2018-19 Onwards (MR-18)	MALLA REDDY ENGINEERING COLLEGE (Autonomous)	B.Tech. VII Semester		
Code: 80521	**BIG DATA ANALYTICS** [Professional Elective - IV]	**L**	**T**	**P**
Credits: 3		3	-	-

Prerequisites: Databases, programming fundamentals.

Course Objectives:
This course enables the students to learn and understand Big data , data analytics, R language, developing map reduce programs, discuss about concepts of big data, make use of Hadoop concepts for designing applications, develop applications using Hadoop I/O and analyze big data using programming tools such as Pig and Hive.

MODULE I: Big data overview, data analytics, and R Language   [09 Periods]
 Big Data Overview : Data Structures, Analyst Perspective on Data Repositories , State of the Practice in Analytics, Bl Versus Data Science, Current Analytical Architecture , Drivers of Big Data ,   Emerging Big Data Ecosystem and a New Approach to Analytics, Key Roles for the New Big Data Ecosystem, Examples of Big Data Analytics. Data Analytics Lifecycle , Model Building and Basic Data Analytic Methods Using R Data Analytics Lifecycle Overview,   Key Roles for a Successful Analytics Project, Background and Overview of Data Analytics Lifecycle - Discovery , Data Preparation, Learning the Business Domain , Model Planning , Model building, Communicate Results, Operationalize and case study example Global Innovation Network and Analysis (GINA)
R Introduction: Introduction to R, Exploratory Data Analysis, Statistical Methods for Evaluation, Hypothesis Testing, Difference of Means, Rank-Sum Test, Errors, Sample Size data

MODULE II: Working with Big Data   [09 Periods]
Hadoop - Google File System, Hadoop Distributed File System (HDFS)– Building blocks of Hadoop (Namenode, Datanode, Secondary Namenode, JobTracker, TaskTracker).
Configuring of Hadoop Cluster -   Introducing and Configuring Hadoop cluster (Local, Pseudo-distributed mode, Fully Distributed mode), Configuring XML files.

MODULE III: Hadoop API and Map Reduce Programs   [09 Periods]
A: Hadoop API - Writing MapReduce Programs: A Weather Dataset, Understanding Hadoop API for MapReduce Framework (Old and New)
B: MapReduce Programs with classes -   Basic programs of Hadoop MapReduce: Driver code, Mapper code, Reducer code, RecordReader, Combiner, Partitioner.

MODULE IV: Hadoop I/O and Implementation   [09 Periods]

Hadoop I/O - The Writable Interface, Writable Comparable and comparators, Writable Classes: Writable wrappers for Java primitives, Text, BytesWritable, NullWritable, ObjectWritable and GenericWritable, Writable collections.
Implementation - Implementing a Custom Writable: Implementing a RawComparator for speed, Custom comparators.

MODULE V: PIG and HIVE HADOOP TOOL   **[12 Periods]**
PIG - HADOOP TOOL -   Hadoop Programming Made Easier - Admiring the Pig Architecture, Going with the Pig Latin Application Flow, Working through the ABCs of Pig Latin, Evaluating Local and Distributed Modes of Running Pig Scripts, Checking out the Pig Script Interfaces, Scripting with Pig Latin.
HIVE – HADOOP TOOL - Saying Hello to Hive, Seeing How the Hive is Put Together, Getting Started with Apache Hive, Examining the Hive Clients, Working with Hive Data Types, Creating and Managing Databases and Tables, Seeing How the Hive Data Manipulation Language Works, Querying and Analyzing Data.

TEXT BOOKS
1. Data Science & Big Data Analytics Discovering, Analyzing, Visualizing and Presenting Data EMC Education Services, Wiley Publishers, 2015.
2. Cay Horstmann, Wiley John Wiley & Sons, "**Big Java**", 4th Edition, INC
3. Tom White, "**Hadoop: The Definitive Guide**" 3rd Edition, O'reilly

REFERENCES
1. Alex Holmes, "**Hadoop in Practice**", MANNING Publ.
2. Srinath Perera, Thilina Gunarathne, "**Hadoop MapReduce**" Cookbook.

E-RESOURCES
1. http://newton.uam.mx/xgeorge/uea/Lab_Prog_O_O/materiales_auxiliares/Big_Java_4th_Ed.pdf
2. http://www.isical.ac.in/~acmsc/WBDA2015/slides/hg/Oreilly.Hadoop.The.Definitive.Guide.3rd.Edition.Jan.2012.pdf-
3. https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf
4. http://www.comp.nus.edu.sg/~ooibc/mapreduce-survey.pdf
5. http://freevideolectures.com/Course/3613/Big-Data-and-Hadoop/18
6. http://freevideolectures.com/Course/3613/Big-Data-and-Hadoop/40

Course Outcomes:
At the end of the course, students will be able to
1. **Develop** simple applications using R language
2. **Analyze** file systems such as GFS and HDFS.
3. **Design** applications by applying Map reduce concepts.
4. **Build** up programs by making use of I/O.
5. **Explore** and inspect the big data using programming tools like Pig and Hive.

CO- PO Mapping
(3/2/1 indicates strength of correlation) 3-Strong, 2-Medium, 1-Weak

COs	Programme Outcomes(POs)												PSOS		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	3	3	3	3	1		1	3	2	1	3	3	2	3
CO2	3	2	3	3	3						2	1	3	2	2
CO3	3	3	3	3	3							3	3	2	2
CO4	3	3	3	3	3						1	3	3	2	2
CO5	2	3	3	3	3						1	3	3	2	2

2018-19 Onwards (MR-18)	MALLA REDDY ENGINEERING COLLEGE (Autonomous)	B.Tech. VI Semester		
Code: 80523	**OBJECT ORIENTED ANALYSIS AND DESIGN**	**L**	**T**	**P**
Credits: 3	[Professional Elective - IV]	**3**	**-**	**-**

Prerequisite: NIL
Course Objectives:
The student will be able to understand the Unified Modeling Language Principles and learns fundamental process pattern for object-oriented analysis and design.

Module I: UML
Introduction to UML [09 Periods]
Importance of modeling, principles of modeling, object oriented modeling, conceptual model of the UML, Architecture, and Software Development Life Cycle.

Module II: Behavioral and structural Modeling [09 Periods]
Basic Behavioral Modeling-I: Use cases, Use case Diagrams, Activity Diagrams.
Basic Structural Modeling: Classes, Relationships, common Mechanisms, and diagrams.

Module III: Behavioral Model II
A: Advanced Structural Modeling [12 Periods]
Advanced classes, advanced relationships, Interfaces, Types and Roles, Packages.
B: Class & Object Diagrams: Terms, concepts, modeling techniques for Class & Object Diagrams.
Basic Behavioral Modeling-II : Interactions, Interaction diagrams

Module IV: Advanced Behavioral Modeling
Advanced Behavioral Modeling [09 Periods]
Events and signals, state machines, processes and Threads, time and space, state chart diagrams.

Module V: Architecture Modeling
Architectural Modeling [09 Periods]
Component, Deployment, Component diagrams and Deploymentdiagrams.
Case Study: The Unified Library application.

UNIT-I
Data structures in Java

Collection framework provides interfaces and class implementations that enable data handling easy and meaningful. Using collections one can store, retrieve and manipulate the data very effectively and easily.

Before we start lets first discuss the Architecture of Collection framework and hierarchy of Classes and Interfaces.
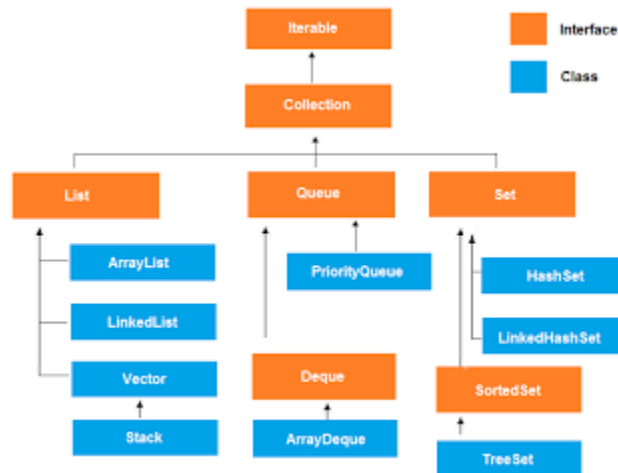
Collection Interface

The Collection Interface resides at the top of the Hierarchy, although Java does not provides a direct implementation of Collection framework but Collection Interface is being implemented by List and Set Classes.

