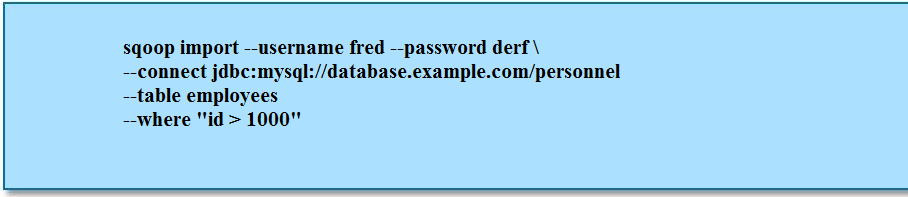
****

**Sqoop: Example**

**Example: import a table called employees from a database called personnel in a MySQL RDBMS**



**Example: as above, but only records with an id greater than 1000**

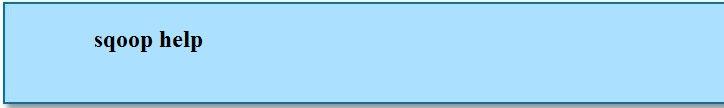
****

**Sqoop: Other Options**

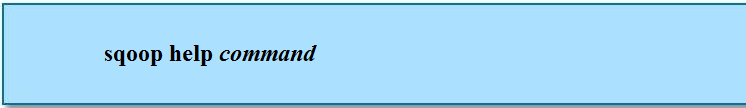
**Sqoop can take data from HDFS and insert it into an already- existing table in an RDBMS with the command**



**For general Sqoop help:**



**For help on a particular command:**



**Hands-On Exercise: Importing Data**

1.Log on to MySql

****

2. see databases in mysql

****

3. now use one database from list of databases

****

4. see all the tables from the database

****

5. describe the movie table

****

6.display all row and columns from movie table

****

7. review the structure and contents of the movie table for first 5 rows.

****

8.describe the movierating table

****

9.review the structure and contents of the movierating table for first 5 rows.

****

10. to quit from MySql

****

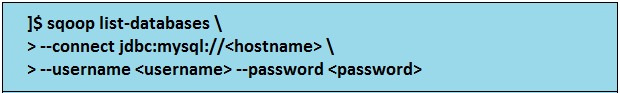
**Import with sqoop:**

We have to invoke sqoop on the command line to perform several commands. With it you can connect to our database server to list the databases along with thier schemas to which we have access, and list the tables available for loading. For database access , we have to give username and password to identify the server.

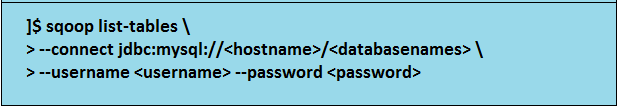
1.show the commands available in sqoop:

****

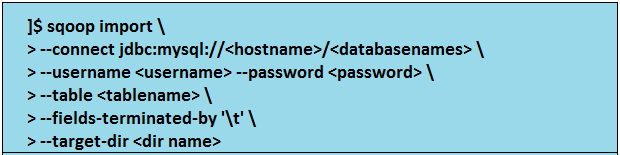
2.list the databases (schemas) in database server:

****

3.list the tables in the movielens database:

****

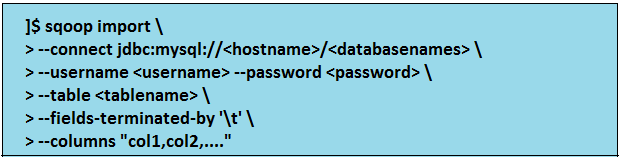
4.import the movie tables into Hadoop

****

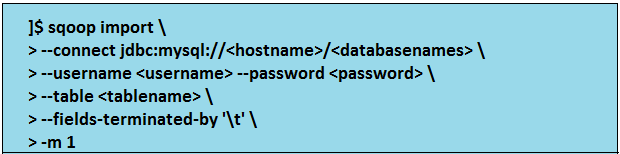
5.To import part of the data from the table by using where clause

****

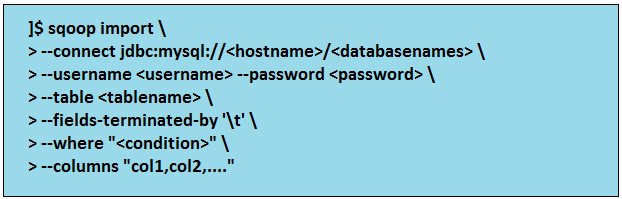
6.To import the specific columns from the table

****

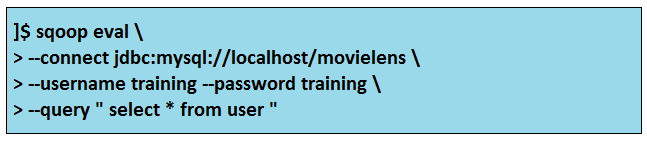
7. To import the table by working with one mapper ( -m 1) .

****

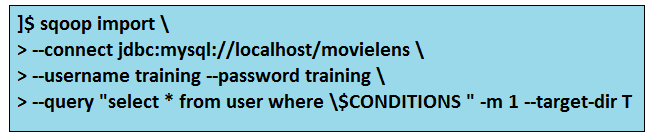
8. To import part of the data for specific columns

****

**Working with eval and query :**

****

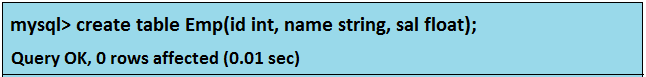
**Working with eval and query**

****

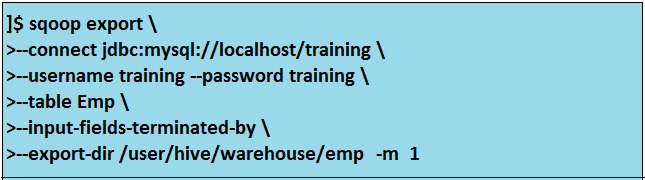
**Sqoop Export syntax:**

EXPORT DATA FROM HDFS TO RELATIONAL TABLE

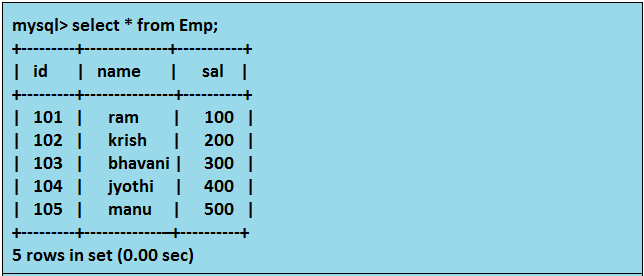
1. Let us create a table on the database into which we want our data to be exported. Let us name it “Emp”



2. Run the following command to export the data from HDFS to RDBMS



3.Now, we see the contents of the table to find the data exported from HDFS.



**Flume**

**Flume** is Open SourceInitially developed by Cloudera.

* Flume is a distributed, reliable, available service for efficiently moving large amounts of data as it is produced
* Ideally suited to gather logs from multiple systems and inserting them into HDFS as they are generated

**Flume’s design goals:**

* Reliability
* Scalability
* Manageability
* Extensibility

**Flume Node Characteristics**

**Each Flume node has a *source* and a *sink***

**Source**

* Tells the node where to receive data from

**Sink**

* Tells the node where to send data to

**Sink can have one or more *decorators***

* Perform simple processing on data as it passes though, such as

Compression

Encryption

awk, grep-like functionality

**Flume’s Design Goals**

**Reliability**

* The ability to continue delivering events in the face of system component failure

**Provides user-configurable reliability guarantees**

– End-to-end

– Once Flume acknowledges receipt of an event, the event will eventually make it to the end of the pipeline

– Store on failure

– Nodes require acknowledgment of receipt from the node one hop downstream

– Best effort

– No attempt is made to confirm receipt of data from the node downstream

**Scalability**

* The ability to increase system performance linearly – or better
* by adding more resources to the system
* Flume scales horizontally
* As load increases, more machines can be added to the configuration

**Manageability**

– The ability to control data flows, monitor nodes, modify the settings, and control outputs of a large system

**Flume provides a central Master, where users can monitor data flows and reconfigure them on the fly**

– Via a Web interface or a scriptable command-line shell

**Extensibility**

– The ability to add new functionality to a system

**Flume can be extended by adding *connectors* to existing storage layers or data platforms**

– General sources include data from files, syslog, and standard

output from a process

– General endpoints include files on the local filesystem or HDFS

– Other connectors can be added

– IRC

– Twitter streams

– HBase

– Developers can write their own connectors in Java

**Flume: Usage Patterns**

* Flume is typically used to ingest log files from real-time systems such as Web servers, firewalls and mailservers into HDFS
* Currently in use in many large organizations, ingesting millions of events per day
* At least one organization is using Flume to ingest over 200 million events per day
* Flume is typically installed and configured by a system administrator

HIVE

Hive Overview and Concepts

• Installation

• Table Creation and Deletion

• Loading Data into Hive

• Partitioning

• Bucketing

• Joins

* UDFs

1. **Introduction:**

* For the people who don’t know java, but still wants to analyze the data in HDFS, hive is an hadoop ecosystem project which built on top of mapreduce.
* Hive was originally devoloped by facebook people who are having more knowledge on mysql. So hive syntax was very much similar to mysql.
* Early Hive development work started at Facebook in 2007
* Today Hive is an Apache project under Hadoop
* Hive contains hive Query language (similar to mysql) and hive interpreter.
* Whenever you want to analyze the data write it in hive queries in hive QL and submit to hive interpreter.
* It converts the query to mapreduce job and submits to jobtracker.
* Each table you created in hive will represent one directory.
* Hive is just adding schema definition to the data presented in HDFS or in Local file system to analyze the data.
* So for storing this schema definition hive uses derby(single user) or mysql(shared) database.
* Hive is an abstraction on top of MapReduce.
* Allows users to query data in the Hadoop cluster without knowing Java or MapReduce.
* Uses the HiveQL language, very similar to SQL.
* The Hive Interpreter runs on a client machine.
* Turns HiveQL queries into MapReduce jobs and submits those jobs to the cluster.
* Hive does NOT provide low latency or real time queries**.**
* Even querying small amounts of data may take minutes**.**
* Designed for scalability and ease-of-use rather than low latency responses

**Note: this does *not* turn the cluster into a relational database server!**

* + It is still simply running MapReduce jobs
  + Those jobs are created by the Hive Interpreter

Ability to bring structure to various data formats

* Simple interface for ad hoc querying, analyzing and summarizing large amounts of data
* Access to files on various data stores such as HDFS and HBase

Hive does NOT provide low latency or realtime queries

• Even querying small amounts of data may take minutes

• Designed for scalability and ease-of-use rather than low latency responses

* Translates HiveQL statements into a set of MapReduce Jobs which are then executed on a Hadoop Cluster.

**HIVE METASTORE:**

To support features like schema(s) and data partitioning, Hive keeps its metadata in a

Relational Database

– Packaged with Derby, a lightweight embedded SQL DB

• Default Derby based is good for evaluation on testing

• Schema is not shared between users as each user has their own instance of embedded Derby

– Can easily switch another SQL installation such as MySQL

**HIVE :**

Re-used from Relational Databases

– Database: Set of Tables, used for name conflicts resolution

– Table: Set of Rows that have the same schema (same columns)

– Row: A single record; a set of columns

– Column: provides value and type for a single value

**Hive Limitations**

All ‘standard’ SQL is not supported like subqueries, for example.

* There is no support for UPDATE or DELETE
* No support for INSERTing single rows
* Relatively limited number of built-in functions
* No datatypes for date or time,It will use the STRING datatype instead.

# Data types in hive:

Hive supports both primitive and complex types. Primitives include numeric, Boolean, String, and timestamp types. The complex data types include arrays, maps, and structs.

**Primitive types :**

**Type Description Literal Example**

TINYINT 1 byte signed integer 1

(-128 to 127)

SMALLINT 2 byte signed integer 1

(-32768 to 32767)

INT 4 byte signed integer 1

BIGINT 8 byte signed integer 1

FLOAT 4 byte single precision 1.0

Floating point number

DOUBLE 8 byte double precision 1.0

floating point number

BOOLEAN true / false value TRUE

STRING character, string ‘a’ , ”a”

BINARY byte array not supported

TIMESTAMP timestamp with ‘2013-11-23 07:37:08.8’

Nanosecond precision

**Complex types :**

ARRAY An order collection of fields. Array(1, 2)

The fields must be of same type

MAP An unordered collection of key-value map(‘a’, 1 , ‘b’ , 2)

Pairs. Key must be primitives; values

may be any type. For a particular map,

the keys must be the same type and

the values must be the same type.

STRUCT A collection of named fields. The fields struct(‘a’ , 1 , 1.0)

May be of different types.

**Creation of Tables:**

* In hive we can create tables in two ways,

1. Internal
2. External

In case of internal table the directory of that table will be stored in /user/hive/warehouse/<tablename>. When we delete internal table, the directory will be deleted from **warehouse** directory.

In case of external table we must give path explicitly, where we want to store that table directory, otherwise it will give us error. When we delete the external table , table will be deleted from the hive shell but not from the directory where we stored that table directory explicitly .so we can use this table if necessary.

The difference between two tables is about LOAD and DROP semantics.

**Internal Tables:**

**hive> create table emp( id int, name string, salary float, city string)**

**> row format delimited**

**> fields terminated by ‘ \t ’ ;**

**OK**

**Time taken : 10.345 seconds**

After creating the table, we can upload data to that table either from local file system or from HDFS.

When we load data into a movie table, it is moved into Hive’s Warehouse directory.

For example:

**Loading data from HDFS to hive table:**

**hive> load data inpath ‘/user/training/emp’**

**>overwrite into table emp;**

We can see this **emp** file in the directory **/user/hive/warehouse/emp**

**hive> select \* from emp;**

**OK**

**-------------------**

if the table is dropped later, using

**hive> drop table emp;**

The table including its metadata and its data, is deleted. It bears repeating that since the initial LOAD performed a move operation, and the DROP performed a delete operation, the data no longer exists anywhere. This is what it means for Hive to manage the data.

If we delete the table, it will be deleted directly from

**/user/hive/warehouse/emp**

**Loading data from local file system:**

Create a file in local file system

**$ Cat>cricket**

Sachin Ind 100

Dravid Ind 60

Ponting Aus 68

Kallis SA 70

Tylor NZ 25

Cook Eng 30 CTRL+D

**hive> load data local inpath ‘/home/training/cricket’**

**>overwrite into table cricket;**

**OK**

**Time taken : 20.977 seconds**

We can see the cricket file in /user/hive/warehouse/cricket/cricket.

**hive > select count(\*) from cricket**

**OK**

**6**

**Time taken : 10.606 seconds**

**hive > select \* from cricket where country=’Ind’;**

**OK**

**Sachin Ind 100**

**Dravid Ind 60**

**Time taken : 32.377 seconds**

**hive> drop table cricket;**

**External Tables:**

An external table behaves differently. We can control the creation and deletion of the data. The location of the external data is specified at the table creation time. Indeed, it

**hive> create external table cricket(name string, country string, centuries int)**

**>row format delimited**

**>fields terminated by ‘ \t ’**

**>location ‘/ODI/Ckt’;**

**hive> load data local inpath ‘/home/training/cricket’**

**>overwrite into table cricket;**

**OK**

**Time taken : 20.977 seconds**

With the external keyword, hive does not know the data. so it will never move the table to its warehouse directory.

So external table is available only in the specified directory (/ODI/Ckt) but not in /userhive/warehouse directory. If we see the table in hive metastore, cricket table will be shown until on it is deleted from the hive.

Once it is deleted, movie table will not be available in hive metadata. But it will be in the external directory ‘/ODI/Ckt ‘.

Note: In External table the data and table is loosely coupled , if we are trying to drop the external table, the table is droped data is available into HDFS

\*\*\*\*External table with location is recommended.\*\*\*\*

Iternal tables are useful if you want hive to manage the complete lifecycle of your data including the deletion, whereas external tables are useful when the files are being used outside of Hive.

**Working With Collection Data Types (map, array, struct) :**

1).CREATE TABLE if not exists employee(name string , sal float,

subordinates array<string>, deductions map<string,float>,

address struct<street:string,city:string,state:string,zip:int>);

default delimiters:

for fields: CTRL+A

for collection items(may be array or map or struct): CTRL+B

for key and value : CTRL+C

for rows :\n

Note: Need to prepare i/p data using vi editor.In vi editor to enter any special characters like CTRL+A or CTRL+B we need to first enter CTRL+V

Load data into created table

---------------------------------

hive> load data inpath '<hdfs location of data>' overwrite into table employee;

hive> load data local inpath '<local fs system>' overwrite into table employee;

Note: overwrite is optional otherwise append will happen.

1).To print column header names while selecting data, we have to use the following command

set hive.cli.print.header=true;

but this command will work only for that hive shell in the terminal but not for new terminals

List all properties and values:

hive> set –v;

hive>select \* from employees;

**Partitions**

• **To increase performance Hive has the capability to partition data**

– The values of partitioned column divide a table into segments

– All partitions will be ignored at query time

– Similar to relational databases’ indexes but not as granular

• **Partitions have to be properly crated by users**

– When inserting data must specify a partition

• **At query time, whenever appropriate, Hive will automatically filter out partitions**

A table may be partitioned in multiple dimensions. For example, in addition to partitioning logs by date , we might also subpartition each date partition by country to permit efficient queries by location.