Data Structure:

A **data structure** is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

Linked List:

A **linked list** is a data structure used for collecting a sequence of objects that allows efficient addition and removal of elements in the middle of the sequence.

The Structure of Linked Lists:

To understand the inefficiency of arrays and the need for a more efficient data structure, imagine a program that maintains a sequence of employee names. If an employee leaves the company, the name must be removed. In an array, the hole in the sequence needs to be closed up by moving all objects that come after it.

Each node in a linked list is connected to the neighboring nodes.

IT Dept   Page 1

Conversely, suppose an employee is added in the middle of the sequence. Then all names following the new hire must be moved toward the end. Moving a large number of elements can involve a substantial amount of processing time.

A linked list structure avoids this movement. A linked list uses a sequence of *nodes*. A node is an object that stores an element and references to the neighboring nodes in the sequence.
A linked list consists of a number of nodes, each of which has a reference to the next node.

Figure: A Linked List Tom Diana Harry

When you insert a new node into a linked list, only the neighboring node references need to be updated.
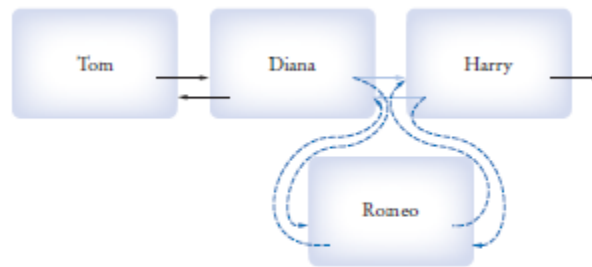

Figure: Inserting a Node into a Linked List

The same is true when you remove a node. What's the catch? Linked lists allow efficient insertion and removal, but element access can be inefficient.

Figure: Removing a Node from a Linked List

For example, suppose you want to locate the fifth element. You must first traverse the first four. This is a problem if you need to access the elements in arbitrary order. The term "random access" is used in computer science to describe an access pattern in which elements are accessed in arbitrary (not necessarily random) order.

In contrast, sequential access visits the elements in sequence. Of course, if you mostly visit all elements in sequence (for example, to display or print the elements), the inefficiency of random access is not a problem. You use linked lists when you are concerned about the efficiency of inserting or removing elements and you rarely need element access in random order.

➢ The Java library provides a LinkedList class in the java.util package. It is a **generic class**, just like the ArrayList class. That is, you specify the type of the list elements in angle brackets, such as **LinkedList<String>** or **LinkedList<Employee>**.

➢ The below Table shows important methods of the LinkedList class. As you can see from Table, there are methods for accessing the beginning and the end of the list directly. However, to visit the other elements, you need a list **iterator**.

Working with Linked Lists	
LinkedList<String> list = new LinkedList<String>();	An empty list.
list.addLast("Harry");	Adds an element to the end of the list. Same as add.
list.addFirst("Sally");	Adds an element to the beginning of the list. list is now [Sally, Harry].
list.getFirst();	Gets the element stored at the beginning of the list; here "Sally".
list.getLast();	Gets the element stored at the end of the list; here "Harry".
String removed = list.removeFirst();	Removes the first element of the list and returns it. removed is "Sally" and list is [Harry]. Use removeLast to remove the last element.

List Iterator:

An iterator encapsulates a position anywhere inside the linked list. Conceptually, you should think of the iterator as pointing between two elements, just as the cursor in a word processor points between two characters. In the conceptual view, think of each element as being like a letter in a word processor, and think of the iterator as being like the blinking cursor between letters.

You obtain a list iterator with the listIterator method of the LinkedList class:

LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator();

Note that the iterator class is also a generic type. A ListIterator<String> iterates through a list of strings; a ListIterator<Book> visits the elements in a LinkedList<Book>. Initially, the iterator points before the first element.

You can move the iterator position with the next method: **iterator.next();**

The next method throws a **NoSuchElementException** if you are already past the end of the list. You should always call the iterator's **hasNext** method before calling next—it returns true if there is a next element.

```
if (iterator.hasNext())
{
  iterator.next();
}
```

The next method returns the element that the iterator is passing. When you use a **ListIterator<String>**, the return type of the next method is String. In general, the return type of the next method matches the list iterator's type parameter.

You traverse all elements in a linked list of strings with the following loop:

```
while (iterator.hasNext())
{
  String name = iterator.next();
}
```

| Methods of the Iterator and ListIterator Interfaces	
ListIterator<String> iter = list.listIterator()	Provides an iterator for visiting all list elements.
String s = iter.next();	Assume that iter points to the beginning of the list [Sally] before calling next. After the call, s is "Sally" and the iterator points to the end.
iter.previous(); iter.set("Juliet");	The set method updates the last element returned by next or previous. The list is now [Juliet].
iter.hasNext()	Returns false because the iterator is at the end of the collection.

if (iter.hasPrevious())
{
s = iter.previous();
}
	hasPrevious returns true because the iterator is not at the beginning of the list. previous and hasPrevious are ListIterator methods.

iter.add("Diana");	Adds an element before the iterator position (ListIterator only). The list is now [Diana, Juliet].
iter.next(); iter.remove();	remove removes the last element returned by next or previous. The list is now [Diana].
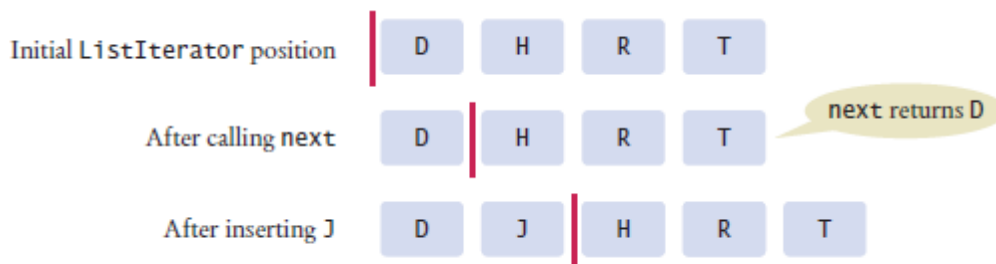
Figure: A Conceptual View of the List Iterator

Example:

```java
import java.util.LinkedList;
import java.util.ListIterator;
 /** This program demonstrates the LinkedList class.  */
public class ListDemo
{
   public static void main(String[] args)
   {
   LinkedList<String> staff = new LinkedList<String>();
   staff.add("Diana");
   staff.add("Harry");
   staff.add("Romeo");
   staff.add("Tom");
   System.out.println("Elements are: "+staff);

   // | in the comments indicates the iterator position
   ListIterator<String> iterator = staff.listIterator(); // |DHRT
   iterator.next(); // D|HRT
   iterator.next(); // DH|RT

   // Add more elements after second element
   iterator.add("Juliet"); // DHJ|RT
   iterator.add("Nina"); // DHJN|RT
   iterator.next(); // DHJNR|T

   // Remove last traversed element
   iterator.remove(); // DHJN|T

   // Print all elements
   System.out.println("After Modifications Elements are: "+staff);
   }
}
```

Output:

Elements are: [Diana, Harry, Romeo, Tom]
After Modifications Elements are: [Diana, Harry, Juliet, Nina, Tom]

Stack:
- A **stack** lets you insert and remove elements only at one end, traditionally called the *top* of the stack.
- New items can be added to the top of the stack. Items are removed from the top of the stack as well.
- Therefore, they are removed in the order that is opposite from the order in which they have been added, called *last-in, first-out* or *LIFO* order.

For example, if you add items A, B, and C and then remove them, you obtain C, B, and A. With stacks, the addition and removal operations are called push and pop.

```
Stack<String> s = new Stack<String>();
s.push("A"); s.push("B"); s.push("C");
while (s.size() > 0)
{
System.out.print(s.pop() + " "); // Prints C B A
}
```

- ➢ There are many applications for stacks in computer science. Consider the undo feature of a word processor. When you select "Undo", the *last* command is undone, then the next-to-last, and so on.
- ➢ Another important example is the **run-time stack** that a processor or virtual machine keeps to store the values of variables in nested methods. Whenever a new method is called, its parameter variables and local variables are pushed onto a stack. When the method exits, they are popped off again.
- ➢ The Java library provides a simple Stack class with methods **push, pop, and peek,** the latter gets the top element of the stack but does not remove it.

Working with Stacks	
Stack<Integer> s = new Stack<Integer>();	Constructs an empty stack.

s.push(1);
s.push(2);
s.push(3);
	Adds to the top of the stack; s is now [1, 2, 3]. (Following the toString method of the Stack class, we show the top of the stack at the end.)

int top = s.pop();	Removes the top of the stack; top is set to 3 and s is now [1, 2].
head = s.peek();	Gets the top of the stack without removing it; Head is set to 2;

Example:
```
import java.util.Stack;
public class StackDemo
{
    public static void main(String[] args)
  {
    Stack<String> s = new Stack<String>();
    s.push("Phani"); s.push("Srikanth"); s.push("Srinu");
    System.out.println("Elements are: "+s);
    s.pop();
    System.out.println("After pop Elements are: ");
    while (s.size() > 0)
    {
        System.out.print(s.pop() + " ");
     }
   }
}
```

Output:

Elements are: [Phani, Srikanth, Srinu]
After pop Elements are: Srikanth Phani

Queue:

A **queue** lets you add items to one end of the queue (the *tail*) and remove them from the other end of the queue (the *head*).

➢ Queues yield items in a *first-in, first-out* or *FIFO* fashion.
➢ Items are removed in the same order in which they were added.

A typical application is a print queue. A printer may be accessed by several applications, perhaps running on different computers. If each of the applications tried to access the printer at the same time, the printout would be garbled. Instead, each application places its print data into a file and adds that file to the print queue.

When the printer is done printing one file, it retrieves the next one from the queue. Therefore, print jobs are printed using the "first-in, first-out" rule, which is a fair arrangement for users of the shared printer.

➢ The Queue interface in the standard Java library has methods add to add an element to the tail of the queue, remove to remove the head of the queue, and peek to get the head element of the queue without removing it.

Working with Queues	
Queue<Integer> q = new LinkedList<Integer>();	The LinkedList class implements the Queue interface.

q.add(1);
q.add(2);
q.add(3);
	Adds to the tail of the queue; q is now [1, 2, 3].

int head = q.remove();	Removes the head of the queue; head is set to 1 and q is [2, 3].
head = q.peek();	Gets the head of the queue without removing it; head is set to 2.

➢ The LinkedList class implements the Queue interface. Whenever you need a queue, simply initialize a Queue variable with a LinkedList object:

```
Queue<String> q = new LinkedList<String>();
q.add("A"); q.add("B"); q.add("C");
while (q.size() > 0)
{
    System.out.print(q.remove() + " ");  // Prints A B C
}
```

Set:

➢ The Set interface in the standard Java library has the same methods as the Collection interface.
➢ There is an essential difference between arbitrary collections and sets.

IT Dept

➤ A set does not admit duplicates. If you add an element to a set that is already present, the insertion is ignored.

➤ The Java platform contains three general-purpose Set implementations: **HashSet, TreeSet, and LinkedHashSet.**

➤ **HashSet:** which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration.

➤ **TreeSet:** which stores its elements in a red-black tree, orders its elements based on ascending order, it is substantially slower than HashSet.

➤ **LinkedHashSet:** which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).

The methods declared by Set are summarized in the following table:

SN	Methods with Description
1	**add( )** Adds an object to the collection
2	**clear( )** Removes all objects from the collection
3	**contains( )** Returns true if a specified object is an element within the collection
4	**isEmpty( )** Returns true if the collection has no elements
5	**iterator( )** Returns an Iterator object for the collection which may be used to retrieve an object
6	**remove( )** Removes a specified object from the collection
7	**size( )** Returns the number of elements in the collection

Example:

Set have its implementation in various classes like HashSet, TreeSet, LinkedHashSet. Following is the example to explain Set functionality:

```
import java.util.*;

public class SetDemo {

 public static void main(String args[]) {
    int count[] = {34, 22,10,60,30,22};
    Set<Integer> set = new HashSet<Integer>();
    try{
      for(int i = 0; i<5; i++){
        set.add(count[i]);
      }
      System.out.println(set);

      TreeSet sortedSet = new TreeSet<Integer>(set);
      System.out.println("The sorted list is:");
      System.out.println(sortedSet);
      System.out.println("The First element of the set is: "+ (Integer)sortedSet.first());
      System.out.println("The last element of the set is: "+ (Integer)sortedSet.last());
    }
    catch(Exception e){}
 }
}
```

This would produce the following result:

```
[amrood]$ java SetDemo
[34, 30, 60, 10, 22]
The sorted list is:
[10, 22, 30, 34, 60]
The First element of the set is: 10
The last element of the set is: 60
```

Map Interface:

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

- Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
- Several methods throw a **NoSuchElementException** when no items exist in the invoking map.
- A **ClassCastException** is thrown when an object is incompatible with the elements in a map.
- A **NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the map.

- An **UnsupportedOperationException** is thrown when an attempt is made to change an unmodifiable map.

SN	Methods with Description
1	**void clear( )** Removes all key/value pairs from the invoking map.
2	**boolean containsKey(Object k)** Returns true if the invoking map contains k as a key. Otherwise, returns false.
3	**boolean containsValue(Object v)** Returns true if the map contains v as a value. Otherwise, returns false
4	**Set entrySet( )** Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map.
5	**boolean equals(Object obj)** Returns true if obj is a Map and contains the same entries. Otherwise, returns false.
6	**Object get(Object k)** Returns the value associated with the key k.
7	**int hashCode( )** Returns the hash code for the invoking map.
8	**boolean isEmpty( )** Returns true if the invoking map is empty. Otherwise, returns false.
9	**Set keySet( )** Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
10	**Object put(Object k, Object v)** Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
11	**void putAll(Map m)** Puts all the entries from m into this map.

12	**Object remove(Object k)**
	Removes the entry whose key equals k.
13	**int size( )**
	Returns the number of key/value pairs in the map.
14	**Collection values( )**
	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Example:

Map has its implementation in various classes like HashMap. Following is the example to explain map functionality:

```java
import java.util.*;

public class CollectionsDemo {

  public static void main(String[] args) {
    Map<String,Integer> m1 = new HashMap<String,Integer > ();
    m1.put("Zara", "8");
    m1.put("Mahnaz", "31");
    m1.put("Ayan", "12");
    m1.put("Daisy", "14");
    System.out.println();
    System.out.println(" Map Elements");
    System.out.print("\t" + m1);
  }
}
```

This would produce the following result:

```
Map Elements
     {Mahnaz=31, Ayan=12, Daisy=14, Zara=8}
```

Generics:

Generics was added in Java 5 to provide **compile-time type checking** and removing risk of **ClassCastException** that was common while working with collection classes. The whole collection framework was re-written to use generics for type-safety. Let's see how generics help us using collection classes safely.

```java
List list = new ArrayList();
list.add("abc");
list.add(new Integer(5)); //OK
String str=(String) list.get(0);   //type casting leading to ClassCastException at runtime
String str=(String) list.get(1);
```

IT Dept   Page 11

Above code compiles fine but throws **ClassCastException** at runtime because we are trying to cast Object in the list to String whereas one of the element is of type Integer. After Java 5, we use collection classes like below.

```
List<String> list1 = new ArrayList<String>(); // java 7 ? List<String> list1 = new ArrayList<>();
list1.add("abc");
//list1.add(new Integer(5)); //compiler error

for(String str : list1){
    //no type casting needed, avoids ClassCastException
}
```

Notice that at the time of list creation, we have specified that the type of elements in the list will be **String**. So if we try to add any other type of object in the list, the program will throw **compile time error**. Also notice that in for loop, we **don't need type casting** of the element in the list, hence removing the **ClassCastException** at runtime.

Generic Class:

We can define our own classes with generics type. A generic type is a class or interface that is parameterized over types. We use angle brackets **(< >)** to specify the **type parameter**.