Partitions are defined at table creation time using the PARTITION BY clause, which takes a list of column definitions.

For example:

**hive > create table logs(ip string, count int)**

**> partitioned by(month string);**

When we load data into a partitioned table, the partition values are specified explicitly:

**hive> load data local inpath ‘/user/training/logoutput/part-00000’**

**>into table logs**

**>partition (month=’January’);**

If you want to check the partitions under the logs table:

**hive> show partitions logs;**

**hive> select id, name, dt**

**>from logs**

**>where country=’India’;**

**Single Table Insertion:**

We have already seen how to use the LOAD DATA operation to import data into a Hive table (or partition) by copying or moving files to the table’s directory. We can also populate a table with data from another Hive tables using an Insert statement. Or at creation time using the CTAS (create table as select).

**Inserts:**

Here is an example of insert statement:

**hive> create table Source(id int, name string,country string)**

**>row format delimited**

**>fields terminated by ‘ \t ’;**

**hive> load data local inpath ‘/home/training/source.txt’**

**>overwrite into table source;**

**hive> create table Target(id int, name string,country string)**

**>row format delimited**

**>fields terminated by ‘ \t ’;**

**hive> insert overwrite table target**

**>select id,name,country**

**>from source;**

**(or)**

**hive>create table target1 as select \* from source;**

For partitioned tables, we can specify the partition to insert into by supplying a PARTITION clause:

**Hive>create table target2(id int, name string,country string)**

**>partitioned by (dt string)**

**>row format delimited**

**>fields terminated by ‘ \t ’;**

**hive> insert overwrite table target2**

**>partition (dt=’2013-11-24’)**

**>select id, name,country**

**>from source;**

`Here the overwrite keyword means that the contents of the target table or the 2013-11-24 partition are replaced by the results of the SELECT statement.

If we want to add records to an already populated nonpartitioned table or partition , use INSERT INTO TABLE

We can specify the partition dynamically by determining the partition value from the SELECT statement.

**hive>insert overwrite table target**

**>partition (dt)**

**>select id , name , dt**

**>from source;**

**Multitable insertion:**

In HiveQL, you can turn the INSERT statement around and start with the FROM clause,

for the same effect:

**hive> create table employee( id int , name string, sal float, did string , dname string, loc string,**

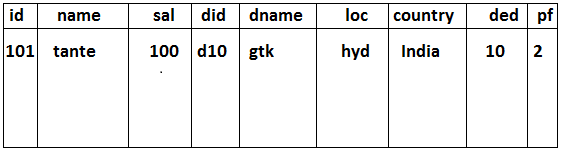
* **country string, ded float, pf int)**
* **row format delimited**
* **fields terminated by ‘\t’;**

**$ cat > employee**

**101 tante 100 d10 gtk hyd India 10 2**

**hive> load data local inpath ‘/home/training/employee’**

* **into table employee;**

****

Here’s an example that computes various statistics over the weather dataset:

**Create three tables now,**

**hive> create table target1( id int, name string, sal float, did string)**

* **row format delimited**
* **fields terminated by ‘\t’;**

**hive> create table target2( did string, dname string, loc string, country string)**

* **row format delimited**
* **fields terminated by ‘\t’;**

**Hive> create table target3(id int, name string, did string, ded float, pf int)**

* **row format delimited**
* **fields terminated by ‘\t’;**

**hive> FROM employee**

* **insert overwrite table target1**
* **select id , name , sal , did**
* **insert overwrite table target2**
* **select did, dname, loc, country**
* **insert overwrite table target3**
* **select id , name , sal , did, ded, pf;**

There is a single source table (records2), but three tables to hold the results from three

Different queries over the source.

**Buckets:**

There are two reasons to organize our tables (or partitions) into buckets. The first is to enable more efficient queries. Bucketing imposes extra structure on the table, which Hive can take advantage of when performing certain queries.

**Mechanism to query and examine random samples of data**

**Break data into a set of buckets based on a hash function of a "bucket column"**

-Capability to execute queries on a sub-set of random data

**Doesn’t automatically enforce bucketing**

-User is required to specify the number of buckets by setting # of reducer

**Joins:**

Joins will be performed when we want the data from multiple tables.Joingin tables must have one common column.

**Inner joins:**

The simplest kind of join is the inner join, where each match in the input tables results

in a row in the output. Consider two small demonstration tables: sales, which lists the

names of people and the ID of the item they bought; and things, which lists the item

ID and its name:

hive> **SELECT \* FROM sales;**

Joe 2

Hank 4

Ali 0

Eve 3

Hank 2

hive> **SELECT \* FROM things;**

2 Tie

4 Coat

3 Hat

1 Scarf

We can perform an inner join on the two tables as follows:

hive> **SELECT sales.\*, things.\***

> **FROM sales JOIN things ON (sales.id = things.id);**

Joe 2 2 Tie

Hank 2 2 Tie

Eve 3 3 Hat

Hank 4 4 Coat

The table in the FROM clause (sales) is joined with the table in the JOIN clause (things),

using the predicate in the ON clause. Hive only supports equijoins, which means that

only equality can be used in the join predicate, which here matches on the id column

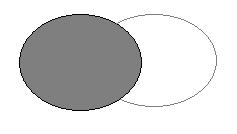
in both tables.

**Outer Joins:**

Outer joins allow you to find non-matches in the tables being joined.

**Left Outer**

Row from the first table are included whether they have a match or not. Columns from the unmatched (second) table are set to null.



hive> **SELECT sales.\*, things.\***

> **FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);**

Ali 0 NULL NULL

Joe 2 2 Tie

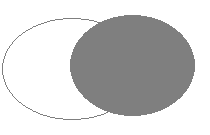
Hank 2 2 Tie

Eve 3 3 Hat

Hank 4 4 Coat

**Right Outer**

The opposite of Left Outer Join: Rows from the second table are included no matter what. Columns from the unmatched (first) table are set to null.



hive> **SELECT sales.\*, things.\***

> **FROM sales RIGHT OUTER JOIN things ON (sales.id = things.id);**

NULL NULL 1 Scarf

Joe 2 2 Tie

Hank 2 2 Tie

Eve 3 3 Hat

Hank 4 4 Coat

**Full Outer**

Rows from both sides are included. For unmatched rows the columns from the ‘other’ table are set to null.

hive> **SELECT sales.\*, things.\***

> **FROM sales FULL OUTER JOIN things ON (sales.id = things.id);**

Ali 0 NULL NULL

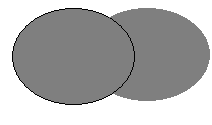
NULL NULL 1 Scarf

Joe 2 2 Tie

Hank 2 2 Tie

Eve 3 3 Hat

Hank 4 4 Coat



Semi joins:

Hive doesn’t support IN subqueries (at the time of writing), but you can use a LEFT SEMI

JOIN to do the same thing.

Consider this IN subquery, which finds all the items in the things table that are in the

sales table:

**SELECT \***

**FROM things**

**WHERE things.id IN (SELECT id from sales);**

We can rewrite it as follows:

hive> **SELECT \***

> **FROM things LEFT SEMI JOIN sales ON (sales.id = things.id);**

2 Tie

3 Hat

4 Coat

**CREATE TABLE...AS SELECT**

It’s often very convenient to store the output of a Hive query in a new table, perhaps

because it is too large to be dumped to the console or because there are further processing

steps to carry out on the result.

The new table’s column definitions are derived from the columns retrieved by the

SELECT clause. In the following query, the target table has two columns named col1

and col2 whose types are the same as the ones in the source table:

**CREATE TABLE target**

**AS**

**SELECT id,name**

**FROM source;**

A CTAS operation is atomic, so if the SELECT query fails for some reason, then the table

is not created.

You can rename a table using the ALTER TABLE statement:

**ALTER TABLE source RENAME TO target;**

Altering Tables

Since Hive uses the schema on read approach, it’s flexible in permitting a table’s definition

to change after the table has been created

You can rename a table using the ALTER TABLE statement:

**ALTER TABLE source RENAME TO target;**

For example, consider adding a new column:

**ALTER TABLE target ADD COLUMNS (col3 STRING);**

**Dropping Table:**

The DROP TABLE statement deletes the data and metadata for a table. In the case of

external tables, only the metadata is deleted—the data is left untouched.

If you want to delete all the data in a table, but keep the table definition (like DELETE or

TRUNCATE in MySQL), then you can simply delete the data files. For example:

hive> **dfs -rmr /user/hive/warehouse/target;**

Another possibility, which achieves a similar effect, is to create a new, empty table that

has the same schema as the first, using the LIKE keyword:

**CREATE TABLE new\_table LIKE existing\_table;**

**UDF (User Defined Function)**

**----------------------------------------**

1) We must write UDF in java by using hadoop and hive API.

2) Compile and create a jar file

3) add/register that jar file with hive

4) We can use as like default function

5) Udfs accept one row as i/p and give onr row as o/p

Udfs must satisfy follow rules

--------------------------------------

1. Udf must be a sub class of org.apache.hadoop.hive.ql.exec.UDF

2. Udf must implement atleast one evaluate(-) method

**3.**

package com.hadoop.hive;

**import** org.apache.commons.lang.StringUtils;

**import** org.apache.hadoop.hive.ql.exec.UDF;

**import** org.apache.hadoop.io.Text;

**public** **class** TestUDF **extends** UDF

{

Text t = **new** Text();

**public** Text evaluate(Text str)

{

**if** (str == **null**) {

**return** **null**;

}

t.set(StringUtils.*strip*(str.toString()));

**return** t;

}

**public** Text evaluate(Text str, String stripChars)

{

**if** (str == **null**) {

**return** **null**;

}

t.set(StringUtils.*strip*(str.toString(), stripChars));

**return** t;

}

}

4.After creating jar file need to add tha jar file in hive.

**hive>ADD JAR /home/training/workspace/hive-examples.jar;**

**5.we have to create temporary function for the class after creating jar file**

**hive> CREATE TEMPORARY FUNCTION strchar AS 'com.hadoop.hive.TestUDF’;**

**hive> create table emp( id int, name string, city string)**

**> row format delimited**

**> fields terminated by ‘ \t ’ ;**

**OK**

**Time taken : 10.345 seconds**

**$ cat > emp**

101 ram hyd

102 krish Chennai

103 subba hyd

104 banana bang

**hive> load data local inpath ‘/home/training/emp’ into table emp;**

**hive> select strchar(name) from emp;**

**ram**

**krish**

**subba**

**banana**

becoz of the above query , strchar function will call one parameterized evaluate(-) method and it removes the empty spaces from extreme positions of the name

**hive> select strchar(name, ‘ab’) from emp;**

**ram**

**krish**

**su**

**nan**

becoz of the above query, strchar function will call two parameterized evaluate(-,-) method and it removes the special characters from extreme positions of the name nut not empty spaces .

PIG

**Pig was originally created at Yahoo! to answer a similar need to Hive**

* Many developers did not have the Java and/or MapReduce knowledge required to write standard MapReduce programs
* But still needed to query data

**Pig is a dataflow language**

* Language is called PigLatin.
* Relatively simple syntax.
* Under the covers, PigLatin scripts are turned into MapReduce jobs and executed on the cluster.
* Installation of Pig requires no modification to the cluster
* The Pig interpreter runs on the client machine
* Turns PigLatin into standard Java MapReduce jobs, which are then submitted to the JobTracker
* There is (currently) no shared metadata, so no need for a shared metastore of any kind.

**Pig Concepts:**

* In Pig, a single element of data is an *atom*
* A collection of atoms – such as a row, or a partial row – is a *tuple*
* Tuples are collected together into *bags*
* Typically, a PigLatin script starts by loading one or more datasets into bags, and then creates new bags by modifying those it already has existed.

**Pig Features**

* Pig supports many features which allow developers to perform sophisticated data analysis without writing Java MapReduce code

– Joining datasets

– Grouping data

– Referring to elements by position rather than name

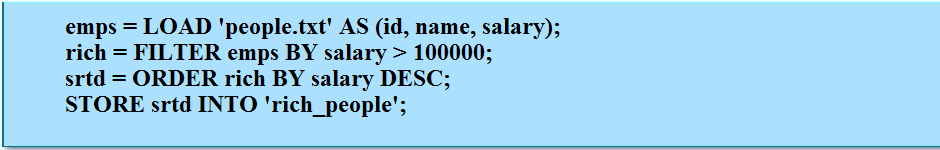
– Useful for datasets with many elements

– Loading non-delimited data using a custom SerDe

– Creation of user-defined functions, written in Java

– And more

**A Sample Pig Script:**



* Here, we load a file into a bag called emps
* Then we create a new bag called rich which contains just those
* records where the salary portion is greater than 100000
* Finally, we write the contents of the srtd bag to a new directory in HDFS
* By default, the data will be written in tab-separated format Alternatively, to write the contents of a bag to the screen, say

Let’s look at a simple example by writing the program to calculate the maximum recorded temperature by year for the weather dataset in Pig Latin. The complete program is only a few lines long:

**Pig Installation:**

Pig is running on client application. We can even install pig on hadoop cluster with the following setup.

Download the compatible version of pig from <http://pig.apache.org/release.html>, and unzip the tar file in correct location.

**% tar xvf pig-x.y.z.tar.gz**

Export the below path……

**% export PIG\_INSTALL=/home/<username>/pig-x.y.z**

**% export PATH=$PATH:$PIG\_INSTALL/bin**

we need to set the JAVA\_HOME environment variable to point to a suitable java installation.

Try typing pig –help to get usage instructions.

**Execution Types:**

**P**ig has two execution modes.

1.local mode

2.MapReduce mode

**Local mode:**

In local mode , pig runs in a single JVM and access the local file system. This mode is suitable only for small data sets but not for big data sets.

We can set this local mode execution type by using “-x” or “-exectype” option. To run in local mode , set the option to local

**]$ pig –x local**

**grunt>**

this starts grunt, the pig interactive shell.

**MapReduce mode:**

In MapReduce mode , pig translates queries into MapReduce jobs and runs on a hadoop cluster. This cluster can be pseudo-distributed mode or fully distributed mode. MapReduce mode will work when we want to run pig on big data sets.

To set the option for mapredcuce mode we should use either of the following command

**]$ pig -x mapreduce**

**Or**

**]$ pig**

-- max\_temp.pig: Finds the maximum temperature by year

**]$ cat > climate**

1949 27 hyd

1950 30 bang

1970 42 chennai

1949 35 mumbai

1970 48 chennai

1960 11 simla

1950 22 bang

Ctrl + d ( to save and exit the file )

**]$ pig -x local**

**Relations, Bags, Tuples, Fields**

Pig Latin statements work with relations. A relation can be defined as follows:

• A relation is a bag (more specifically, an outer bag).

• A bag is a collection of tuples.

• A tuple is an ordered set of fields.

• A field is a piece of data.

A Pig relation is a bag of tuples. A Pig relation is similar to a table in a relational database,

where the tuples in the bag correspond to the rows in a table. Unlike a relational table,

however, Pig relations don't require that every tuple contain the same number of fields or that the fields in the same position (column) have the same type.

Also note that relations are unordered which means there is no guarantee that tuples are

processed in any particular order. Furthermore, processing may be parallelized in which case tuples are not processed according to any total ordering.

Pig is more flexible in defining schema. First we can load data and then we can decide which data type is suitable for the field names, where as in SQL data types can be declared before the data is loaded into the system. Pig is designed for analyzing plain input files with no

Associated type information, so it is quite natural to choose types for fields later than you would with an RDBMS.

**grunt>env = LOAD ‘/home/training/climate’**

**AS (year:int , temp:int , city:chararray);**

Here the LOAD operator is a relation , which is just a set of tuples. A tuple is just like a row of data in a database table, with multiple fields in a particular order.

In this example , the LOAD function produces a set of (year, temperature, city) tuples that are presented in the input file. We write a relation with one tuple per line , where tuples are represented as comma-separated items in parentheses.

For simplicity, the program assumes that the input is tab-delimited text, with each line

having just year, temperature, and quality fields.

This line describes the input data we want to process. The year:int notation describes the field’s name and type; a chararray is like a Java string, and an int is like a Java int. The LOAD operator takes a URI argument; here we are just using a local file, but we could even refer to an HDFS URI. The AS clause (which is optional) gives the fields names to make it convenient to refer to them in subsequent statements.

Bag or Relations are given names or aliases those which we are referring. This relation is given the env alias. We can examine the contents of an alias using the DUMP operator.

A ***tuple***is just like a row of data in a database table, with multiple fields in a particular order. In this example, the LOAD function produces a set of (year,temp, city ) tuples that are present in the input file. We write a relation with one tuple per line, where tuples are represented as comma-separated items in parentheses:

**grunt> DUMP env;**

(1949,27,hyd)

(1950,30,bang)

(1970,42,Chennai)

(1949,35,mumbai)

(1970,48,Chennai)

(1960,11,simla)

(1950,22,bang)

we can even see the schema of the relation by using DESCRIBE operator on the relation’s alias.

**Types**

Pig has four numeric types:

Int

Long

Float

double,

which are identical to their ava counterparts. There is also a bytearray type, like Java’s byte array type for representing blob of binary data, and chararray, which, like java.lang.String, represents textual data in UTF-16 format, although it can be loaded or stored in UTF-8 format.

Pig does not have types corresponding to Java’s boolean,6 byte, short, or char primitive types. These are all easily represented using Pig’s int type, or chararray for char.