To understand the benefit, let's see we have a simple class as:

```
public class GenericsTypeOld
{
        private Object t;
        public Object get()
        {
                return t;
        }
        public void set(Object t)
        {
                this.t = t;
        }
         public static void main(String args[])
        {
                GenericsTypeOld type = new GenericsTypeOld();
                type.set("Pankaj");
                String str = (String) type.get(); //type casting, error prone and can cause ClassCastException
        }
}
```

Notice that while using this class, we have to use type casting and it can produce **ClassCastException** at runtime. Now we will use java generic class to rewrite the same class as shown below.

```
public class GenericsType<T>
{
        private T t;
        public T get()
        {
                return this.t;
        }
        public void set(T t1)
        {
                this.t=t1;
        }
        public static void main(String args[])
        {
                GenericsType<String> type = new GenericsType<String>();
                type.set("Pankaj"); //valid
                GenericsType type1 = new GenericsType(); //raw type
                type1.set("Pankaj"); //valid
                type1.set(10); //valid and autoboxing support
        }
}
```

- ➢ Notice the use of GenericsType class in the main method. We don't need to **do type-casting** and we can remove **ClassCastException** at runtime.
- ➢ If we don't provide the type at the time of creation, compiler will produce a warning that "GenericsType is a raw type. References to generic type GenericsType<T> should be parameterized".
- ➢ When we don't provide type, the type becomes Object and hence it's allowing both String and Integer objects but we should always try to avoid this because we will have to use type casting while working on raw type that can produce runtime errors.

Tip: We can use **@SuppressWarnings("rawtypes")** annotation to suppress the compiler warning, check out java annotations tutorial.

- ➢ Also notice that it supports java **autoboxing**. That means, the numerical number is automatically convert into String.

Java Generic Type:

- ➢ Java Generic Type Naming convention helps us understanding code easily and having a naming convention is one of the best practices of java programming language.
- ➢ So generics also come with its own naming conventions.
- ➢ Usually type parameter names are **single, uppercase letters** to make it easily distinguishable from java variables.
- ➢ The most commonly used type parameter names are:

  - ❖ E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)

- ❖ K – Key (Used in Map)
- ❖ N – Number
- ❖ T – Type
- ❖ V – Value (Used in Map)
- ❖ S,U,V etc. – 2nd, 3rd, 4th types

Java Generic Method:

Sometimes we don't want whole class to be parameterized, in that case we can create java generics method. Since constructor is a special kind of method, we can use generics type in constructors too.

Here is a class showing example of java generic method.

```java
class GenericsType<T>
{
        private T t;
        public T get()
        {
                return this.t;
        }
        public void set(T t1)
        {
                this.t=t1;
        }
}
public class GenericsMethods
{
        //Java Generic Method
        public static <T> boolean isEqual(GenericsType<T> g1, GenericsType<T> g2)
        {
                return g1.get().equals(g2.get());
        }

        public static void main(String args[])
        {
                GenericsType<String> g1 = new GenericsType<>();
                g1.set("Pankaj");
                GenericsType<String> g2 = new GenericsType<>();
                g2.set("Pankaj");

                boolean isequal = GenericsMethods.<String>isEqual(g1, g2);
                //above statement can be written simply as
                isequal = GenericsMethods.isEqual(g1, g2);
                        System.out.println(isequal);

        }
}
```

Notice the isEqual method signature showing syntax to use generics type in methods. Also notice how to use these methods in our java program. We can specify type while calling these methods or we can invoke them like a normal method. Java compiler is smart enough to determine the type of variable to be used, this facility is called as **type inference**.

Wrapper classes:

Introduction:

Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object (with java.io.File), an image can be treated as an object (with java.awt.Image) and a simple data type can be converted into an object (with wrapper classes).

Wrapper classes are used to convert any data type into an object.

The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java language. For example, upto JDK1.4, the data structures accept only objects to store. A data type is to be converted into an object and then added to a Stack or Vector etc. For this conversion, the designers introduced **wrapper classes**.

➢ Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class.
➢ The wrapper classes are part of the **java.lang package**, which is imported by default into all Java programs.

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int  x = 25;
Integer  y = new Integer(33);
```
The first statement declares an int variable named x and initializes it with the value 25. The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.

Below table lists wrapper classes in Java API with constructor details.

Primitive	Wrapper Class	Constructor Argument
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String

double	Double	double or String
long	Long	long or String
short	Short	short or String

Below is wrapper class hierarchy as per Java API
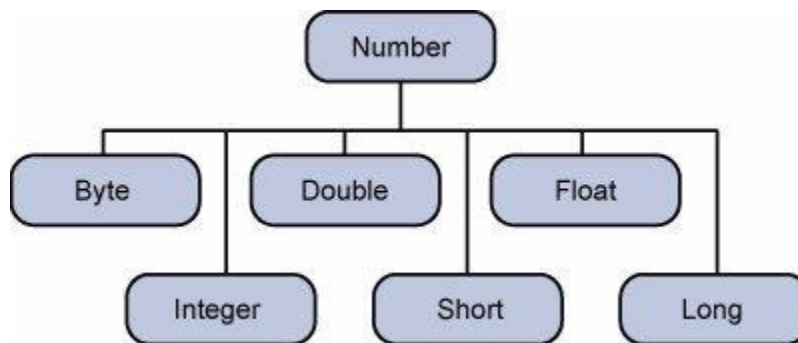
As explain in above table all wrapper classes (except Character) take String as argument constructor. Please note we might get NumberFormatException if we try to assign invalid argument in constructor. For example to create Integer object we can have following syntax.

Integer intObj = new Integer (25);
Integer intObj2 = new Integer ("25");

Here in we can provide any number as string argument but not the words etc. Below statement will throw run time exception (NumberFormatException)

Integer intObj3 = new Integer ("Two");

The following discussion focuses on the Integer wrapper class, but applies in a general sense to all eight wrapper classes.

The most common methods of the Integer wrapper class are summarized in below table. Similar methods for the other wrapper classes are found in the Java API documentation.

Method	Purpose
parseInt(s)	returns a signed decimal integer value equivalent to string s
toString(i)	returns a new String object representing the integer i
byteValue()	returns the value of this Integer as a byte
doubleValue()	returns the value of this Integer as an double

floatValue()	returns the value of this Integer as a float
intValue()	returns the value of this Integer as an int
shortValue()	returns the value of this Integer as a short
longValue()	returns the value of this Integer as a long
int compareTo(int i)	Compares the numerical value of the invoking object with that of i. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compare(int num1, int num2)	Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2.
boolean equals(Object intObj)	Returns true if the invoking Integer object is equivalent to intObj. Otherwise, it returns false.

Wrapping and Unwrapping:

int marks = 50;
Integer m1 = new Integer(marks);

In the above statement, an **int** data type **marks** is given an object form **m1** just by passing the variable to the constructor of **Integer** class. Wherever, **marks** is required as an object, **m1** can be used.
After using the Integer object in programming, now the programmer may require back the data type, as objects cannot be used in arithmetic operations. Now the object **m1** should be unwrapped.

int iv = m1.intValue();

For unwrapping, the method **intValue()** of Integer class used. The **int** value **iv** can be used in arithmetic operations.

Let's see java program which explain few wrapper classes methods.

```java
public class WrappingUnwrapping
{
 public static void main(String args[])
 {                                        //  data types
   byte grade = 2;
   int marks = 50;
   float price = 8.6f;                                // observe a suffix of <strong>f</strong> for float
   double rate = 50.5;
                                        // data types to objects
   Byte g1 = new Byte(grade);              // wrapping
   Integer m1 = new Integer(marks);
   Float f1 = new Float(price);
   Double r1 = new Double(rate);
                                        // let us print the values from objects
   System.out.println("Values of Wrapper objects (printing as objects)");
   System.out.println("Byte object g1:  " + g1);
   System.out.println("Integer object m1:  " + m1);
   System.out.println("Float object f1:  " + f1);
   System.out.println("Double object r1:  " + r1);
            // objects to data types (retrieving data types from objects)
   byte bv = g1.byteValue();          // unwrapping
   int iv = m1.intValue();
   float fv = f1.floatValue();
   double dv = r1.doubleValue();
                                        // let us print the values from data types
   System.out.println("Unwrapped values (printing as data types)");
   System.out.println("byte value, bv: " + bv);
   System.out.println("int value, iv: " + iv);
   System.out.println("float value, fv: " + fv);
   System.out.println("double value, dv: " + dv);
 }
}
```

```
C:\COMPUTED\system32\cmd.exe

C:\snr\way2java\javalang>java WrappingUnwrapping
Values of Wrapper objects (printing as objects)
Byte object g1:   2
Integer object m1:   50
Float object f1:   8.6
Double object r1:   50.5
Unwrapped values (printing as data types)
byte value, bv: 2
int value, iv: 50
float value, fv: 8.6
double value, dv: 50.5
```

As you can observe from the screenshot, constructors of wrapper classes are used to convert data types into objects and the methods of the form **XXXValue()** are used to retrieve back the data type from the object.

Serialization:

Java provides a mechanism, called object **serialization** where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

➢ Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.
➢ Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