The numeric, textual, and binary types are simple atomic types. Pig Latin also has three complex types for representing nested structures: tuple, bag, and map.

**Category Type Description Literal example**

Numeric int 32-bit signed integer 1

long 64-bit signed integer 1L

float 32-bit floating-point number 1.0F

double 64-bit floating-point number 1.0

Text chararray Character array in UTF-16 format 'a'

Binary bytearray Byte array Not supported

Complex tuple Sequence of fields of any type (1,'pomegranate')

Bag An unordered collection

of tuples, possibly with

duplicates {(1,'pomegranate'),(2)}

map A set of key-value pairs.

Keys must be character arrays;

values may be any type ['a'#'pomegranate']

**Bag Schema :**

A relation in pig may have an associated schema, which gives the fields in the relation names and types. We have already seen how an AS clause in a LOAD statement is used to attach a schema to a relation.

**Case 1:**

**grunt>env = LOAD ‘/home/training/climate’**

**AS (year:int , temp:int , city:chararray);**

**grunt> DESCRIBE env;**

env: {year: int, temp: int, city: chararray}

this tells us that env relation has three fields, with aliases year, temperature , and city. These are the names we have given in the AS clause

**Case 2:**

if we are not specifying any fields types ( data types) for the aliases, then pig will set them as bytearray types by default.

**grunt>** **env1 = LOAD ‘/home/training/climate’**

**AS (year, temp, city);**

For the above example , we have not given field types for the aliases ( year,temp and city). So these fields will be specified as bytearray types.

**grunt> DESCRIBE env1;**

env1: {year:bytearray, temp:bytearray, city:bytearray}

**Case 3:**

**grunt>** **env2 = LOAD ‘/home/training/climate’**

**AS (year:int, temp, city);**

if we are specifying type for one field , that alias only will be specified with the given type and other fields will be specified as bytearray types. Below example shows this scenario.

**grunt> DESCRIBE env2;**

env2: {year:int, temp:bytearray, city:bytearray}

**Case 4:**

**grunt> env3 = LOAD ‘/home/training/climate’;**

if we are not specifying AS clause then no schema will be generated. Simply it is saying that “ **schema for relation unknown**”

**grunt> DESCRIBE env3;**

schema for env3 unknown

**Filtering Data:**

Once we have loaded some data into a relation, often the next step is to filter is to remove the data that you are not interested in. By filtering the data , we can minimize the amount of data flowing through the system, which can improve efficiency.

**grunt>** **filtered\_env = FILTER env BY temp>30;**

**FILTER** is a command to filter the tuple from the given bag. Here in the above example **FILTER** command will filter the env bag with the temperature greater than 30 and stores the output in filtered\_env bag.

**grunt> DUMP filtered\_env;**

(1970,42,Chennai)

(1949,35,mumbai)

(1970,48,Chennai)

For every newly generated bag pig will automatically generates schema. From the above **filered\_env** bag , pig will generate schema as below….

**grunt> DESCRIBE filtered\_env**

filtered\_env: {year: int, temp: int, city: chararray}

**Grouping Data:**

The group command will group the data in a single relation. GROUP creates a relation whose first field is the grouping field, which is given the alias group. The second field is a bag containing the grouped fields with the same schema as the original relation.

**grunt>grouped\_env = GROUP env BY year;**

From the above statement GROUP command will group all the tuple in to a single relation for the same alias year

**grunt> DUMP grouped\_env;**

(1949 , {(1949,27,hyd)})

(1950, {(1950,30,bang),(1950,22,bang)})

(1970, {(1970,42,Chennai),(1970,48,Chennai)})

(1949, {(1949,35,Mumbai)})

(1960, {(1960,11,simla)})

For the above grouped\_env new schema will be generated as follows…….

**grunt> DESCRIBE grouped\_env;**

**grouped\_env: { group: int , env : { year: int, temp: int, city: chararray}}**

Here GROUP creates a relation whose first field is the grouping field, which is given the alias group. The second field is a bag containing the grouped fields with the same schema as the original relation.

There are two more special grouping operations: ALL and ANY. ALL command will group all the tuples in a relation in a single group, as if the GROUP function were a constant.

**Grunt> group\_env = GROUP env ALL;**

**Grunt> DUMP group\_env;**

(all , {(1949,27,hyd),(1950,30,bang),(1970,42,Chennai),…………….,(1950,22,bang)})

**grunt> DESCRIBE group\_env;**

**group\_env: { group: int , env : { year: int, temp: int, city: chararray}}**

**FOREACH…..GENERATE :**

The FOREACH…..GENERATE operator is used to act on every row in a relation. It can be used to remove fields or to generate new ones.

**grunt>max\_temp = FOREACH grouped\_env GENERATE group, MAX(env.temp);**

here FOREACH……GENERATE will generate the group and maximum temperature from the above grouped\_env bag.

**grunt>DUMP max\_temp;**

(1949,27)

(1950,30)

(1970,48)

(1949,35)

(1960,11)

**Null Validations:**

In SQL database , sql will enforce the constraints in a table’s schema at load time, for example, trying to load a string into a column that is declared to be a numeric type will fail.

In Pig, if the value can’t cast to the type declared in the schema, it will substitute a null value.

Let’s see how this will work out when we have the following input for the weather data which has ‘e’ character in the place of integer

**]$ cat > climate**

1949 e hyd

1950 30 bang

1970 42 chennai

1949 35 mumbai

1970 e chennai

1960 11 simla

1950 22 bang

**grunt>env = LOAD ‘/home/training/climate’**

**AS (year:int , temp:int , city:chararray);**

**grunt> DUMP env;**

(1949, ,hyd)

(1950,30,bang)

(1970,42,Chennai)

(1949,35,mumbai)

(1970, ,Chennai)

(1960,11,simla)

(1950,22,bang)

**ILLUSTRATE:**

With the ILLUSTRATE operator, Pig provides a tool for generating a reasonably complete

and concise dataset. Here is the output from running ILLUSTRATE

grunt> **ILLUSTRATE max\_temp;**

-------------------------------------------------------------------------------

| records | year:chararray | temperature:int | quality:int |

-------------------------------------------------------------------------------

| | 1949 | 27 | hyd |

| | 1950 | 30 | bang |

| | 1970 | 42 | chennai |

-------------------------------------------------------------------------------

--------------------------------------------------------------------------------------------

| grouped\_records | group:int | env:bag{:tuple(year:int, |

temperature:int,quality:int)} |

--------------------------------------------------------------------------------------------

| | 1949 | {(1949, 22, 1)} |

| | 1950 | {(1950,30,ban) | (1950,22,bang)}) |

--------------------------------------------------------------------------------------------

---------------------------------------------------

| max\_temp | group:int | :int |

---------------------------------------------------

| | 1949 | 27 |

| | 1950 | 30 |

---------------------------------------------------

**STORING DATA:**

Pig is an interface to process the data. Once we process the data, it’s better to store otherwise once we come out of grunt shell everything will be lost. So storing the data will work out with **STORE** command.

Here is an example to store the data

**grunt> STORE max\_temp into ‘MaxTemp’ using PigStorage(‘:’);**

**grunt> cat out;**

1949:27

1950:30

1970:48

1949:35

1960:11

If we are not using PigStorage, it will take tab delimiter in between fields by default as follows

**grunt> STORE max\_temp into ‘MaxTemp’ ;**

**grunt> cat out;**

1949 27

1950 30

1970 48

1949 35

1960 11

**Pig Latin relational operators:**

**Category** **Operator** **Description**

Loading and storing LOAD Loads data from the

filesystem or other storage into a relation

STORE Saves a relation to the

filesystem or other storage

DUMP Prints a relation to the

console

Filtering FILTER Removes unwanted rows

from a relation

DISTINCT Removes duplicate rows

from a relation

FOREACH...GENERATE Adds or removes fields from

a relation

MAPREDUCE Runs a MapReduce job using

a relation as input

STREAM Transforms a relation using

an external program

SAMPLE Selects a random sample of a

relation

Grouping and joining JOIN Joins two or more relations

COGROUP Groups the data in two or

more relations

GROUP Groups the data in a single

relation

CROSS Creates the cross-product of

two or more relations

Sorting ORDER Sorts a relation by one or

more fields

LIMIT Limits the size of a relation to

a maximum number of tuples

Combining and splitting UNION Combines two or more

relations into one

SPLIT Splits a relation into two or

more relations

**Pig Latin diagnostic operators:**

**Operator** **Description**

DESCRIBE Prints a relation’s schema

EXPLAIN Prints the logical and physical plans

ILLUSTRATE Shows a sample execution of the logical plan, using a generated subset of the input

**Category Command Description**

Hadoop Filesystem

cat Prints the contents of one or more files

cd Changes the current directory

copyFromLocal Copies a local file or directory to a Hadoop filesystem

copyToLocal Copies a file or directory on a Hadoop filesystem to the local filesystem

cp Copies a file or directory to another directory

fs Accesses Hadoop’s filesystem shell

ls Lists files

mkdir Creates a new directory

mv Moves a file or directory to another directory

pwd Prints the path of the current working directory

rm Deletes a file or directory

rmf Forcibly deletes a file or directory (does not fail if the file or directory does not exist)

Hadoop MapReduce

kill Kills a MapReduce job

Utility exec Runs a script in a new Grunt shell

Help Shows the available commands and options

quit Exits the interpreter

run Runs a script within the existing Grunt shell

set Sets Pig options and MapReduce job properties

sh Run a shell command from within Grunt

The file system commands can operate on files or directories in any Hadoop filesystem, and they are very similar to the hadoop fs commands (which is not surprising, as both are simple wrappers around the Hadoop FileSystem interface). You can access all of the Hadoop filesystem shell commands using Pig’s fs command. For example, fs -ls will show a file listing, and fs -help will show help on all the available commands.

**Grouping and Joining Data:**

GROUP command will work on single Bag.The GROUP command will group the data in a single relation. GROUP creates a relation whose first field is the grouping field, which is given the alias group. The second field is a bag containing the grouped fields with the same schema as the original relation.

Where as COGROUP will work on more than one bag to group similar tuples into a single Bag on a specified alias . The COGROUP statement is similar to JOIN, but instead creates a nested set of output tuples.

The COGROUP generates a tuple for each unique grouping key. The first field of each tuple is the key, and the remaining fields are bags of tuples from the relations with a matching key.

If for a particular key a relation has no matching key, the bag for that relation is empty.

COGROUP is similar to FULL OUTER JOIN

Lets work with an example.

**]$ cat > sales**

1 a hyd

2 b Chennai

3 c delhi

6 d hyd

**]$ cat > goods**

1 g10

5 g20

4 g30

3 g40

Load above files into two bags….

Grunt shell with local mode….

**grunt> sales = LOAD ‘/home/training/sales’ as (sid:int, sname:int, city:chararray)**

**grunt> DUMP sales;**

(1,a,hyd)

(2,b,Chennai)

(3,c,delhi)

(6,d,hyd)

**grunt> goods = LOAD ‘/home/training/goods’ as (sid:int, gname:chararray)**

**grunt> DUMP goods;**

(1,g10)

(5,g20)

(4,g30)

(3,g40)

**grunt> c = COGROUP sales by sid , goods by sid;**

(1,{{(1,a,hyd), (6,d,hyd)},{(1,g10)}})

(2,{{(2,b,Chennai)},{}})

(3,{{(3,c,delhi)},{(3,g40)}})

(4,{{},{(4,g30)}})

(5,{{},{(5,g20)}})

(6,{{(6,d,hyd)},{}})

From the above example it groups all the tuples into a single bag with the alias sid…. So Here COGROUPING is similar to FULL OUTER JOIN except grouping field. Means,. From both the bags, exactly matched tuples will be grouped , unmatched tuples will be given as null from left bag and even null from right bag.

**grunt> c1 = COGROUP sales by sid INNER, goods by sid;**

(1,{{(1,a,hyd), (6,d,hyd)},{(1,g10)}})

(2,{{(2,b,Chennai)},{}})

(3,{{(3,c,delhi)},{(3,g40)}})

(6,{{(6,d,hyd)},{}})

From the above example, we have used INNER command for the left bag, then entire left bag will be grouped, from right bag it will group all matched tuples and unmatched tuples will be grouped as null or empty. Here this is very much similar to LEFT OUTER JOIN except grouping.

**grunt> c2 = COGROUP sales by sid , goods by sid INNER;**

(1,{{(1,a,hyd), (6,d,hyd)},{(1,g10)}})

(3,{{(3,c,delhi)},{(3,g40)}})

(4{{},{(4,g30)}})

(5{{},{(5,g20)}})

From the above example , if we can use INNER for right bag then, entire right bag will be grouped , from left bag it will group matched tuples and unmatched tuples will be grouped as null. Here this is very much similar to RIGHT OUTER JOIN except grouping.

**grunt> c = COGROUP sales by sid INNER, goods by sid INNER;**

(1,{{(1,a,hyd), (6,d,hyd)},{(1,g10)}})

(3,{{(3,c,delhi)},{(3,g40)}})

From the above example , if we can use INNER for both the bags , then all matched tuples will be grouped from both the bags and it won’t group unmatched tuples. This is very much similar to INNER JOIN except grouping.

]

**JOINING DATA:**

**Inner Join**

**grunt> c = JOIN sales by sid , goods by sid ;**

(1,a,hyd,1,g10)

(3,c,delhi,3,g40)

This is inner join , so it gives exactly matched tuples from both the bags

**Outer Join:**

**Left outer join:**

**grunt> c = JOIN sales by sid LEFT , goods by sid ;**

(1,a,hyd,1,g10)

(2,b,Chennai,,)

(3,c,delhi,3,g40)

(6,d,hyd,,)

Here left outer join will give entire left bag but it gives exactly matched tuples from right bag and unmatched tuples will be given as null.

**Right outer join:**

**grunt> c = JOIN sales by sid RIGHT , goods by sid ;**

(1,a,hyd,1,g10)

(3,c,delhi,3,g40)

(,,,4,g30)

(,,,g20)}})

Here right outer join will give entire left bag but it gives exactly matched tuples from left bag and unmatched tuples will be given as null

**Full outer join:**

**grunt> c = JOIN sales by sid FULL, goods by sid;**

(1,a,hyd,1,g10)

(2,b,Chennai,,)

(3,c,delhi,3,g40)

(,,,4,g30)

(,,,5,g20)

(6,d,hyd,,)

Here it give both the bags , exactly matched tuples will be given as it is but unmatched tuples will be given as null from left bag and unmatched tuples will be given as null from right bag.

**Choosing Between Pig and Hive:**

**Typically, organizations wanting an abstraction on top of standard MapReduce will choose to use either Hive or Pig**

**Which one is chosen depends on the skillset of the target users**

– Those with an SQL background will naturally gravitate towards

Hive

– Those who do not know SQL will often choose Pig

**Each has strengths and weaknesses; it is worth spending some time investigating each so you can make an informed decision**

**Some organizations are now choosing to use both**

– Pig deals better with less-structured data, so Pig is used to

manipulate the data into a more structured form, then Hive is used to query that structured data.

**HBASE**

We live in an era in which we are all connected over the Internet and expect to find

results instantaneously, whether the question

Because of this, companies have become focused on delivering more targeted information, such as recommendations or online ads, and their ability to do so directly

influences their success as a business. Systems like *Hadoop*† now enable them to gather

and process petabytes of data, and the need to collect even more data continues to

increase with, for example, the development of new machine learning algorithms.

Where previously companies had the liberty to ignore certain data sources because

there was no cost-effective way to store all that information, they now are likely to lose

out to the competition. There is an increasing need to store and analyze every data point

they generate. The results then feed directly back into their e-commerce platforms and

may generate even more data.

HBase is a distributed column-oriented database built on top of HDFS. HBase is the Hadoop application to use when you require real-time read/write random-access to very large datasets.

Joins, complex queries, triggers, views, and foreign-key constraints become prohibitively expensive to run on a scaled RDBMS or do not work at all.

HBase is not relational and does not support SQL, but given the proper problem space, it is able to do what an RDBMS cannot: host very large, sparsely populated tables on clusters made from commodity hardware.

The HBase project was started toward the end of 2006 by Chad Walters and Jim Kellerman at Powerset. It was modeled after Google’s “Bigtable: A Distributed Storage System for Structured Data”.

The first HBase release was bundled as part of Hadoop 0.15.0 in October 2007. In May 2010, HBase graduated from a Hadoop subproject to become an Apache Top Level Project. Production users of HBase include Adobe, StumbleUpon, Twitter, and groups at Yahoo!.

**Column oriented DataBases:**

Column oriented databases save their data grouped by columns. Subsequent column values are stored contiguously on disk. This differs from usual row-oriented approach of traditional databases which store entire rows contiguously.