```
public final void writeObject(Object x) throws IOException
```

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

```
public final Object readObject() throws IOException, ClassNotFoundException
```

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

To demonstrate how serialization works in Java, use the Employee class. Suppose that we have the following Employee class, which implements the Serializable interface:

```java
public class Employee implements java.io.Serializable
{
   public String name;
   public String address;
   public transient int SSN;
   public int number;
    public void mailCheck()
   {
     System.out.println("Mailing a check to " + name + " " + address);
   }
}
```

Notice that for a class to be serialized successfully, two conditions must be met:

- The class must implement the java.io.Serializable interface.
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked **transient**.

If you are curious to know if a Java Standard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements java.io.Serializable, then it is serializable; otherwise, it's not.

Serializing an Object:

The **ObjectOutputStream** class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

Note: When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

```java
import java.io.*;
public class SerializeDemo
{
  public static void main(String [] args)
  {
    Employee e = new Employee();
    e.name = "Phani Kumar";
    e.address = "Ongole";
    e.SSN = 11122333;
    e.number = 101;
    try
    {
      FileOutputStream fileOut = new FileOutputStream("H://employee.ser");
      ObjectOutputStream out = new ObjectOutputStream(fileOut);
      out.writeObject(e);
      out.close();
      fileOut.close();
      System.out.printf("Serialized data is saved in H://employee.ser");
    }
    catch(IOException i)
    {
      i.printStackTrace();
    }
  }
}
```

Deserializing an Object:

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output:

```java
import java.io.*;
public class DeserializeDemo
{
  public static void main(String [] args)
  {
    Employee e = null;
    try
    {
      FileInputStream fileIn = new FileInputStream("H://employee.ser");
      ObjectInputStream in = new ObjectInputStream(fileIn);
      e = (Employee) in.readObject();
      in.close();
      fileIn.close();
    }catch(IOException i)
    {
      i.printStackTrace();
      return;
    }catch(ClassNotFoundException c)
    {
      System.out.println("Employee class not found");
      c.printStackTrace();
      return;
    }
    System.out.println("Deserialized Employee...");
    System.out.println("Name: " + e.name);
    System.out.println("Address: " + e.address);
    System.out.println("SSN: " + e.SSN);
    System.out.println("Number: " + e.number);
  }
}
```

This would produce the following result:

```
Deserialized Employee...
Name: Phani Kumar
Address:Ongole
SSN: 0
Number:101
```

Here are following important points to be noted:

- The try/catch block tries to catch a **ClassNotFoundException**, which is declared by the **readObject()** method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
- Notice that the return value of **readObject()** is cast to an Employee reference.
- The value of the SSN field was 11122333 when the object was serialized, but because the field is **transient**, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

1. The Google File System

The Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

GFS provides a familiar file system interface, though itdoes not implement a standard API such as POSIX. Files areorganized hierarchically in directories and identified by path-names. We support the usual operations to create, delete,open, close, read, and write files.

Moreover, GFS has snapshot and record append operations. Snapshot creates a copy of a file or a directory treeat low cost. Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client's append

Architecture

A GFS cluster consists of a single master and multiple chunk servers and is accessed by multiple clients, as shown in Figure .

Each of these is typically a commodity Linux machine running a user-level server process. It is easy to run both a chunkserver and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.

Each of these is typically a commodity Linux machine running a user-level server process. It is easy to run both a chunkserver and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.
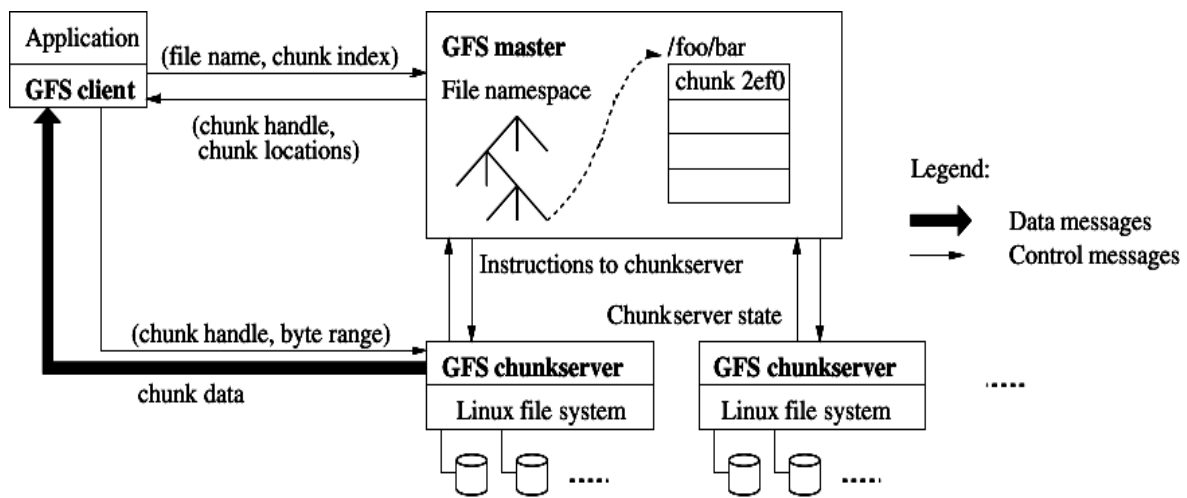
Figure 1: GFS Architecture

Files are divided into fixed-size chunks. Each chunk is identified by an immutable and globally unique 64 bit chunk handle assigned by the master at the time of chunk creation.

Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, we store three replicas, though users can designate different replication levels for different regions of the file namespace. The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks.

It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkserver in HeartBeat messages to give it instructions and collect its state.

2

GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application. Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers. We do not provide the POSIX API and therefore need not hook into the Linux vnode layer.

Neither the client nor the chunkserver caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues.(Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory.

Single Master

Having a single master vastly simplifies our design and enables the master to make sophisticated chunk placementApplication and replication decisions using global knowledge. However,we must minimize its involvement in reads and writes sothat it does not become a bottleneck. Clients never readand write file data through the master. Instead, a client asks the master which chunkservers it should contact.

Chunk Size

Chunk size is one of the key design parameters. We have chosen 64 MB, which is much larger than typical file system block sizes. Each chunk replica is stored as a plain Linux file on a chunkserver and is extended only as needed.

Lazy space allocation avoids wasting space due to internalfragmentation, perhaps the greatest objection against such a large chunk size.

A large chunk size offers several important advantages.

First, it reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information.

3

The reduction is especially significant for our work loads because applications mostly read and write large files sequentially.

Even for small random reads, the client can comfortably cache all the chunk location information for a multi-TB working set. Second, since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time.

Third, it reduces the size of the metadata stored on the master. This allows us to keep the metadata in memory, which in turn brings other advantages .

On the other hand, a large chunk size, even with lazy space allocation, has its disadvantages. A small file consists of a small number of chunks, perhaps just one. The chunkservers storing those chunks may become hot spots if many clients are accessing the same file. In practice, hot spots have not been a major issue because our applications mostly read large multi-chunk files sequentially.

However, hot spots did develop when GFS was first used by a batch-queue system: an executable was written to GFS as a single-chunk file and then started on hundreds of machines at the same time.

Metadata

The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk's replicas.

All metadata is kept in the master's memory. The first two types (namespaces and file-to-chunk mapping) are also kept persistent by logging mutations to an operation log stored on the master's local disk and replicated on remote machines. Using a log allows us to update the master state simply, reliably, and without risking inconsistencies in the event of a master crash. The master does not store chunk location information persistently. Instead, it asks each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster.

In-Memory Data Structures

Since metadata is stored in memory, master operations are fast. Furthermore, it is easy and efficient for the master to periodically scan through its entire state in the background.

This periodic scanning is used to implement chunk garbage collection, re-replication in the presence of chunkserver failures, and chunk migration to balance load and disk space usage across chunkservers.

Consistency Model

GFS has a relaxed consistency model that supports our highly distributed applications well but remains relatively simple and efficient to implement. We now discuss GFS's guarantees and what they mean to applications.

We also highlight how GFS maintains these guarantees but leave the details to other parts of the paper.

Guarantees by GFS

Operation Log

The operation log contains a historical record of critical metadata changes. It is central to GFS. Not only is it the only persistent record of metadata, but it also serves as a logical time line that defines the order of concurrent operations.

Advantages and disadvantages of large sized chunks in Google File System

Chunks size is one of the key design parameters. In GFS it is 64 MB, which is much larger than typical file system blocks sizes. Each chunk replica is stored as a plain Linux file on a chunk server and is extended only as needed.

Advantages

1. It reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information.

5

2. Since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persistent TCP connection to the chunk server over an extended period of time.

3. It reduces the size of the metadata stored on the master. This allows us to keep the metadata in memory, which in turn brings other advantages.

Disadvantages

1. Lazy space allocation avoids wasting space due to internal fragmentation.

2. Even with lazy space allocation, a small file consists of a small number of chunks, perhaps just one. The chunk servers storing those chunks may become hot spots if many clients are accessing the same file. In practice, hot spots have not been a major issue because the applications mostly read large multi-chunk files sequentially. To mitigate it, replication and allowance to read from other clients can be done.

2. Hadoop Distributed File System (HDFS) Building blocks of Hadoop :

A. Namenode
B. Datanode
C. Secondary Name node
D. JobTracker
E. TaskTracker

Hadoop is made up of 2 parts:

1. HDFS – Hadoop Distributed File System
2. MapReduce – The programming model that is used to work on the data present in HDFS.

HDFS – Hadoop Distributed File System

HDFS is a file system that is written in Java and resides within the user space unlike traditional file systems like FAT, NTFS, ext2, etc that reside on the kernel space. HDFS was primarily written to store large amounts of data (terrabytes and petabytes). HDFS was built inline with Google's paper on GFS.

6

MapReduce

MapReduce is the programming model that uses Java as the programming language to retrieve data from files stored in the HDFS. All data in HDFS is stored as files. Even MapReduce was built inline with another paper by Google.

Google, apart from their papers did not release their implementations of GFS and MapReduce. However, the Open Source Community built Hadoop and MapReduce based on those papers.

The initial adoption of Hadoop was at Yahoo Inc., where it gained good momentum and went onto be a part of their production systems. After Yahoo, many organizations like LinkedIn, Facebook, Netflix and many more have successfully implemented Hadoop within their organizations.

Hadoop uses HDFS to store files efficiently in the cluster. When a file is placed in HDFS it is broken down into blocks, 64 MB block size by default. These blocks are then replicated across the different nodes (*DataNodes*) in the cluster. The default replication value is 3, i.e. there will be 3 copies of the same block in the cluster. We will see later on why we maintain replicas of the blocks in the cluster.

A Hadoop cluster can comprise of a single node (single node cluster) or thousands of nodes.

Once you have installed Hadoop you can try out the following few basic commands to work with HDFS:

- hadoop fs -ls
- hadoop fs -put <path_of_local> <path_in_hdfs>
- hadoop fs -get <path_in_hdfs> <path_of_local>
- hadoop fs -cat <path_of_file_in_hdfs>
- hadoop fs -rmr <path_in_hdfs>

  the different components of a Hadoop Cluster are:

  NameNode (Master) – NameNode, Secondary NameNode, JobTracker
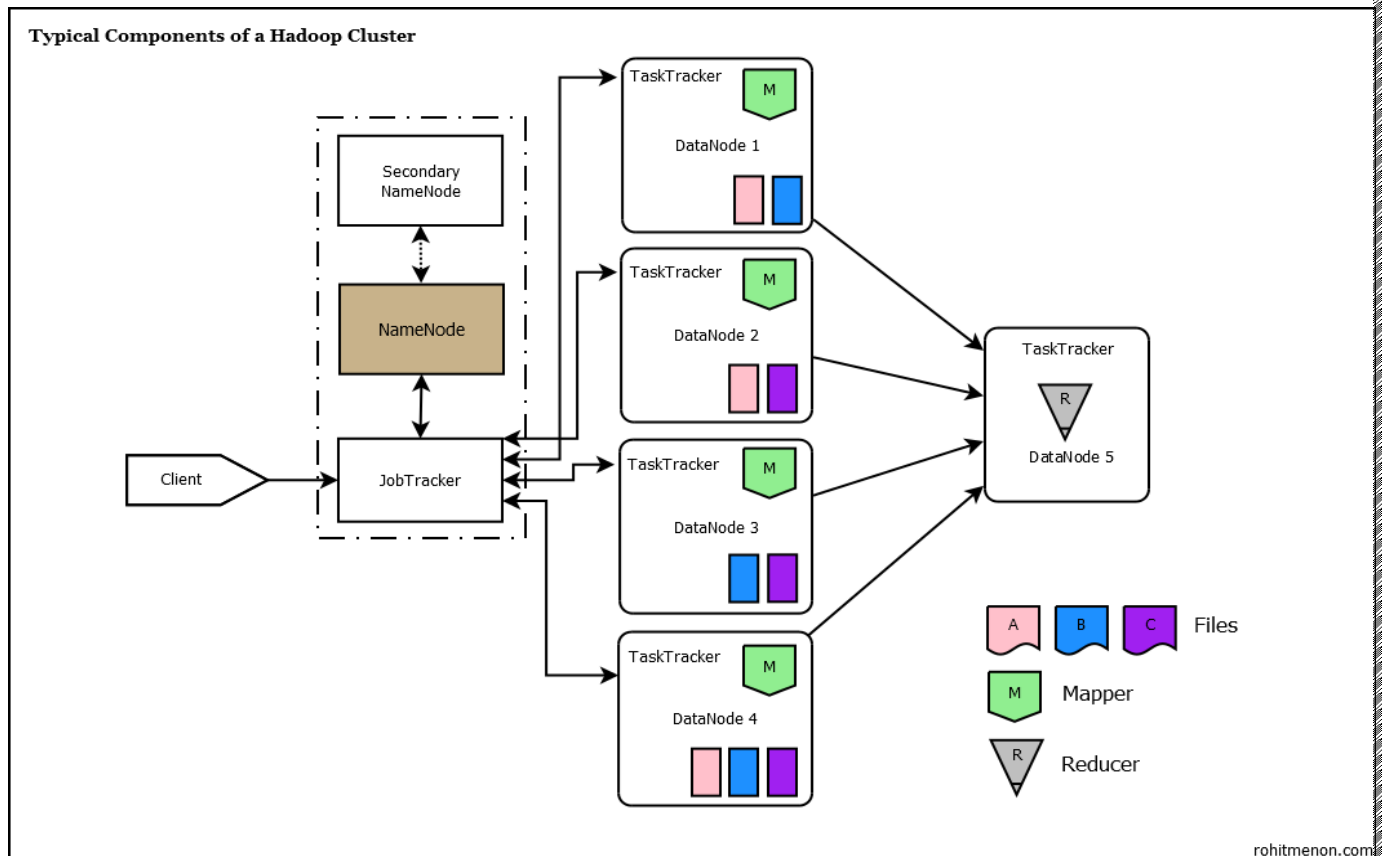
  DataNode 1 (Slave) – TaskTracker, DataNode

  DataNode 2 (Slave) – TaskTracker, DataNode

  DataNode 3 (Slave) – TaskTracker, DataNode

  DataNode 4 (Slave) – TaskTracker, DataNode

  DataNode 5 (Slave) – TaskTracker, DataNode

7

Typical Components of a Hadoop Cluster

The above diagram depicts a 6 Node Hadoop Cluster

In the diagram you see that the *NameNode*, *Secondary NameNode* and the*JobTracker* are running on a single machine. Usually in production clusters having more those 20-30 nodes, the daemons run on separate nodes.

Hadoop follows a Master-Slave architecture. As mentioned earlier, a file in HDFS is split into blocks and replicated across *Datanodes* in a Hadoop cluster. You can see that the three files A, B and C have been split across with a replication factor of 3 across the different *Datanodes*.

Now let us go through each node and daemon:

NameNode

The NameNode in Hadoop is the node where Hadoop stores all the location information of the files in HDFS. In other words, it holds the metadata for HDFS. Whenever a file is placed in the cluster a corresponding entry of it location is maintained by the NameNode. So, for the files A, B and C we would have something as follows in the NameNode:

File A – DataNode1, DataNode2, DataNode4

8

File B – DataNode1, DataNode3, DataNode4

File C – DataNode2, DataNode3, DataNode4

This information is required when retrieving data from the cluster as the data is spread across multiple machines. **The NameNode is a Single Point of Failure for the Hadoop Cluster.**

Secondary NameNode

IMPORTANT – The *Secondary NameNode* is not a failover node for the *NameNode*.

The secondary name node is responsible for performing periodic housekeeping functions for the *NameNode*. It only creates checkpoints of the file system present in the *NameNode*.

DataNode

The *DataNode* is responsible for storing the files in HDFS. It manages the file blocks within the node. It sends information to the *NameNode* about the files and blocks stored in that node and responds to the *NameNode* for all filesystem operations.

JobTracker

JobTracker is responsible for taking in requests from a client and assigning *TaskTrackers* with tasks to be performed. The *JobTracker* tries to assign tasks to the *TaskTracker* on the *DataNode* where the data is locally present (Data Locality). If that is not possible it will at least try to assign tasks to *TaskTrackers* within the same rack. If for some reason the node fails the *JobTracker* assigns the task to another *TaskTracker* where the replica of the data exists since the data blocks are replicated across the *DataNodes*. This ensures that the job does not fail even if a node fails within the cluster.

TaskTracker

TaskTracker is a daemon that accepts tasks (Map, Reduce and Shuffle) from the *JobTracker*. The *TaskTracker* keeps sending a heart beat message to the *JobTracker* to notify that it is alive. Along with the heartbeat it also sends the free slots available within it to process tasks. *TaskTracker* starts and monitors the Map & Reduce Tasks and sends progress/status information back to the *JobTracker*.

All the above daemons run within have their own JVMs.

A typical (simplified) flow in Hadoop is a follows:

1. A Client (usaually a MapReduce program) submits a job to the *JobTracker*.

2. The *JobTracker* get information from the *NameNode* on the location of the data within the *DataNodes*. The *JobTracker* places the client program (usually a jar file along with the

configuration file) in the HDFS. Once placed, *JobTracker* tries to assign tasks to *TaskTrackers* on the DataNodes based on data locality.

3. The *TaskTracker* takes care of starting the Map tasks on the *DataNodes* by picking up the client program from the shared location on the HDFS.

4. The progress of the operation is relayed back to the *JobTracker* by the *TaskTracker*.

5. On completion of the Map task an intermediate file is created on the local filesystem of the *TaskTracker*.

6. Results from Map tasks are then passed on to the Reduce task.

7. The Reduce tasks works on all data received from map tasks and writes the final output to HDFS.

8. After the task complete the intermediate data generated by the *TaskTracker* is deleted.

   A very important feature of Hadoop to note here is, that, the program goes to where the data is and not the way around, thus resulting in efficient processing of data.

3. Introducing and Configuring Hadoop cluster

A. Local

B. Pseudo-distributed mode

C. Fully Distributed mode

Hadoop is supported by GNU/Linux platform and its flavors. Therefore, we have to install a Linux operating system for setting up Hadoop environment.

Pre-installation Setup

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

Creating a User

At the beginning, it is recommended to create a separate user for Hadoop to isolate Hadoop file system from Unix file system. Follow the steps given below to create a user:

- Open the root using the command "su".
- Create a user from the root account using the command "useradd username".
- Now you can open an existing user account using the command "su username".

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

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

SSH Setup and Key Generation

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

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

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

Installing Java

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