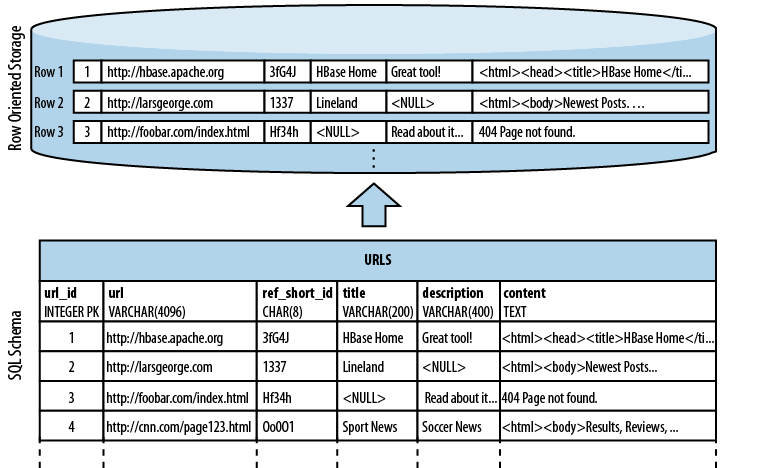
The reason to store values on a per-column basis instead is based on the assumption

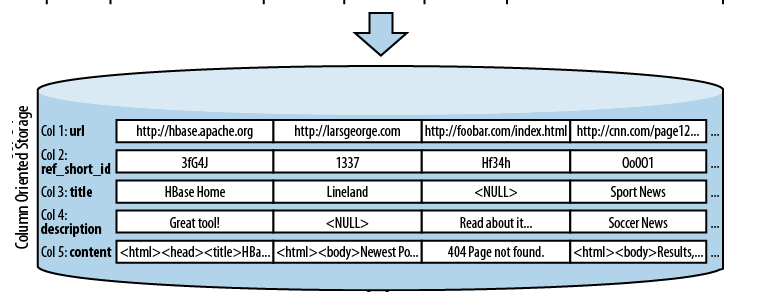
that, for specific queries, not all of the values are needed. This is often the case in

analytical databases in particular, and therefore they are good candidates for this different

storage schema.

**Sql schema:**

****

****

**Data model:**

* There are many variations in how the data is stored, which include key/value stores
* (compare to a HashMap), semistructured, column-oriented stores, and documentoriented
* stores. How is your application accessing the data? Can the schema evolve over time?

(Table, Rowkey,Family,column,Timestamp) -----------> value

* Table rowkeys are byte arrays , these sorted by row keys, a tables primary key. The sort is byte ordered. All table accesses via the table primary key.
* Row columns are grouped into column familys. All column family members have a prefix.
* Ex:

Temperature: air

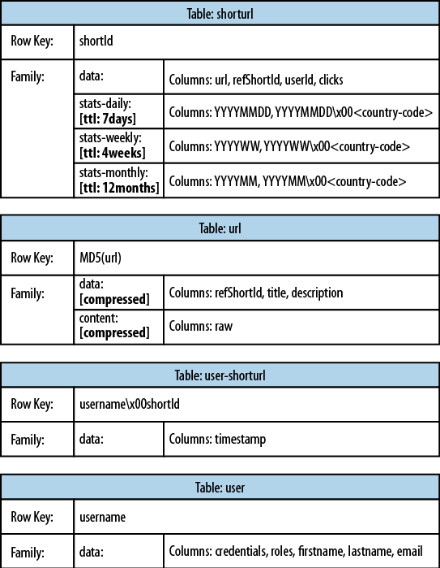
Temperature: der

Station :id

* Physically all column family members are stored together on the filesystem. So we can specifically call HBASE as column-family oriented DB.
* In RDBMS because of fixed schema we added null where there is no value. But for HBASE’s storage architecture you simply omit the whole column. Here nulls don’t occupy any space.

**Regions:**

* Tables are automatically partitioned into regions. Each region comprises of a subset of table’s rows.
* A region is denoted by the table it belongs to , its first row inclusive, and last row exclusive.
* Initially a table comprises a single region , but as the size of region grows, after a threshold , it splits a row boundary into 2 new regions of equal size.
* Each region is exactly served by a region server and it can serve may regions at any time .
* Regions are units that get distributed over an HBase cluster.
* In this way the table is too big for any one server can be carried by a cluster of servers with each node hosting a subset of total tables regions.

****

**Basics of HBase:**

**Start hbase master service**



**Start hbase shell**



1.create table named test with a single column family name data using defaults for table and columnfamily attributes.

**hbase(main)> create ‘test’ , ‘data’**

**0 rows in 1.3066**

2.to list the tables

**hbase(main)> list**

**Test**

3.inserting data into three different rows and columns in the data column family and then list table context.

hbase(main) > put ‘test’ , ‘row1’ , ‘data:1’ , ‘value1’

hbase(main) > put ‘test’ , ‘row2’ , ‘data:2’ , ‘value2’

hbase(main) > put ‘test’ , ‘row3’ , ‘data:3’ , ‘value3’

To list the table content

**hbase (main) > scan ‘test’**

ROW COLUMN+CELL

Row1 column=data:1 timestamp=122--- , value=value1

Row2 column=data:2 timestamp=123--- , value=value2

Row3 column=data:3 timestamp=124--- , value=value3

4.to remove table we need to disable it and then drop table.

**hbase (main) > disable ‘test’**

**hbase (main) > drop ‘test’**

**note: the above operations we can do with java client also.**

**Program:**

**Basic table administration and access.**

**import** org.apache.hadoop.conf.Configuration;

**import** org.apache.hadoop.hbase.HBaseConfiguration;

**import** org.apache.hadoop.hbase.HColumnDescriptor;

**import** org.apache.hadoop.hbase.HTableDescriptor;

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

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

**import** org.apache.hadoop.hbase.client.Put;

**import** org.apache.hadoop.hbase.client.Scan;

**import** org.apache.hadoop.hbase.client.ResultScanner;

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

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

**import** java.io.IOException;

**public** **class** MakeTable {

**public** **static** **final** **byte**[] *TABLE\_NAME* = "USER".getBytes();

**public** **static** **final** **byte**[] *COLFAM\_NAME* = "INFO".getBytes();

**public** **static** **final** **byte**[] *COL\_VALUE* = "NAME".getBytes();

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

InterruptedException {

Configuration conf = HBaseConfiguration.*create*();

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

**if**(admin.tableExists(*TABLE\_NAME*)) {

admin.disableTable(*TABLE\_NAME*);

admin.deleteTable(*TABLE\_NAME*);

}

HTableDescriptor desc = **new** HTableDescriptor(*TABLE\_NAME*);

HColumnDescriptor coldef = **new** HColumnDescriptor(*COLFAM\_NAME*);

desc.addFamily(coldef);

coldef.setMaxVersions(1);

admin.createTable(desc);

HTable userTable = **new** HTable(conf, *TABLE\_NAME*);

Put row1 = **new** Put(Bytes.*toBytes*("42"));

row1.add(*COLFAM\_NAME*, *COL\_VALUE*, Bytes.*toBytes*("Diana"));

Put row2 = **new** Put(Bytes.*toBytes*("43"));

row2.add(*COLFAM\_NAME*, *COL\_VALUE*, Bytes.*toBytes*("Doug"));

Put row3 = **new** Put(Bytes.*toBytes*("44"));

row3.add(*COLFAM\_NAME*, *COL\_VALUE*, Bytes.*toBytes*("Steve"));

userTable.put(row1);

userTable.put(row2);

userTable.put(row3);

admin.flush(*TABLE\_NAME*);

Scan userScan = **new** Scan();

ResultScanner scanner = userTable.getScanner(userScan);

**for** (Result result : scanner ) {

System.*out*.println(Bytes.*toString*(result.getValue(*COLFAM\_NAME*,

*COL\_VALUE*)));

}

userTable.close();

}

}

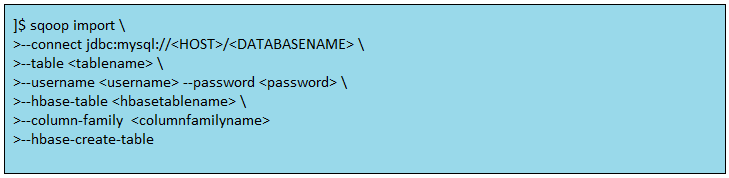
Executing hbase jar file

$hadoop jar hbase.jar MakeTable

* org.apache.hadoop.hbase.**HBaseConfiguration** class instance will return Configuration that read hbase configuration from hbase-site.xml and hbase-default.xml.
* This configuration we will subsequently used to create instances of **HBaseAdmin** and **HTable**.
* **HBaseAdmin** is used for administering your hbase cluster for adding and dropping tables.
* HTable is used to access a specific table.
* To create a table we need to first create an instance of **HBaseAdmin** and then ask to create table. In our example we are asking to create table named ‘test’ with single column family named ‘data’.
* Use methods on org.apache.hadoop.**HTableDescriptor** and **HColumnDescriptor** to change table schema.
* Operations on table we require instance of HTable passing it our configuration instance and name of the table.
* Here use **put** to put a single cell value of ‘value1’ into row ‘row1’ on column named ‘data:1’
* Get is used to get the cell values.
* Scan is used to scan over the table created.
* Finally if you want to drop the table , first disable it and then delete it.

**Importing Data from RDBMS to HBase**

--------------------------------------------------

****

**HBase Hive integration – Querying HBase by Hive**

### Create HBase table

|  |  |
| --- | --- |
|  | Hbase (main) > create 'hivehbase', 'ratings'  Hbase (main)> put 'hivehbase', 'row1', 'ratings:userid', 'user1'  Hbase (main)> put 'hivehbase', 'row1', 'ratings:bookid', 'book1'  Hbase (main)> put 'hivehbase', 'row1', 'ratings:rating', '1'   Hbase (main)> put 'hivehbase', 'row2', 'ratings:userid', 'user2'  Hbase (main)> put 'hivehbase', 'row2', 'ratings:bookid', 'book1'  Hbase (main)> put 'hivehbase', 'row2', 'ratings:rating', '3'   Hbase (main)> put 'hivehbase', 'row3', 'ratings:userid', 'user2'  Hbase (main)> put 'hivehbase', 'row3', 'ratings:bookid', 'book2'  Hbase (main)> put 'hivehbase', 'row3', 'ratings:rating', '3'    Hbase (main)> put 'hivehbase', 'row4', 'ratings:userid', 'user2'  Hbase (main)> put 'hivehbase', 'row4', 'ratings:bookid', 'book4'  Hbase (main)> put 'hivehbase', 'row4', 'ratings:rating', '1' |

### Provide necessary jars to Hive

Create a folder named *auxlib* in Hive root directory and put the following jars in it. The jars can be found in HBase lib directory. Hive-HBase handler is present in Hive lib directory.

* Guava
* Hive-Hbase handler
* HBase
* Zookeeper

### Create Hive external table

Since we want the data to be fetched from HBase we would need to create an external table for Hive.

**CREATE EXTERNAL TABLE hbasehive\_table**

**(key string, userid string,bookid string,rating int)**

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

**WITH SERDEPROPERTIES**

**("hbase.columns.mapping" = ":key,ratings:userid,ratings:bookid,ratings:rating")**

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

**Oozie**

**Introduction:**

**The Motivation for Oozie:**

**Many problems cannot be solved with a single MapReduce job**

**Instead, a *workflow* of jobs must be created**

**Simple workflow:**

– Run JobA

– Use output of JobA as input to JobB

– Use output of JobB as input to JobC

– Output of JobC is the final required output

**Easy if the workflow is linear like this**

– Can be created as standard Driver code

**If the workflow is more complex, Driver code becomes much more difficult to maintain**

**Example: running multiple jobs in parallel, using the output from all of those jobs as the input to the next job**

**Example: including Hive or Pig jobs as part of the workflow**

**What is Oozie?**

**Oozie is a ‘workflow engine’ runs on a server** typically outside the cluster

**Runs workflows of Hadoop jobs**

– Including Pig, Hive, Sqoop jobs

– Submits those jobs to the cluster based on a workflow definition

**Workflow definitions are submitted via HTTP**

**Jobs can be run at specific times**

– One-off or recurring jobs

**Jobs can be run when data is present in a directory**

**Oozie Workflow Basics**

**Oozie workflows are written in XML**

**Workflow is a collection of actions**

– MapReduce jobs, Pig jobs, Hive jobs etc.

**A workflow consists of *control flow nodes* and *action nodes***

**Control flow nodes define the beginning and end of a workflow**

– They provide methods to determine the workflow execution path

– Example: Run multiple jobs simultaneously

**Action nodes trigger the execution of a processing task, such as**

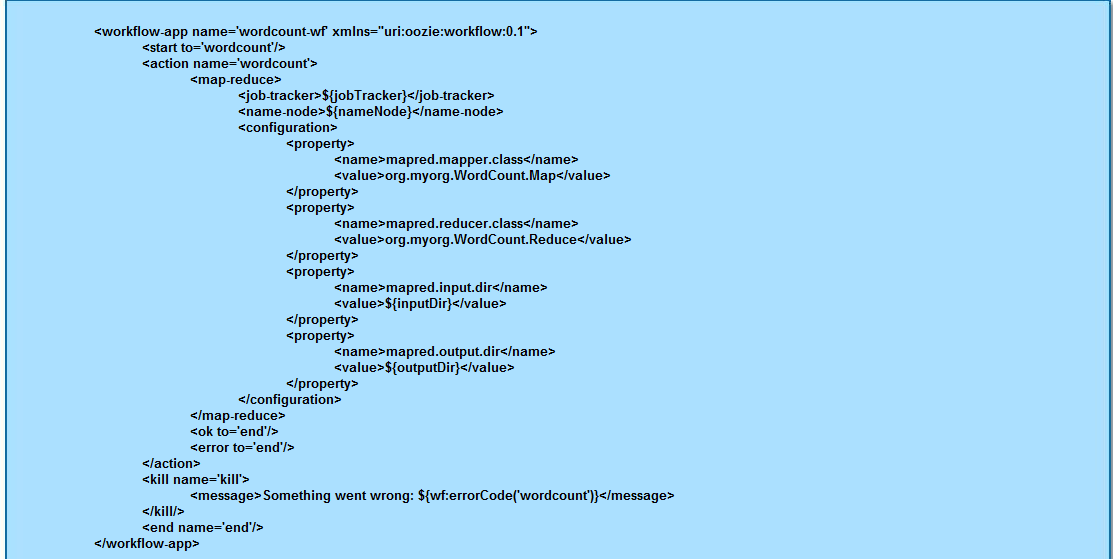
– A MapReduce job

– A Pig job

– A Sqoop data import job

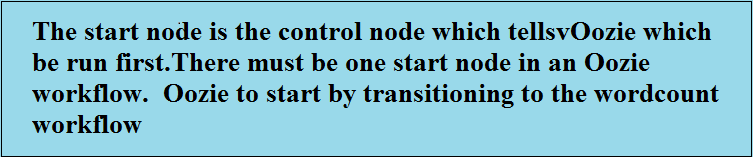
**Simple Oozie Example**

**Simple example workflow for WordCount:**

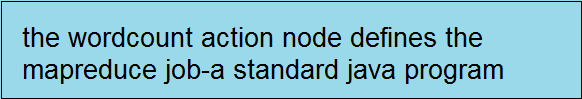
****

**<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">**

**<start to='wordcount'/>**

****

**<action name='wordcount'>**

****

**<map-reduce>**

**<job-tracker>${jobTracker}</job-tracker>**

**<name-node>${nameNode}</name-node>**

**<configuration>**

**<property>**

**<name>mapred.mapper.class</name>**

**<value>org.myorg.WordCount.Map</value>**

**</property>**

**<property>**

**<name>mapred.reducer.class</name>**

**<value>org.myorg.WordCount.Reduce</value>**

**</property>**

**<property>**

**<name>mapred.input.dir</name>**

**<value>${inputDir}</value>**

**</property>**

**<property>**

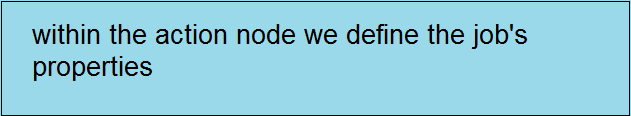
**<name>mapred.output.dir</name>**

**<value>${outputDir}</value>**

**</property>**

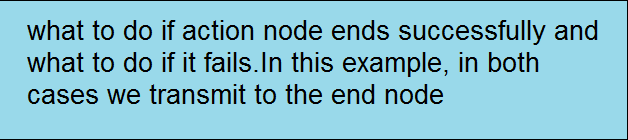
**</configuration>**

**</map-reduce>**

****

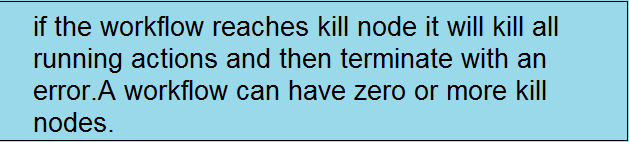
**<ok to='end'/>**

**<error to='end'/>**

****

**</action>**

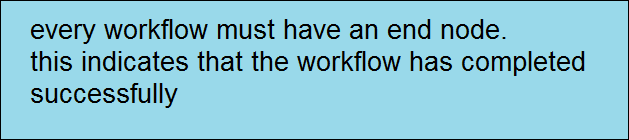
**<kill name='kill'>**

****

**<message>Something went wrong: ${wf:errorCode('wordcount')}</message>**

**</kill/>**

**<end name='end'/>**

****

**</workflow-app>**

**Other Oozie Control Nodes**

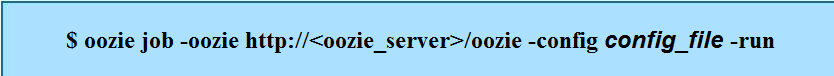
**A decision control node allows Oozie to determine the workflow execution path based on some criteria** Similar to a switch-case statement

**fork and join control nodes split one execution path into multiple execution paths which run concurrently**

* fork splits the execution path
* join waits for all concurrent execution paths to complete before proceeding
* fork and join are used in pairs

**Submitting an Oozie Workflow**

To submit an Oozie workflow using the command-line tool:

****

**Oozie can also be called from within a Java program**

Via the Oozie client API

**Hands on exercises:**

**1. Change the directories to the oozie-labs directory within the exercises directory**

**2. Start the oozie server**



**3. Change directories to lab1-java-mapreduce/job**



**4. Inspect the contents of the job.properties and workflow.xml files.you will see that this is**

**Our standard wordcount job.**

**5.change directories back to the main oozie-labs directory**



**6.we have provided a simple shell to submit the oozed workflow. Inspect run.sh**



**7.submit the workflow to the oozie server**



**8.inspect the progress of the job**



**9.When the job has completed , inspect HDFS to confirm that the output has been produced expected.**

**10.repeat the above procedure for lab2-sort-wordcount. Notice when you inspect workflow.xml that this workflow includes two MapReduce jobs which run one after other. When tou inspect the output in HDFS you will see that the second job sorts the output of the job into descending numerical order.**

**Software installation process:**

**Step 1:**

Choose the virtualization s/w that suites your operating system.