```
$ java -version
```

If everything is in order, it will give you the following output.

```
java version "1.7.0_71"
Java(TM) SE Runtime Environment (build 1.7.0_71-b13)
Java HotSpot(TM) Client VM (build 25.0-b02, mixed mode)
```

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

Step 1

Download java (JDK <latest version> - X64.tar.gz) by visiting the following linkhttp://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads1880260.html.

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

Step 2

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

```
$ cd Downloads/
$ ls
jdk-7u71-linux-x64.gz
$ tar zxf jdk-7u71-linux-x64.gz
$ ls
jdk1.7.0_71   jdk-7u71-linux-x64.gz
```

Step 3

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

```
$ su
password:
# mv jdk1.7.0_71 /usr/local/
# exit
```

Step 4

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

```
export JAVA_HOME=/usr/local/jdk1.7.0_71
export PATH=$PATH:$JAVA_HOME/bin
```

Now apply all the changes into the current running system.

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

Step 5

Use the following commands to configure java alternatives:

12

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

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

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

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

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

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

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

Downloading Hadoop

Download and extract Hadoop 2.4.1 from Apache software foundation using the following commands.

```
$ su
password:
# cd /usr/local
# wget http://apache.claz.org/hadoop/common/hadoop-2.4.1/
hadoop-2.4.1.tar.gz
# tar xzf hadoop-2.4.1.tar.gz
# mv hadoop-2.4.1/* to hadoop/
# exit
```

Hadoop Operation Modes

Once you have downloaded Hadoop, you can operate your Hadoop cluster in one of the three supported modes:

- **Local/Standalone Mode** : After downloading Hadoop in your system, by default, it is configured in a standalone mode and can be run as a single java process.
- **Pseudo Distributed Mode** : It is a distributed simulation on single machine. Each Hadoop daemon such as hdfs, yarn, MapReduce etc., will run as a separate java process. This mode is useful for development.
- **Fully Distributed Mode** : This mode is fully distributed with minimum two or more machines as a cluster. We will come across this mode in detail in the coming chapters.

Installing Hadoop in Standalone Mode

Here we will discuss the installation of **Hadoop 2.4.1** in standalone mode.

13

There are no daemons running and everything runs in a single JVM. Standalone mode is suitable for running MapReduce programs during development, since it is easy to test and debug them.

Setting Up Hadoop

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

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

Before proceeding further, you need to make sure that Hadoop is working fine. Just issue the following command:

```
$ hadoop version
```

If everything is fine with your setup, then you should see the following result:

```
Hadoop 2.4.1
Subversion https://svn.apache.org/repos/asf/hadoop/common -r 1529768
Compiled by hortonmu on 2013-10-07T06:28Z
Compiled with protoc 2.5.0
From source with checksum 79e53ce7994d1628b240f09af91e1af4
```

It means your Hadoop's standalone mode setup is working fine. By default, Hadoop is configured to run in a non-distributed mode on a single machine.

Example

Let's check a simple example of Hadoop. Hadoop installation delivers the following example MapReduce jar file, which provides basic functionality of MapReduce and can be used for calculating, like Pi value, word counts in a given list of files, etc.

```
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.2.0.jar
```

Let's have an input directory where we will push a few files and our requirement is to count the total number of words in those files. To calculate the total number of words, we do not need to write our MapReduce, provided the .jar file contains the implementation for word count. You can try other examples using the same .jar file; just issue the following commands to check supported MapReduce functional programs by hadoop-mapreduce-examples-2.2.0.jar file.

```
$ hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduceexamples-2.2.0.jar
```

14

Step 1

Create temporary content files in the input directory. You can create this input directory anywhere you would like to work.

```
$ mkdir input
$ cp $HADOOP_HOME/*.txt input
$ ls -l input
```

It will give the following files in your input directory:

```
total 24
-rw-r--r-- 1 root root 15164 Feb 21 10:14 LICENSE.txt
-rw-r--r-- 1 root root 101 Feb 21 10:14 NOTICE.txt
-rw-r--r-- 1 root root 1366 Feb 21 10:14 README.txt
```

These files have been copied from the Hadoop installation home directory. For your experiment, you can have different and large sets of files.

Step 2

Let's start the Hadoop process to count the total number of words in all the files available in the input directory, as follows:

```
$ hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduceexamples-2.2.0.jar
wordcount input output
```

Step 3

Step-2 will do the required processing and save the output in output/part-r00000 file, which you can check by using:

```
$cat output/*
```

It will list down all the words along with their total counts available in all the files available in the input directory.

```
"AS     4
"Contribution" 1
"Contributor" 1
"Derivative 1
"Legal 1
```

```
"License"     1
"License");   1
"Licensor"    1
"NOTICE"       1
"Not    1
"Object"      1
"Source"      1
"Work"   1
"You"   1
"Your")  1
"[]"    1
"control"     1
"printed     1
"submitted"    1
(50%)    1
(BIS),   1
(C)     1
(Don't)  1
(ECCN)   1
(INCLUDING    2
(INCLUDING,   2
.............
```

Installing Hadoop in Pseudo Distributed Mode

Follow the steps given below to install Hadoop 2.4.1 in pseudo distributed mode.

Step 1: Setting Up Hadoop

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

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

```
export HADOOP_COMMON_HOME=$HADOOP_HOME

export HADOOP_HDFS_HOME=$HADOOP_HOME

export YARN_HOME=$HADOOP_HOME

export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native

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

export HADOOP_INSTALL=$HADOOP_HOME
```

Now apply all the changes into the current running system.

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

Step 2: Hadoop Configuration

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

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

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

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

The following are the list of files that you have to edit to configure Hadoop.

core-site.xml

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

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

```
<configuration>

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

17

```
   </property>


</configuration>
```

hdfs-site.xml

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

Let us assume the following data.