1. Vmplayer/Virtualbox for Window os/Linux os

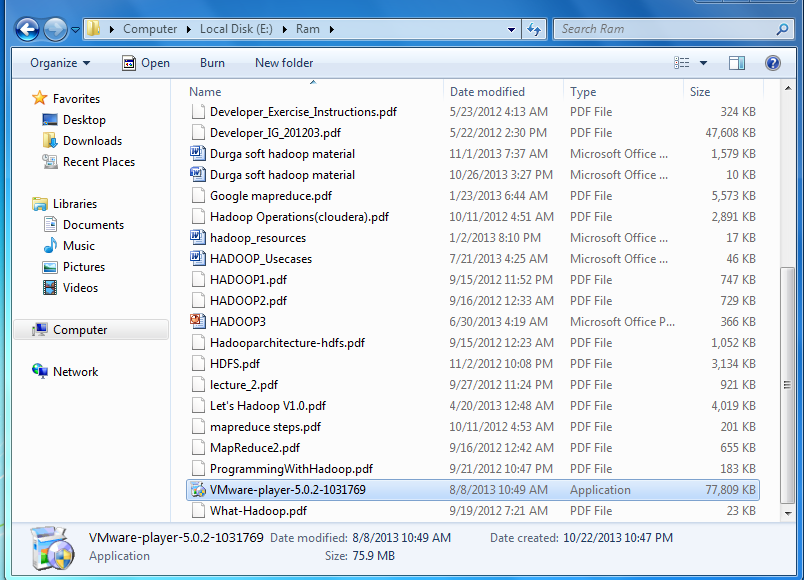
Refer the link below for vmplayer

https://my.vmware.com/web/vmware/free#desktop\_end\_user\_computing/vmware\_player/5\_0

2.VmFusion for MAC os

Go to the location in which drive you have saved your **vmware-player.**

Choose vmware-player and double click on it



**Step 2:**

Once you are double clicking on that it will ask you to choose either “yes” or “no”.

Choose “yes”



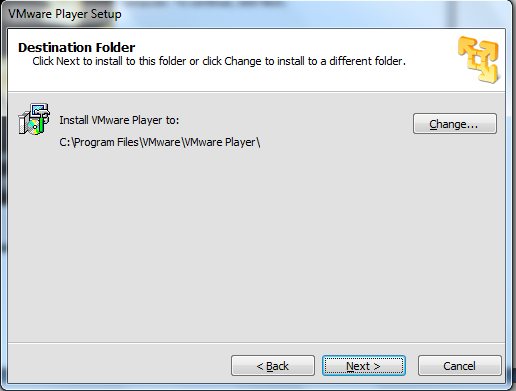
**Step 3:**

After the completion of the above process we can get below table then choose ----> Next

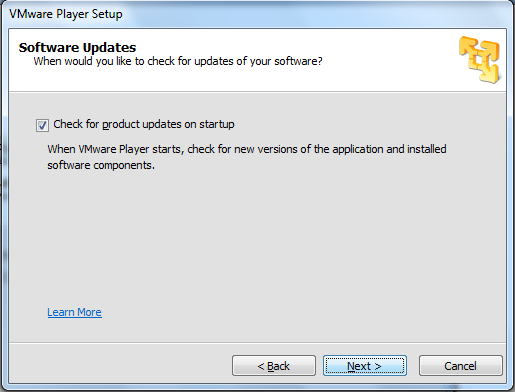


**Step 4:**

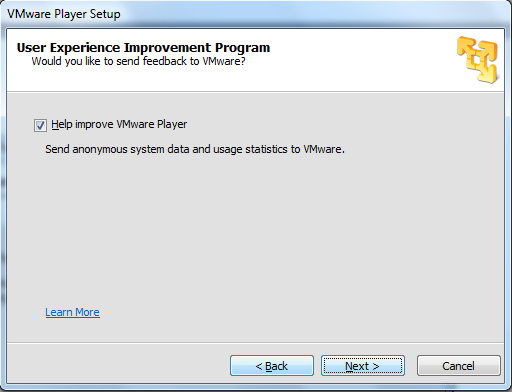
Choose browing location and say ------> Next



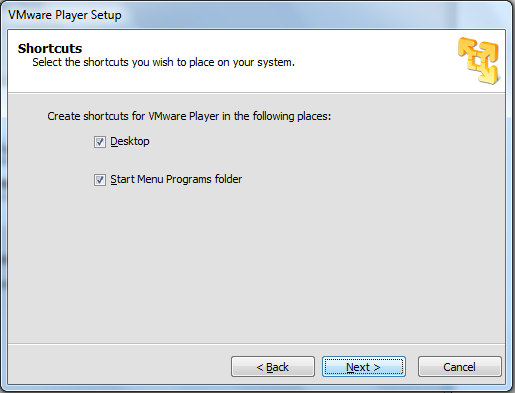
**Step 5:**



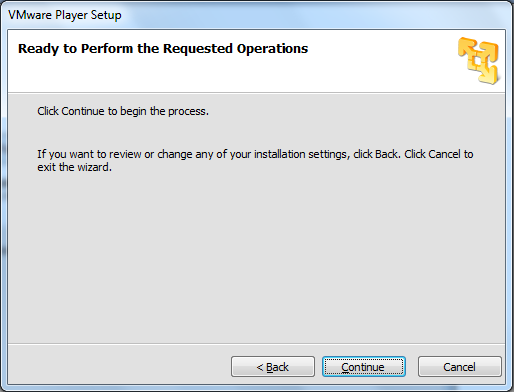
**Step 6:**



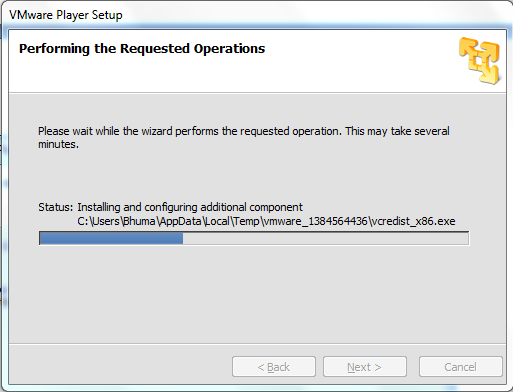
**Step 7:**



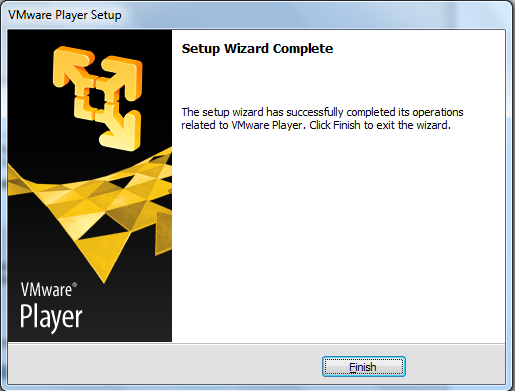
**Step 8:**



**Step 9:**



**Step 10:**



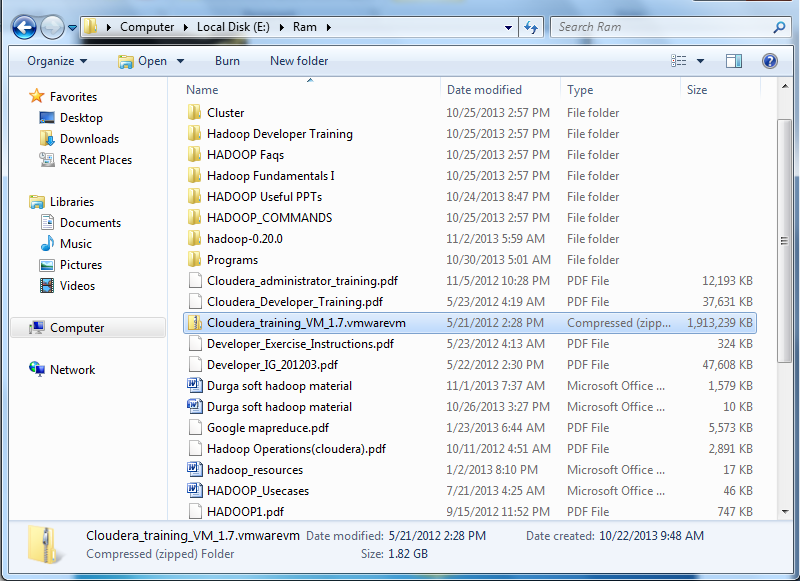
**Step 11:**

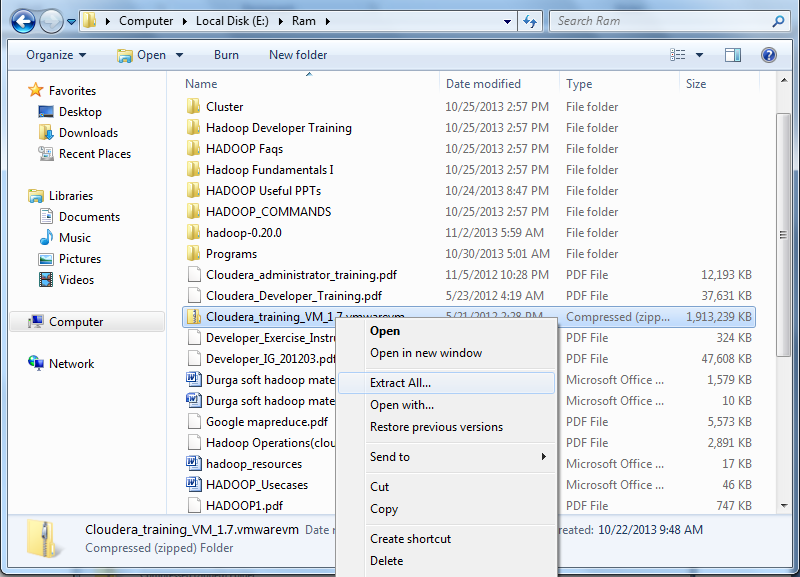
You will see an icon “vmWare-player” icon on Desktop.

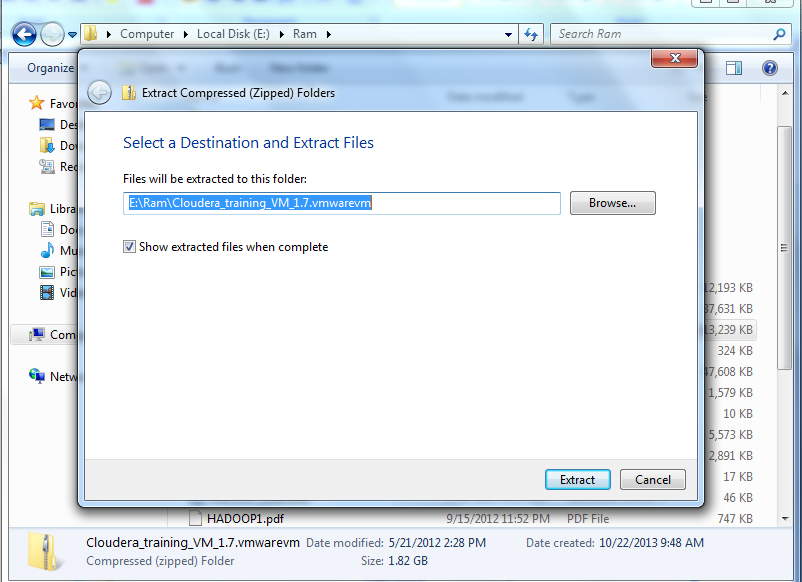


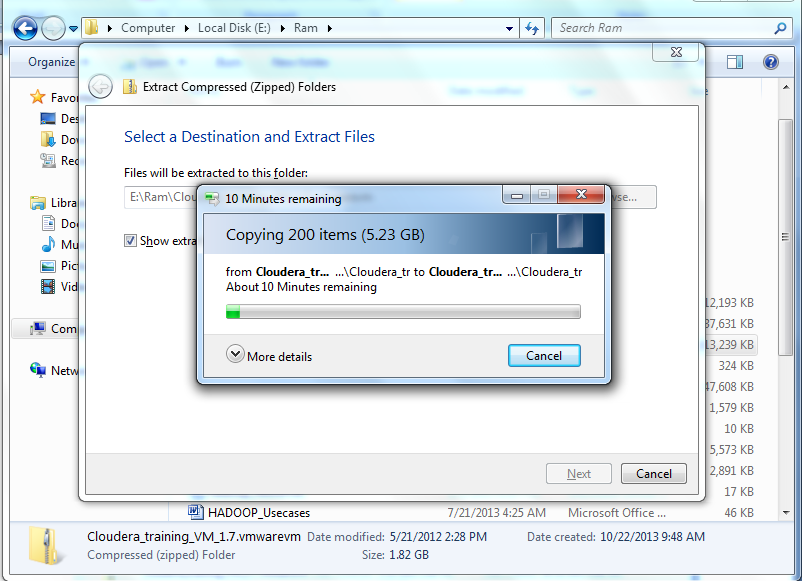
**Step 12:**

Extract “cloudera\_training\_VM\_1.7.vmwarevm” template zip file and right click on that and choose Extract All



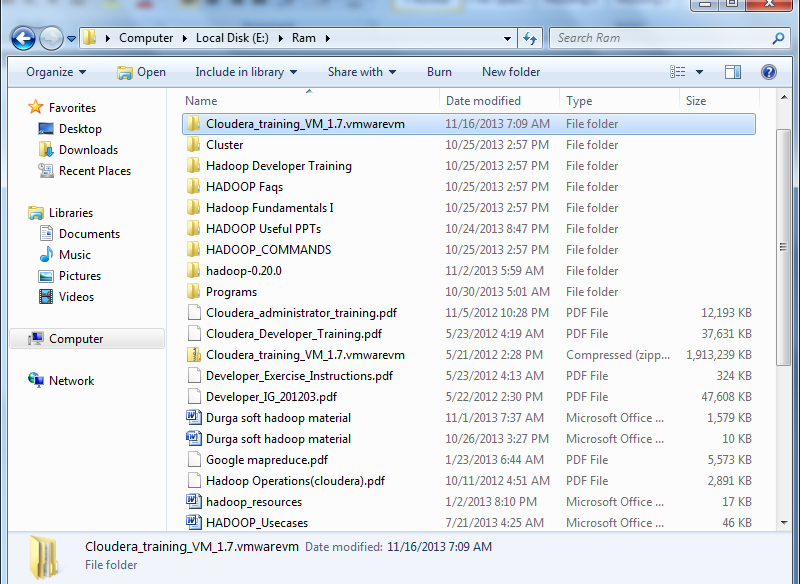






**Step 13:**

After extracting all files you will see “cloudera\_training\_VM\_1.7.vmware” folder



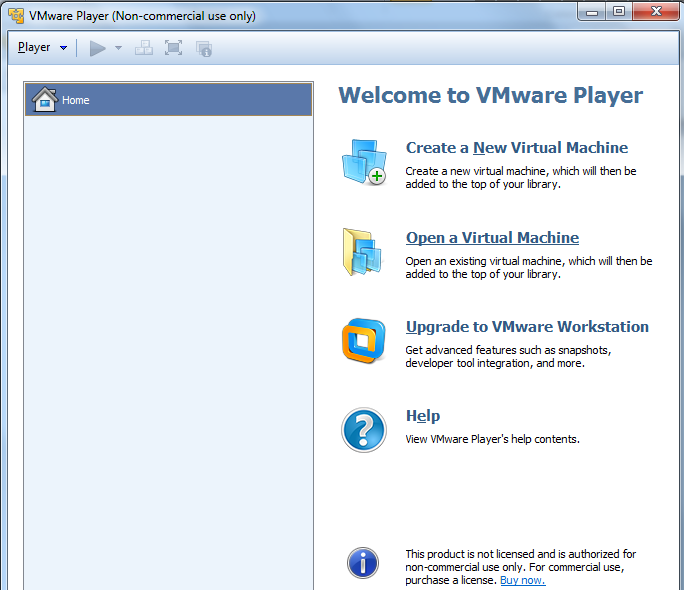
**Step 14:**

Now go to Desktop and open **“VmWare player”** icon by double clicking on it…….



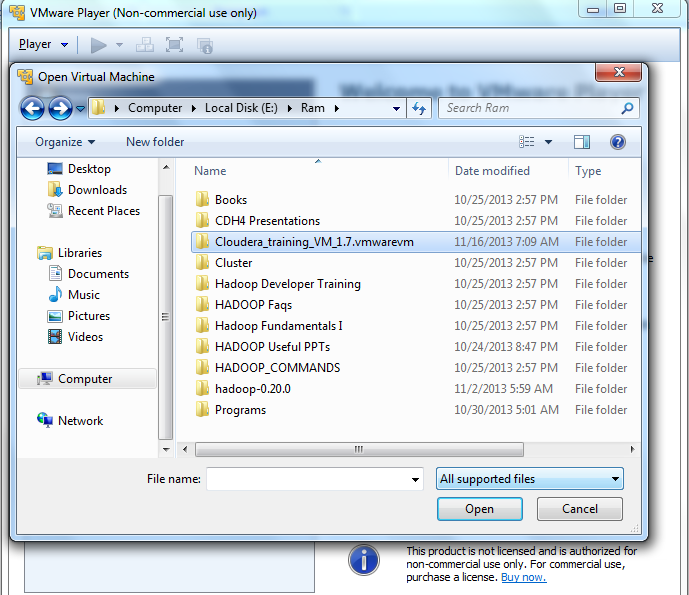
**Step 15:**

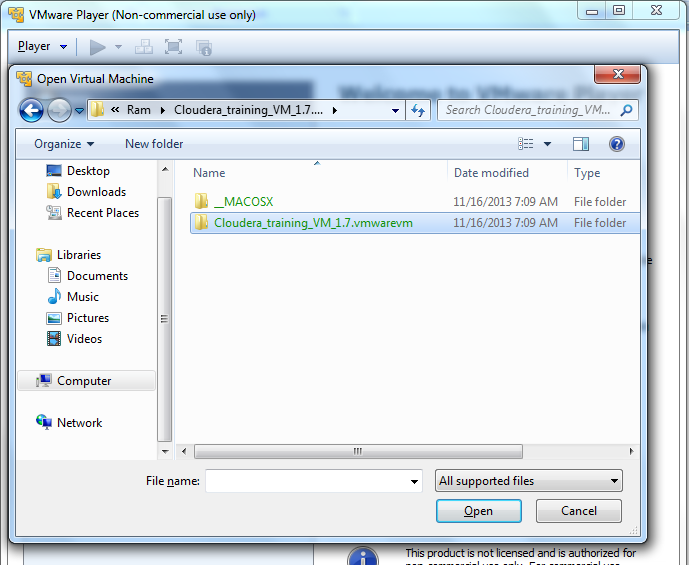
Choose **“Open a Virtual Machine”** option on right side to your vm (second option)

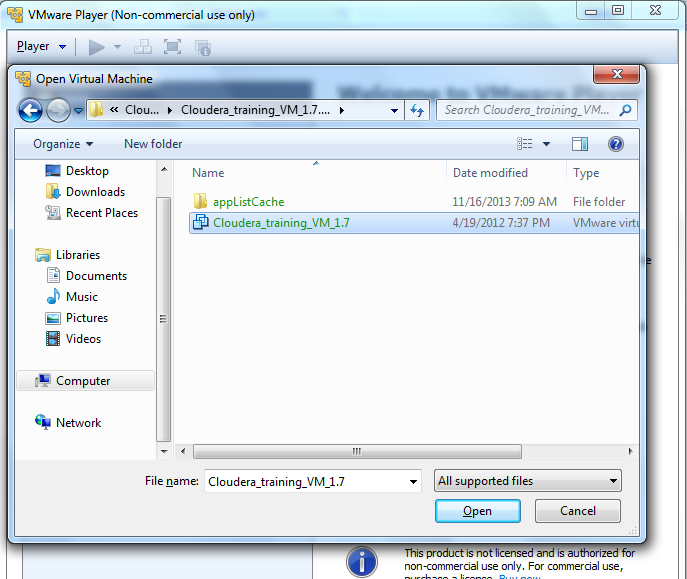


**Step 16:**

Browse the location where you have extracted “**Cloudera\_Training\_VM.1.7.vmware.”**

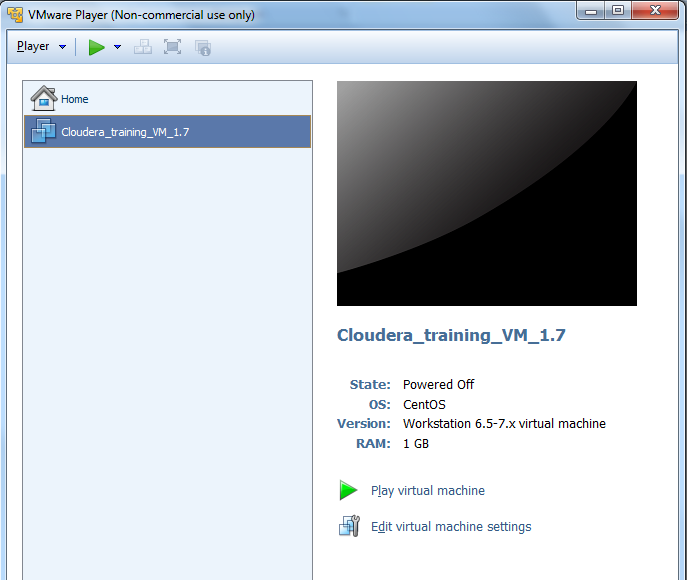


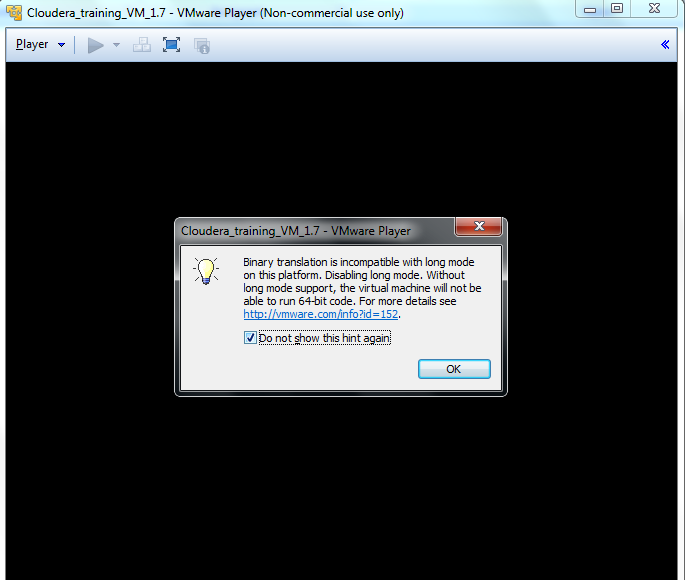


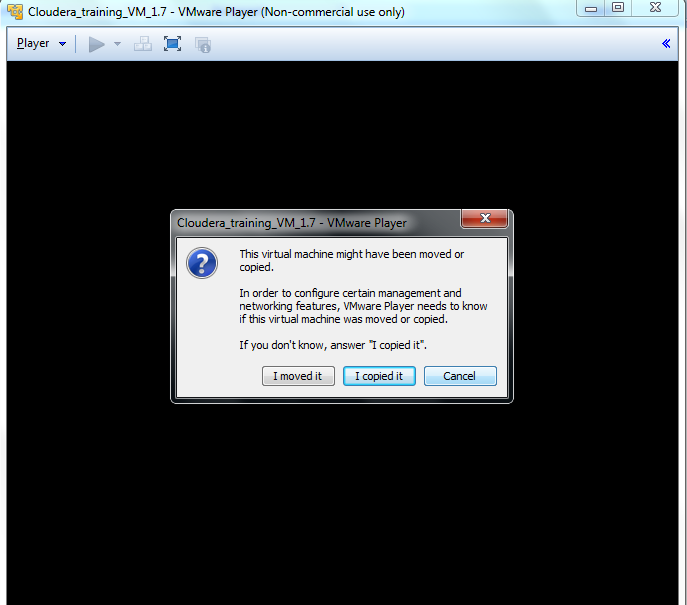


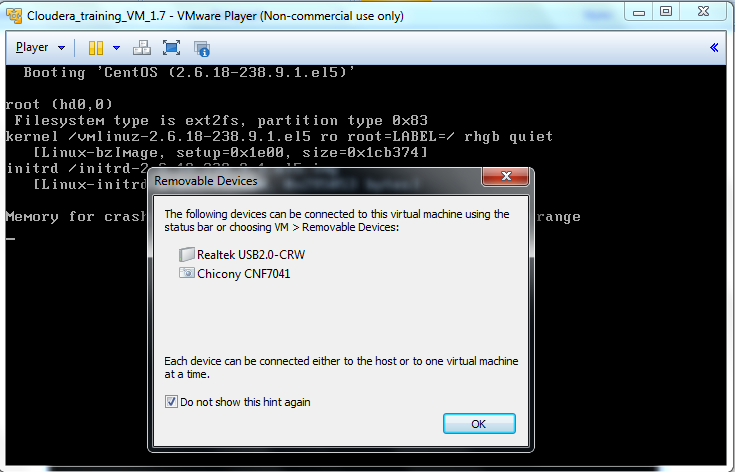
**Step 17:**

After completing the browsing you can see the **“Cloudera\_Training\_VM\_1.7”** in vmware player and doble click on that



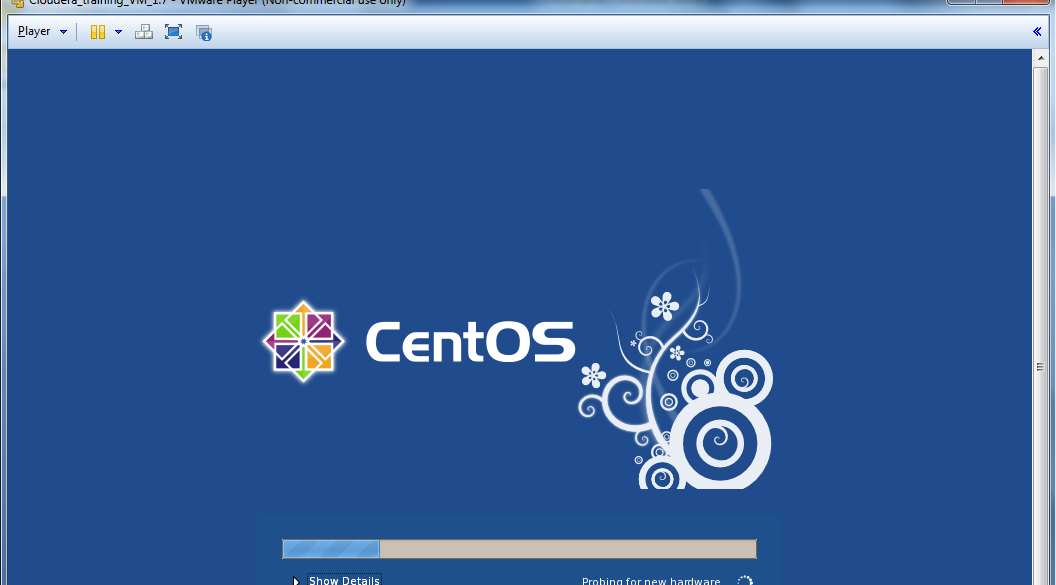


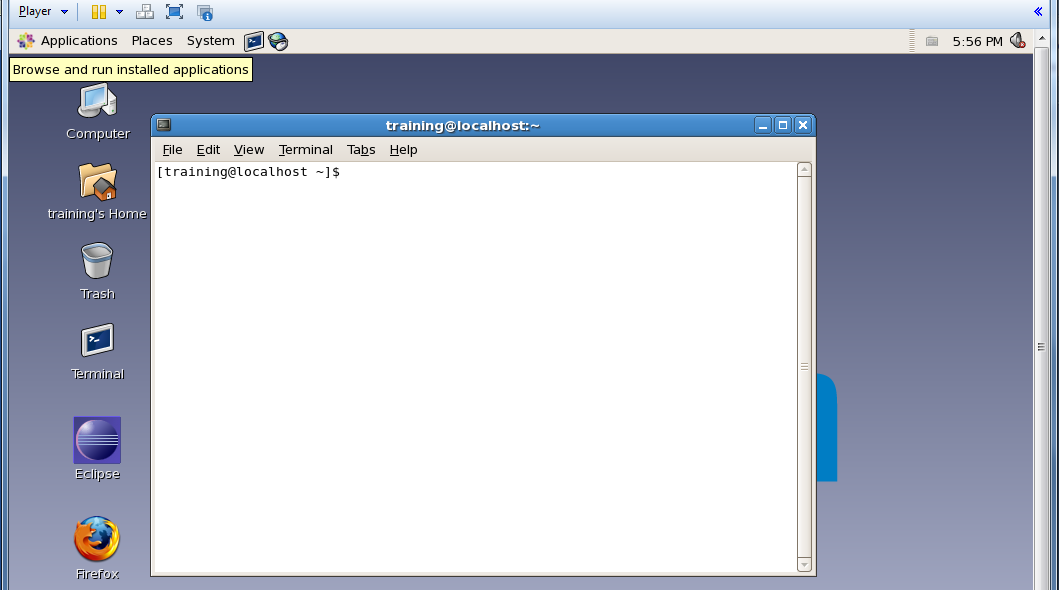




**Step 18:**

Don’t do any thing while it is under processing until it shows desktop icons





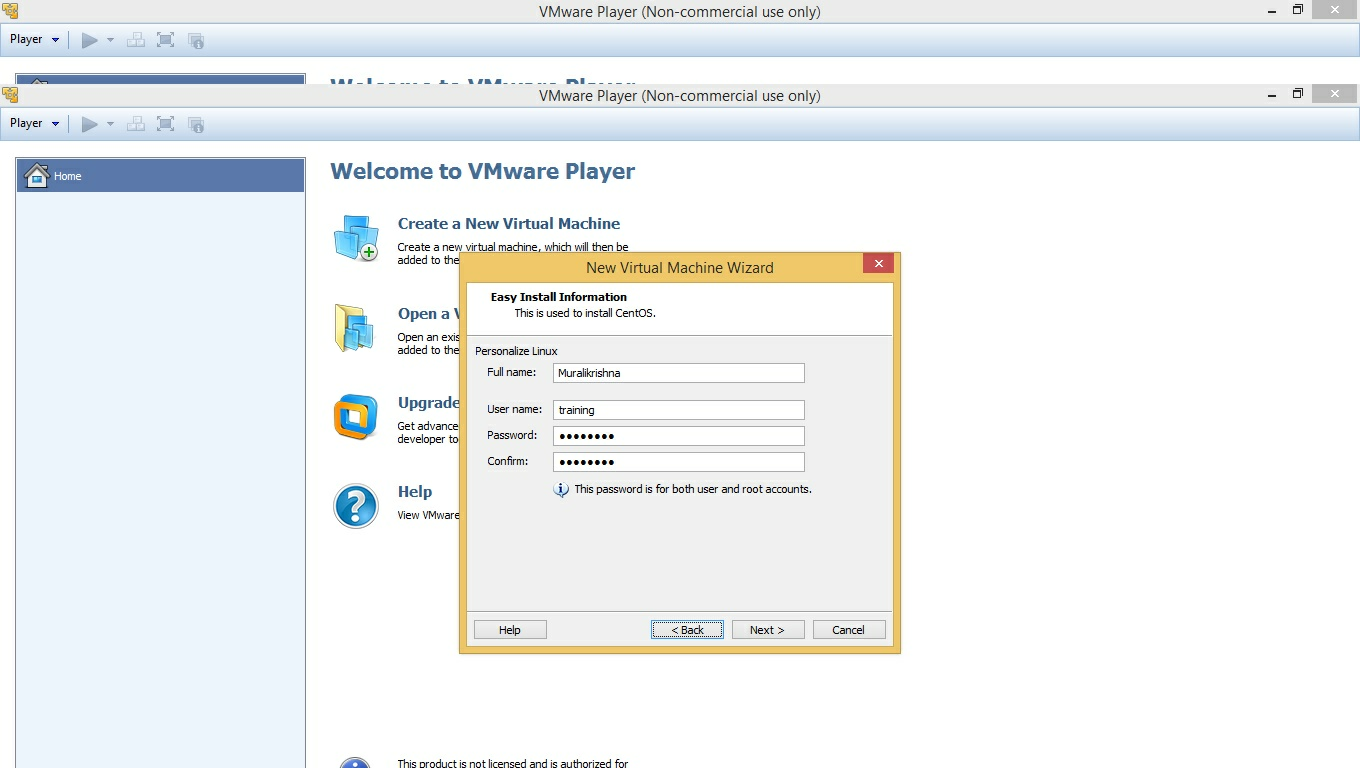
**Single Node Cluster:**

1).Install centos iso image in vm workstation/vmplayer.

a).Double click on vmplayer icon it will open vm

b).Click Create a New Virtual Machine and give the location of centos iso image.

c).Click on next and give the full name and username for example training and password and click on next.



c).click on next->next->next->finish

Now centos will be installed.

**Os level configurations for HADOOP**

**Root user # symbol,training user $ symbol**

From root user do the following operations

* Disable SELinux.

#vi /etc/selinux/config

* + Change SELINUX=enforcing to SELINUX=disabled.
* Disable iptable
  + # /etc/init.d/iptables stop
* Add your created user to sudoers file(in our example training)

$su it will ask root password

#vi /etc/sudoers

Add following line after root ALL=(ALL) ALL

training ALL=(ALL) NOPASSWD:ALL

2).Download java1.6 based on centos 32 bit or 64 bit

For 32 bit operation system

jdk-6u43-linux-i586-rpm.bin

3).Give executable permission to jdk-6u43-linux-i586-rpm.bin

$chmod 755 jdk-6u43-linux-i586-rpm.bin

4).Install java from root user (where java is there)

#./jdk-6u43-linux-i586-rpm.bin

5).export java home

#export JAVA\_HOME=/usr/java/jdk1.6.0\_43

6).Download required hadoop version

ex:hadoop-1.0.3.tar.gz

7).come to trainig user(training)

#su training

password:password

8).unzip hadoop tar file

$goto the location where hadoop downloaded

$tar -zxvf hadoop-1.0.3.tar.gz

9).making hadoop recoznize java

cd /hadoop/conf

$vi hadoop-env.sh

and add following line

export JAVA\_HOME=/usr/java/jdk1.6.0\_43

save and quit

10). Goto your home dir do following configurations for recognizing java,java jps and hadoop

$cd ~

$vi .bashrc

* Add following lines

export JAVA\_HOME=/usr/java/jdk1.6.0\_43

export HADOOP\_INSTALL=<hadoop installed location>

export PATH=$PATH:$HADOOP\_INSTALL/bin:$JAVA\_HOME/bin

11). Configuring HADOOP\_INSTALL directory to hadoop instalation directory

$export HADOOP\_INSTALL=<hadoop installed location>

note:

a).making update-alternatives working

add following line to .bashrc file in your home dir

export PATH=$PATH:/sbin:/usr/sbin:/usr/local/sbin

b).for ssh configuration

Switch to root user

$su

#cd goto root home dir

#ssh-keygen

enter

enter

enter

#cp .ssh/id\_rsa.pub .ssh/authorized\_keys

#chmod 700 .ssh

#chmod 640 .ssh/authorized\_keys

#chmod 600 .ssh/id\_rsa

#cp .ssh/authorized\_keys /home/training/.ssh/authorized\_keys

#cp .ssh/id\_rsa /home/training/.ssh/id\_rsa

#chown training /home/training/.ssh/authorized\_keys

#chown training /home/training/.ssh/id\_rsa

#chmod 700 /home/training/.ssh

#chmod 640 /home/training/.ssh/authorized\_keys

#chmod 600 /home/training/.ssh/id\_rsa

vi /etc/ssh/ssh\_config

StrictHostChecking no

PasswordAuthentication no

#service sshd restart

12).set all configurations

#vi /home/training/hadoop/hadoop-1.0.3/conf/core-site.xml

<property>

<name>fs.default.name</name>

<value>hdfs://localhost:8020</value>

</property>

#vi /home/training/hadoop/hadoop-1.0.3/conf/mapred-site.xml

<property>

<name>mapred.job.tracker</name>

<value>localhost:8021</value>

</property>

#vi /home/training/hadoop/hadoop-1.0.3/conf/hdfs-site.xml

<property>

<name>dfs.replication</name>

<value>1</value>

</property>

13). Format the namenode

$hadoop namenode -format

14).start all the services

$/home/training/hadoop/hadoop-1.0.3/bin/start-all.sh

15).Open browser and check weather the services start or not

Type in the browser http://localhost:50070 or <http://localhost:50030>

**HIVE Installation**

Step1**:** Download the required hive version from below site

<http://www.apache.org/dyn/closer.cgi/hive/>

Step2: unzip the downloaded hive

$**tar xzf hive-*x.y.z*-dev.tar.gz**

Step3: goto home directory

$cd

$vi .bashrc

**export HIVE\_INSTALL=/home/tom/hive-*x.y.z*-dev**

**export PATH=$PATH:$HIVE\_INSTALL/bin**

Step4: From command prompt do the following

$ **export HIVE\_INSTALL=/home/tom/hive-*x.y.z*-dev**

$ **export PATH=$PATH:$HIVE\_INSTALL/bin**

**INSTALLING ECLIPSE:**

Step1: change to root user and do the following

$su

Step2: Download eclipse

ex:eclipse-java-europa-winter-linux-gtk.tar.gz

#tar -zxvf eclipse-java-europa-winter-linux-gtk.tar.gz

Step2). change permissions for eclipse dir

#chmod -R +r /home/training/eclipse

Step3).Create Eclipse executable on /usr/bin path

#touch /usr/bin/eclipse

#chmod 755 /usr/bin/eclipse

Step4). Open eclipse file with your favourite editor

#vi /usr/bin/eclipse

#!/bin/sh

export ECLIPSE\_HOME="/home/training/eclipse"

$ECLIPSE\_HOME/eclipse $\*

Step4).bring eclipse icon on desktop

Create following file, with our favourite editor

#vi /usr/share/applications/eclipse.desktop

[Desktop Entry]

Encoding=UTF-8

Name=Eclipse

Comment=Eclipse SDK 4.3.1

Exec=eclipse

Icon=/home/training/eclipse/icon.xpm

Terminal=false

Type=Application

Categories=GNOME;Application;Development;

StartupNotify=true

Step5).Go to Applications menu->programming->right click on eclipse icon->add this to desktop

Rightclick on the eclipse icon on desktop goto properties->Launcher->command: /home/training/eclipse/./eclipse

**Installing MySql:**

Step1:

$sudo yum install mysql-server

Step2:

$ cd /etc/init.d

$sudo ./mysqld start

Step3:

sudo /usr/bin/mysql\_secure\_installation

**Installing Sqoop:**

Step1:download sqoop from below site

<http://sqoop.apache.org/>

Step2:

$vi .bashrc

export SQOOP\_HOME=/home/training/sqoop-1.4.4.bin\_\_hadoop-1.0.0

export PATH=$PATH:$SQOOP\_HOME

export HADOOP\_COMMON\_HOME=$HADOOP\_INSTALL

export HADOOP\_MAPRED\_HOME=$HADOOP\_INSTALL

Step3: Download respective mysql connector and place it in $SQOOP\_HOME/lib folder

Exercise1:

First connect to MySQL

***$ mysql -u root -p***

***Enter password:***

Create database ‘testDb’ and use ‘testDb’ database as a current database.

***mysql> create database testDb;***

***mysql> use testDb;***

Create table ‘stud1′

***mysql> create table stud1(id integer,name char(20));***

***mysql> exit;***

* **HDFS File ‘student’**

***$hadoop dfs -cat /user/training/student***

***1,Archana***

***2,XYZ***

**Sqoop Export**

***$sqoop export --connect jdbc:mysql://localhost/testDb --table stud1 -m 1 --export-dir /user/hduser/student***

This example takes the files in ***/user/hduser/student*** and injects their contents in to the “stud1” table in the“testDb” database. The target table must already exist in the database.

***Note:***

If you will get this

Error

*com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Access denied for user ''@'localhost' to database 'testDb'*

Solution

Grant all privileges on testDb database to user:

*mysql> grant all privileges on testDb.\* to ''@localhost ;*

**Table Contents in MySQL**

***mysql> use testDb;***

***mysql> select \* from stud1;***

***+------+----------+***

***| id   | name     |***

***+------+----------+***

***| 1    | Archana |***

***| 2    | XYZ      |***

***+------+----------+***

***2 rows in set (0.00 sec)***

**HBASE INSTALLATION:**

Step1: Download Hbase from below link

<https://hbase.apache.org/>

Step2: unzip downloaded file

$tar hbase-0.94.16.tar.gz

Step3:

$vi .bashrc

export HBASE\_HOME=/home/training/hbase-0.94.16

export PATH=$PATH:$HBASE\_HOME/bin

step4:

$cd /home/training/hbase-0.94.16/bin

$./start-hbase.sh

**PIG INSTALLATION:**

Step1). Download PIG from below site

<http://pig.apache.org/releases.html#Download>

Step2).$vi .bashrc

export PIG\_INSTALL=/home/training/pig-0.12.0

export PATH=$PATH:$PIG\_INSTALL/bin

Multinode Cluster Configurations

Upload an “ovf” file into the server

File->Deploy ovf file ->next->Browse the location of ouf file->next(by selecting (“thin” or “thick”)->next->done

After making cluster with as many numbers of nodes and every node must set with either with master daemons or with slave daemons.

**Step1:**

]$sudo vi /etc/sysconfig/network-scripts/ifcfg-eth0

DEVICE=eth0

#BootPROTO=dhcp

IPADDR=192.168.30.171

NETMASK=255.255.255.0

GATEWAY=192.168.30.254

ONBOOT=YES

:wq! (to save and quit the file)

Here ‘dhcp’ allocates one host address to the system by default, due that we are commenting that to provide our own IPaddress.

This is to set the IP address for the vm node in the cluster for a single machine.Here we must comment the BOOTPROTO=dhcp and we must give IPADRR,NETMASIC and GATEWAY.Once we did this.

**Step2:**

]$sudo vi /etc/sysconfig/network

NETWORKING=yes

NETWORKING\_IPV6=no

GATEWAY=192.168.30.254

HOSTNAME=cloudera\_training

:wq!

To get connecting our node in the network we are giving GATEWAY here.

**Step3:**

Set the nameserver

]$sudo vi /etc/resolv.conf

Nameserver 208.67.220.220

Now you are getting an IP address.

**Step4:**

Give dns

]$sudo vi /etc/nsswitch.conf

……………………….

……………………………..

………………………………

hosts: files dns

:wq!

By default ‘dns’ will be there once we set above setting. In case if it is not there then we must give dns to get ping with other nodes.

**Step5:**

Now ping the node

]$ping [www.yahoo/google.com](http://www.yahoo/google.com)

If is is not working , you just restart the network services by using following command.

]$sudo /etc/init.d/network restart

------------------------------ [ok]

----------------------------------- [ok]

------------------------------------ [ok]

Now again ping the node.

]$ping [www.google.com](http://www.google.com)

These are all the steps, will make our cluster network settings.

Now goto next step to install hadoop-0.20 in our nodes in cluster.

**Step6:**

]$sudo yum install hadoop-0.20

Complete!

]$sudo yum install hadoop-0.20-namenode

Complete!

]$sudo yum install hadoop-0.20-secondarynamenode

Complete!

]$sudo yum install hadoop-0.20-jobtracker

Complete!

]$sudo yum install hadoop-0.20-datanode

Complete!

]$sudo yum install hadoop-0.20-tasktracker

Complete!

After installing all packages of hadoop-0.20 we have to set the configurations to the nodes.

**Step7:**

Create a new set of blank configuration files. The ‘conf.empty’ directory contains blank files,so we will copy those to a new directory that is ‘conf.class’.

]$sudo cp –r /etc/hadoop-0.20/conf.empty \

>/etc/hadoop-0.20/conf.class

Here ‘\’ after conf.empty is a symbol that it is saying next line.

**Step8:**

The below command is to tell the alternating system that this is the new directory to use for hadoop’s configuration.

]$sudo /usr/sbin/alternatives—install \

>/etc/hadoop-0.20/conf hadoop-0.20-conf \

>/etc/hadoop-0.20/conf.class 99

Here we are setting the priority that is 99

**Step9:**

Check that the above command In step8 has worked.

]$ /usr/sbin/update-alternatives \

>--display hadoop-0.20-conf

You should see that the current best version is cof.class . This is the version we are going to edit.

**Step10:**

Change directories into the configuration directory.

]$ cd /etc/hadoop-0.20/conf.class

**Setting up the configurations files**

Your group should use one machine to be the master node.The others will be slave ndoes. We will put the same configuration files, we will need to use ‘sudo’.

]$ sudo vi core-site.xml

<configuration>

<property>

<name>fs.default.name</name>

<value>hdfs://node1:8020</value>

</property>

</configuration>

]$ sudo vi hdfs-site.xml

<configuration?

<property>

<name>dfs.name.dir</name>

<value>/home/disk1/dfs/nn, /home/disk2/dfs/nn</value>

</property>

<property>

<name>dfs.data.dir</name>

<value>/home/disk1/dfs/dn, /home/disk2/dfs/dn</value>

</property>

<property>

<name>dfs.http.address</name>

<value>node1:50070</value>

</property>

</configuration>

:wq!

**NOTE:** we have JBOD contains 7 disks and we have one interface to name node, so we need to create two mount points like

/home/disk1/sfs/nn

/home/disk2/dfs/nn

If NFS server is there , we need to give nfs mount point also.

Here the main purpose of two disks is maintaining the redundancy.

]$sudo vi mapred-site.xml

<configuration>

<property>

<name>mapred.local.dir</name>

<value>/home/disk1/mapred/local, /home/disk2/mapred/local</value>

</property>

<property>

<name>mapred.job.tracker</name>

<value>node1:8021</value>

</property>

<property>

<name>mapred.system.dir</name>

<value>/mapred/system</value>

</property>

<property>

<name>mapreduce.jobtracker.stagging.root.dir</name>

<value>/user</value>

</property>

</configuration>

:wq!

Now create the directories which are specified in configuration files.

**Step11:**

Creating directories

]$sudo mkdir –p /home/disk1/dfs/nn

]$sudo mkdir –p /home/disk2/dfs/nn

]$sudo mkdir –p /home/disk1/dfs/nn

]$sudo mkdir –p /home/disk2/dfs/nn

]$sudo mkdir –p /home/disk1/mapred/local

]$sudo mkdir –p /home/disk2/mapred/local

**Step12:**

Change the ownership of the directories.the hdfs user should own the HDFS directories.

]$sudo chown –R hdfs:hadoop /home/disk1/dfs/nn

]$sudo chown –R hdfs:hadoop /home/disk2/dfs/nn

]$sudo chown –R hdfs:hadoop /home/disk1/dfs/nn

]$sudo chown –R hdfs:hadoop /home/disk1/dfs/nn

Mapred user should own the mapreduce directories

]$sudo chown –R mapred:hadoop /home/disk1/dfs/nn

]$sudo chown –R mapred:hadoop /home/disk2/dfs/nn

**Step13:**

Because we are running on a virtual machine with very limited Ram, we will reduce the heap size of hadoop daemons to 200MB.

edit **hadoop-env.sh**

uncomment the line which refers to HADOOP\_HEAPSIZE (remove the # character) and change it so it reads

]$ sudo vi hadoop-env.sh

]#export HADOOP\_HEAPSIZE=2000

The above shown export line with comment (#) will be there by default. So we must change it to uncommented lie as below and change the value as you require according to the RAM

Export HADOOP\_HEAPSIZE = 2000

**Step14:**

When every one has completed these steps , format the namenode.

Note: do this only on the master node.

]$ sudo –u hdfs hadoop namenode –format

If you are asked whether you want to reformat the file system , answer ‘Y’)

**Step15:**

Start the HDFS daemons

**On master node only:**

]$ sudo /etc/init.d/hadoop-0.20-namenode start

]$ sudo /etc/init.d/hadoop-0.20-secondarynamenode start

**On slave nodes only**

]$ sudo /etc/init.d/hadoop-0.20-datanode start

**Step16:**

Create a home directory for the user training. Only one person should do this. Choose one member of the team to enter these commands

]$ sudo –u hdfs hadoop fs –mkdir /user/training

]$ sudo –u hdfs hadoop fs –chown training /user/training

**Step17:**

Create the mapreduce system directory within HDFS. Only one person should do this. Choose one member of the team to enter these commands

]$ sudo –u hdfs hadoop fs –mkdir /mapred/system

]$ sudo –u hdfs hadoop fs –chown mapred:hadoop /mapred/system

**Step18:**

Start the mapreduce daemons

**On master node only:**

]$ sudo /etc/init.d/hadoop-0.20-jobtracker start

**On slave node only:**

**]$ sudo /etc/init.d/hadoop-0.20-tasktracker start**

**Step 19:**

Use the sudo jps command to ensure that the relevant daemons are running successfully on your node. If they are not, find and fix the problems and restart the daemons.

**]$ sudo jps**

**4537 NameNode**

**4674 Jobtracker**

**4232 Tasktracker**

**4675 Datanode**

**4978 SecondaryNameNode**

**Step 20:**

Test the cluster upload some data to the cluster and run a mapreduce job.

**Hadoop Interview Tips:**

**1.What is HDFS?**

A. HDFS, the Hadoop Distributed File System, is a distributed file system designed to hold very large amounts of data (terabytes or even petabytes), and provide high-throughput access to this information. Files are stored in a redundant fashion across multiple machines to ensure their durability to failure and high availability to very parallel applications

**2.What are the Hadoop configuration files?**1. hdfs-site.xml  
2. core-site.xml  
3. mapred-site.xml

**3.How NameNode Handles data node failures?**

NameNode periodically receives a Heartbeat and a Blockreport from each of the DataNode in the cluster.Receipt of a Heartbeat implies that the DataNode is functioning properly.

When NameNode notices that it has not received a heartbeat message from a DataNode after a certain amount of time, the DataNode is identified as dead. Since blocks will be under replicated the system NameNode begins replicating the blocks that were stored on the dead DataNode.

The NameNode takes responsibility of the replication of the data blocks from one DataNode to another.The replication data transfer happens directly between DataNodes and the data never passes through the NameNode.

**4.What is MapReduce in Hadoop?**  
Hadoop MapReduce is a specially designed framework for distributed processing of large data sets on clusters of commodity hardware. The framework itself can take care of scheduling tasks, monitoring them and reassigning of failed tasks.

**5.What is the responsibility of NameNode in HDFS ?**

NameNode is a master daemon for creating metadata for blocks, stored on DataNodes. Every DataNode sends heartbeat and block report to NameNode. If NameNode not receives any heartbeat then it simply identifies that the DataNode is dead. This NameNode is the single Point of failover. If NameNode goes down HDFS cluster is inaccessible.

**6.What it the responsibility of SecondaryNameNode in HDFS?**

SecondaryNameNode is the mater Daemon to create Housekeeping work for NameNode. SecondaryNameNode is not the backup of NameNode but it is the backup for metadata of the NameNode.

**7.What is the DataNode in HDFS?**

DataNode is the slave daemon of NameNode for storing actual data blocks. Each DataNode stores number of 64MB blocks.

**8.What is the JobTracker in HDFS?**

JobTracker is a mater daemon for assigning tasks to TaskTrackers in different DataNodes where it can find data blocks for input file.

**9.How can we list all job running in a cluster?**

]$ hadoop job -list

**10.How can we kill a job?**

]$ hadoop job –kill jobid

**11.Whats the default port that jobtrackers listens to**

[http://localhost:50030](http://localhost:50030/)

**12.Whats the default port where the dfs namenode web ui will listen on**

[http://localhost:50070](http://localhost:50070/)

**13.What is Hadoop Streaming**

A. Streaming is a generic API that allows programs written in virtually any language to be

used as Hadoop Mapper and Reducer implementations

**14.Whats is Distributed Cache in Hadoop**

A. Distributed Cache is a facility provided by the Map/Reduce framework to cache files

(text, archives, jars and so on) needed by applications during execution of the job. The

framework will copy the necessary files to the slave node before any tasks for the job are

executed on that node.

**15.What is the benifit of Distributed cache, why can we just have the file in HDFS and have the application read it**

A. This is because distributed cache is much faster. It copies the file to all trackers at the

start of the job. Now if the task tracker runs 10 or 100 mappers or reducer, it will use the

same copy of distributed cache. On the other hand, if you put code in file to read it from

HDFS in the MR job then every mapper will try to access it from HDFS hence if a task

tracker run 100 map jobs then it will try to read this file 100 times from HDFS. Also

HDFS is not very efficient when used like this.

**16.Is it possible to provide multiple input to Hadoop? If yes then how can you give multiple directories as input to the Hadoop job**

A. Yes, The input format class provides methods to add multiple directories as input to a

Hadoop job

**17.What will a hadoop job do if you try to run it with an output directory that is already present? Will it** **overwrite it - warn you and continue - throw an exception and exit**

A. The hadoop job will throw an exception and exit.

**18.How can you set an arbitary number of mappers to be created for a job in Hadoop**

A. This is a trick question. You cannot set it

**19.How can you set an arbitary number of reducers to be created for a job in Hadoop**

A. You can either do it progamatically by using method setNumReduceTasksin the

JobConfclass or set it up as a configuration setting

**20.How will you write a custom partitioner for a Hadoop job**

A. To have hadoop use a custom partitioner you will have to do minimum the following

three

- Create a new class that extends Partitioner class

- Override method getPartition

- In the wrapper that runs the Map Reducer, either

- add the custom partitioner to the job programtically using method setPartitionerClass

or

- add the custom partitioner to the job as a config file (if your wrapper reads from config

file or oozie)

**21.How did you debug your Hadoop code?**

A. There can be several ways of doing this but most common ways are

- By using counters

- The web interface provided by Hadoop framework

**22.What does the term "Replication factor" mean**

A. Replication factor is the number of times a file needs to be replicated in HDFS

**23.What is the default replication factor in HDFS**

A. The default replication factor is 3

**24. What is the typical block size of an HDFS block**

A. The default HDFS block size is 64Mb or 128Mb

**25.What is the benefit of having such big block size (when compared to block size of linux file system like ext)**

A. It allows HDFS to decrease the amount of metadata storage required per file (the list of

blocks per file will be smaller as the size of individual blocks increases). Furthermore, it

allows for fast streaming reads of data, by keeping large amounts of data sequentially laidout on the disk

**26.Why is it recommended to have few very large files instead of a lot of small files in HDFS**

A. This is because the Name node contains the meta data of each and every file in HDFS

and more files means more metadata and since namenode loads all the metadata in

memory for speed hence having a lot of files may make the metadata information big

enough to exceed the size of the memory on the Name node

**27.What alternate way does HDFS provides to recover data in case a Namenode, without backup, fails and cannot be recovered**

A. There is no way. If Namenode dies and there is no backup then there is no way to recover data

**28.Describe how a HDFS client will read a file in HDFS, like will it talk to data node or namenode ... how will data flow etc**

A. To open a file, a client contacts the Name Node and retrieves a list of locations for the

blocks that comprise the file. These locations identify the Data Nodes which hold each

block. Clients then read file data directly from the Data Node servers, possibly in parallel.

The Name Node is not directly involved in this bulk data transfer, keeping its overhead to

a minimum.

**29.Using linux command line. how will you List the the number of files in a HDFS directory**

* hadoop fs -ls

**30.Using linux command line. how will Create a directory in HDFS**

* hadoop fs -mkdir

**31.Using linux command line. how will you Copy file from your local directory to HDSF**

* hadoop fs -put localfile hdfsfile

**32.What platforms and Java versions does Hadoop run on?**

A. Java 1.6.x or higher, preferably from Sun. Linux and Windows are the supported

operating systems, but BSD, Mac OS/X, and OpenSolaris are known to work. (Windows

requires the installation of Cygwin).

**33.Is there an easy way to see the status and health of a cluster?**

A. There are web-based interfaces to both the JobTracker (MapReduce master) and

NameNode (HDFS master) which display status pages about the state of the entire

system.

By default, these are located at http://job.tracker.addr:50030/ and

<http://name.node.addr:50070/>.

The JobTracker status page will display the state of all nodes, as well as the job queue

and status about all currently running jobs and tasks. The NameNode status page will

display the state of all nodes and the amount of free space, and provides the ability to

browse the DFS via the web.

You can also see some basic HDFS cluster health data by running:

$ bin/hadoop dfsadmin –report

**34.Do I have to write my job in Java?**

A. No. There are several ways to incorporate non-Java code.

**35.How do I submit extra content (jars, static files, etc) for my job to use during runtime?**

A. The distributed cache feature is used to distribute large read-only files that are needed by

map/reduce jobs to the cluster. The framework will copy the necessary files from a URL

(either hdfs: or http:) on to the slave node before any tasks for the job are executed on

that node. The files are only copied once per job and so should not be modified by the

application.

Copying content into lib is not recommended and highly discouraged. Changes in that

directory will require Hadoop services to be restarted.

**36.How do I change final output file name with the desired name rather than in partitions like part-00000, part-00001?**

A. You can subclass the OutputFormat.java class and write your own. You can look at the

code of TextOutputFormat MultipleOutputFormat.java etc. for reference. It might be the

case that you only need to do minor changes to any of the existing Output Format classes.

To do that you can just subclass that class and override the methods you need to change.

**37.How do you gracefully stop a running job?**

A. hadoop job -kill <JOBID>

**38.How the HDFS Blocks are replicated?**

A. HDFS is designed to reliably store very large files across machines in a large cluster. It

stores each file as a sequence of blocks; all blocks in a file except the last block are the

same size. The blocks of a file are replicated for fault tolerance. The block size and

replication factor are configurable per file. An application can specify the number of

replicas of a file. The replication factor can be specified at file creation time and can be

changed later. Files in HDFS are write-once and have strictly one writer at any time. The

NameNode makes all decisions regarding replication of blocks. HDFS uses rack-aware

replica placement policy. In default configuration there are total 3 copies of a datablock

on HDFS, 2 copies are stored on datanodes on same rack and 3rd copy on a different

rack.

**39.How the Client communicates with HDFS?**

A. The Client communication to HDFS happens using Hadoop HDFS API. Client

applications talk to the NameNode whenever they wish to locate a file, or when they want

to add/copy/move/delete a file on HDFS. The NameNode responds the successful

requests by returning a list of relevant DataNode servers where the data lives. Client

applications can talk directly to a DataNode, once the NameNode has provided the

location of the data.

**40.What is HDFS Block size? How is it different from traditional file system block size?**

**A.** In HDFS data is split into blocks and distributed across multiple nodes in the cluster.

Each block is typically 64Mb or 128Mb in size. Each block is replicated multiple times.

Default is to replicate each block three times. Replicas are stored on different nodes.

HDFS utilizes the local file system to store each HDFS block as a separate file. HDFS

Block size can not be compared with the traditional file system block size.

**41.When is the reducers are started in a MapReduce job?**

A. In a MapReduce job reducers do not start executing the reduce method until the all Map

jobs have completed. Reducers start copying intermediate key-value pairs from the

mappers as soon as they are available. The programmer defined reduce method is called

only after all the mappers have finished.

**42.If reducers do not start before all mappers finish then why does the progress on MapReduce job shows something like Map(60%) Reduce(15%)? Why reducers progress percentage is displayed when mapper is not finished yet?**

A. Reducers start copying intermediate key-value pairs from the mappers as soon as they are available. The progress calculation also takes in account the processing of data transfer

which is done by reduce process, therefore the reduce progress starts showing up as soon as any intermediate key-value pair for a mapper is available to be transferred to reducer.

Though the reducer progress is updated still the programmer defined reduce method is

called only after all the mappers have finished.

**43.What is the Hadoop MapReduce API contract for a key and value Class?**

A. The Key must implement the org.apache.hadoop.io.WritableComparable interface.

The value must implement the org.apache.hadoop.io.Writable interface.

**44.What are combiners? When should I use a combiner in my MapReduce Job?**

A. Combiners are used to increase the efficiency of a MapReduce program. They are used to aggregate intermediate map output locally on individual mapper outputs. Combiners can

help you reduce the amount of data that needs to be transferred across to the reducers.

You can use your reducer code as a combiner if the operation performed is commutative

and associative. The execution of combiner is not guaranteed, Hadoop may or may not

execute a combiner. Also, if required it may execute it more then 1 times. Therefore your

MapReduce jobs should not depend on the combiners execution.

**45.Where is the Mapper Output (intermediate kay-value data) stored ?**

A. The mapper output (intermediate data) is stored on the Local file system (NOT HDFS) of each individual mapper nodes. This is typically a temporary directory location which can

be setup in config by the hadoop administrator. The intermediate data is cleaned up after the Hadoop Job completes.

**46.Name the most common InputFormats defined in Hadoop? Which**

**one is default ?**

A. Following 2 are most common InputFormats defined in Hadoop

- TextInputFormat

- KeyValueInputFormat

- SequenceFileInputFormat

TextInputFormatis the hadoop default

**47. What is the difference between TextInputFormat and KeyValueInputFormat class**

A. TextInputFormat: It reads lines of text files and provides the offset of the line as key to the Mapper and actual line as Value to the mapper KeyValueInputFormat: Reads text file and parses lines into key, val pairs.

Everything up to the first tab character is sent as key to the Mapper and the remainder of the line is sent as value to the mapper.

**48. What is InputSplit in HadoopA.**

A. When a hadoop job is run, it splits input files into chunks and assign each split to a mapper to process. This is called Input Split

**49. How is the splitting of file invoked in Hadoop Framework**

A. It is invoked by the Hadoop framework by running getInputSplit()method of the Input format class (like FileInputFormat) defined by the user

**50. Consider case scenario: In M/R system,**

**- HDFS block size is 64 MB**

**- Input format is FileInputFormat**

**- We have 3 files of size 64K, 65Mb and 127Mb**

**then how many input splits will be made by Hadoop framework?**

A. Hadoop will make 5 splits as follows

- 1 split for 64K files

- 2 splits for 65Mb files

- 2 splits for 127Mb file

**51. What is the purpose of RecordReader in Hadoop**

The InputSplithas defined a slice of work, but does not describe how to access it. The RecordReaderclass actually loads the data from its source and converts it into (key, value) pairs suitable for reading by the Mapper. The RecordReader instance is defined by the InputFormat

**52. After the Map phase finishes, the hadoop framework does**

**"Partitioning, Shuffle and sort". Explain what happens in this phase?**

A. Partitioning is the process of determining which reducer instance will receive which intermediate keys and values. Each mapper must determine for all of its output (key, value) pairs which reducer will receive them. It is necessary that for any key, regardless of which mapper instance generated it, the destination partition is the same

- Shuffle

After the first map tasks have completed, the nodes may still be performing several more map tasks each. But they also begin exchanging the intermediate outputs from the map tasks to where they are required by the reducers. This process of moving map outputs to the reducers is known as shuffling.

- Sort

Each reduce task is responsible for reducing the values associated with several intermediate keys. The set of intermediate keys on a single node is automatically sorted by Hadoop before they are presented to the Reducer

**53. If no custom partitioner is defined in the hadoop then how is data partitioned before its sent to the reducer?**

A.The default partitioner computes a hash value for the key and assigns the partition based on this result.

**54. What is a Combiner**

A. The Combiner is a "mini-reduce" process which operates only on data generated by a mapper. The Combiner will receive as input all data emitted by the Mapper instances on a given node. The output from the Combiner is then sent to the Reducers, instead of the output from the Mappers.

**55. Give an example scenario where a cobiner can be used and where it cannot be used**

A. There can be several examples following are the most common ones

- Scenario where you can use combiner

Getting list of distinct words in a file

- Scenario where you cannot use a combiner

Calculating mean of a list of numbers

**56.What is job tracker**

A. Job Tracker is the service within Hadoop that runs Map Reduce jobs on the

cluster

**57. What are some typical functions of Job Tracker**

The following are some typical tasks of Job Tracker

- Accepts jobs from clients

-It talks to the NameNode to determine the location of the data

-It locates TaskTracker nodes with available slots at or near the data

-It submits the work to the chosen Task Tracker nodes and monitors progress of each task by receiving heartbeat signals from Task tracker

**58.What is task tracker**

A. Task Tracker is a node in the cluster that accepts tasks like Map, Reduce and Shuffle operations - from a JobTracker

**59. Whats the relationship between Jobs and Tasks in Hadoop**

One job is broken down into one or many tasks in Hadoop.

**60. Suppose Hadoop spawned 100 tasks for a job and one of the task**

**failed. What will hadoop do ?**

It will restart the task again on some other task tracker and only if the task fails more than 4 (default setting and can be changed) times will it kill the job

**61.Hadoop achieves parallelism by dividing the tasks across many nodes, it is possible for a few slow nodes to rate-limit the rest of the program and slow down the program. What mechanism Hadoop provides to combat this**

A. Speculative Execution

**62. How does speculative execution works in Hadoop**

A. Job tracker makes different task trackers process same input. When tasks

complete, they announce this fact to the Job Tracker. Whichever copy of a

task finishes first becomes the definitive copy. If other copies were

executing speculatively, Hadoop tells the Task Trackers to abandon the

tasks and discard their outputs. The Reducers then receive their inputs

from whichever Mapper completed successfully, first.

**Sqoop interview tips:**

**1.How to Import data from RDBMS to HDFS?**

]$ sqoop import \

>--connect jdbc:mysql://<hostname>/<Database name> \

>--username <DB username> --password <password> \

>--table <table name> --fields-terminated-by ‘\t’

**2. How to export data from HDFS to RDBMS ?**

]$ sqoop export \

> --connect jdbc:mysql://<hostname>/<Database name> \

> --username <DB username> --password <DB password> \

> --table <table name> \

> --input-fields-terminated-by ‘\t’ \

> --export-dir <HDFS directory name>

**3. By default how many number of mappers sqoop is working with?**

A. By default sqoop is working with 4 mappers.

**4. Can’t we work with possible number of mappers?**

A. By default , sqoop is running with 4 mappers. Sometimes even our file is <64MB also , still this sqoop is working with 4 mappers. So if we want , we can work with variable number of mappers

By using “ -m <no.of mappers> ” .

**5. How many number of reducers sqoop is working with ?**

A. sqoop will never work with any more reducers. So Mappers only will give output files directly.

**6. How to list number of DataBases from RDBMS with sqoop?**

]$ sqoop list-databases \

>--connect jdbc:mysql://<hostname> \

>--username <DB username> --password <password>

**7. How to list number of tables from a specific Database in RDBMS with sqoop?**

]$ sqoop list-tables \

>--connect jdbc:mysql://<hostname>/<databasename> \

>--username <DB username> --password <password>

**8. How to import data from RDBMS to HBASE with sqoop ?**

]$ sqoop import \

>--connect jdbc:mysql://<hostname>/<databasename> \

>--username <DB username> --password <DB password> \

>--table <table name> \

>--hbase-table <hbase table name> \

>--column-family <column family name> \

>--hbase-create-table

**9. How import all tables at a time from a specific database ?**

]$ sqoop import-all-tables \

>--connect jdbc:mysql://<hostname>/<databasename> \

>--username <DB username> --password <password>

**10. How to work with number of mappers?**

]$ sqoop import \

>--connect jdbc:mysql://<hostname>/<databasename> \

>--username <DB username> --password <password> -m <no.of mappers>

**11. How to import data from RDBMS to Hive?**

]$ sqoop import \

>--connect jdbc:mysql://<hostname>/<database name> \

>--username <DB username> --password <DB password> \

>--table <tablename> \

>--hive-table <tablename> \

>--create-hive-table --hive-import --hive-home <path to hive home>