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

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

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

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

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

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

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

```
<configuration>

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


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


  <property>
    <name>dfs.data.dir</name>
```

```
    <value>file:///home/hadoop/hadoopinfra/hdfs/datanode </value>
  </property>


</configuration>
```

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

yarn-site.xml

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

```
<configuration>


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


</configuration>
```

mapred-site.xml

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

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

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

```
<configuration>


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

</configuration>

Verifying Hadoop Installation

The following steps are used to verify the Hadoop installation.

Step 1: Name Node Setup

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

```
$ cd ~
$ hdfs namenode -format
```

The expected result is as follows.

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

Step 2: Verifying Hadoop dfs

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

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

The expected output is as follows:

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

Step 3: Verifying Yarn Script

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

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

The expected output as follows:

```
starting yarn daemons
starting resourcemanager, logging to /home/hadoop/hadoop
2.4.1/logs/yarn-hadoop-resourcemanager-localhost.out
localhost: starting nodemanager, logging to /home/hadoop/hadoop
2.4.1/logs/yarn-hadoop-nodemanager-localhost.out
```
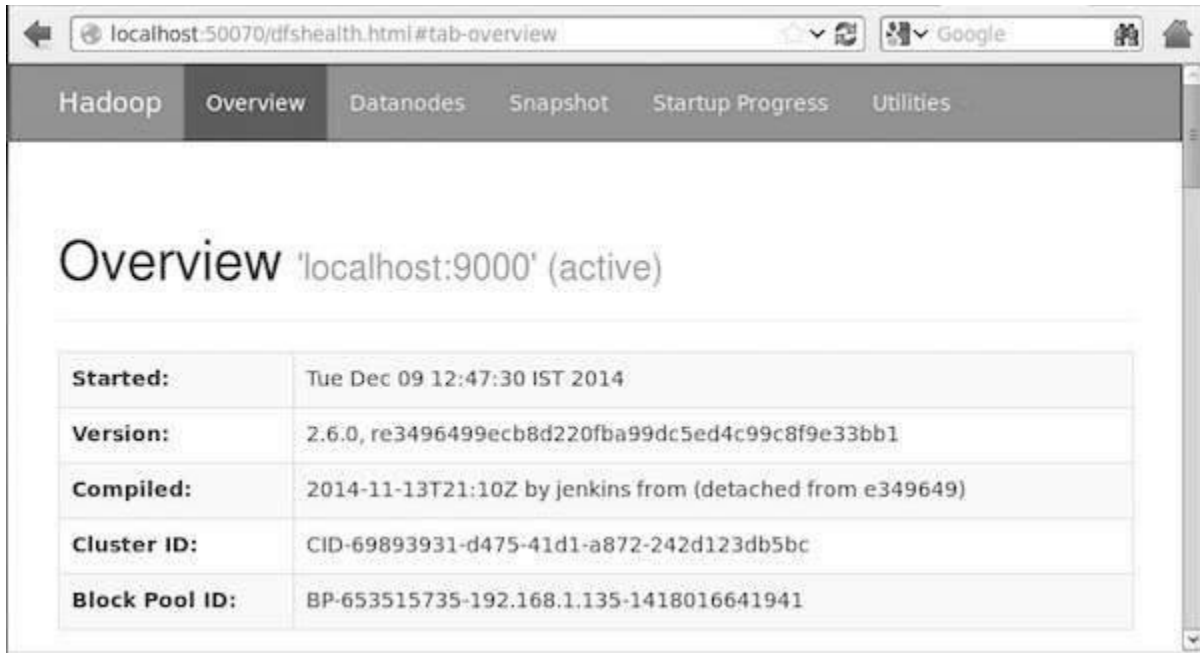
Step 4: Accessing Hadoop on Browser

The default port number to access Hadoop is 50070. Use the following url to get Hadoop services on browser.
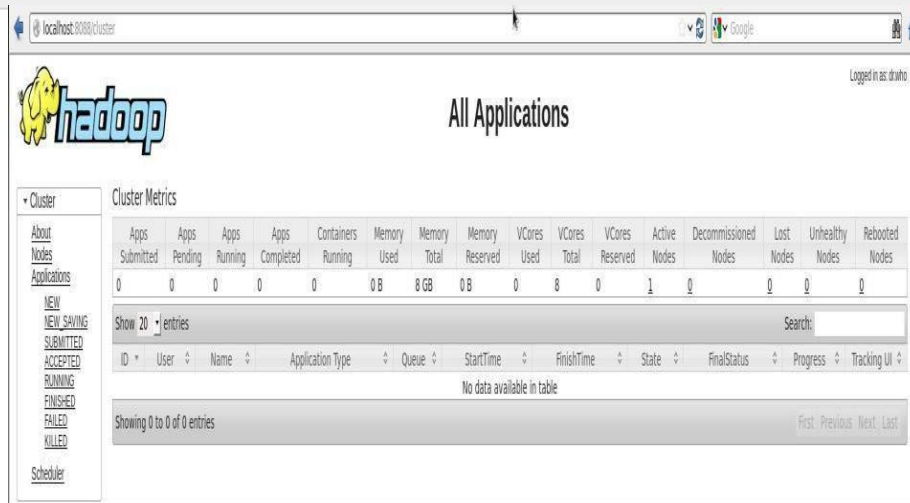
```
http://localhost:50070/
```

Step 5: Verify All Applications for Cluster

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

http://localhost:8088/

C .Fully Distributed mode

Compatibility Requirements

S.No	Category	Supported
1	Languages	Java, Python, Perl, Ruby etc.
2	Operating System	Linux (Server Deployment) Mostly preferred, Windows (Development only), Solaris.
3	Hardware	32 bit Linux (64 bit for large deployment)

Installation Items

S.No	Item	Version
1	jdk-6u25-linux-i586.bin	Java 1.6 or higher
2	hadoop-0.20.2-cdh3u0.tar.gz	Hadoop 0.20.2

Note: Both Items are required to be installed on Namenode and Datanode machines

Installation Requirements

S.No	Requirement	Reason
1	Operating system – Linux recommended for server deployment (Production env.)	
2	Language – Java 1.6 or higher	
3	Ram – at least 3 GB/node	
4	Hard disk – at least 1 TB	For namenode machine.
5	Should have root credentials	For changing some system files you need admin permissions.

High level Steps

Step #	Activity	Check
1	Binding IP address with the host name under /etc/hosts	
2	Setting passwordless SSH	
3	Installing Java	
4	Installing Hadoop	
5	Setting JAVA HOME and HADOOP HOME variables	
6	Updating .bash_profile file for hadoop	
7	Creating required folders for namenode and datanode	
8	Configuring the .xml files	
9	Setting the masters and slaves in all the machines	
10	Formatting the namenode	
11	Starting the Dfs services and mapred services	
12	Stopping all services	

Binding IP address with the host names

Before starting the installation of hadoop, first you need to bind the IP address of the machines along with their host names under /etc/hosts file.

First check the hostname of your machine by using following command :

$ hostname

Open /etc/hosts file for binding IP with the hostname

$ vi /etc/hosts

Provide ip & hostname of the all the machines in the cluster

e.g: 10.11.22.33 hostname1

10.11.22.34   hostname2

Setting Passwordless SSh login

SSH is used to login from one system to another without requiring passwords. This will be required when you run a cluster, it will not prompt you for the password again and again.

First log in on Host1 **(hostname of namenode machine)** as hadoop user and generate a pair of authentication keys. Command is:

hadoop@Host1$ ssh-keygen –t rsa
Note: Give the hostname which you got in step 5.3.1. Do not enter any passphrase if asked.
Now use ssh to create a directory   ~/.ssh as user hadoop on Host2 **(Hostname other than namenode machine).**

hadoop@Host1$ ssh hadoop@Host2 mkdir –p .ssh
hadoop@Host2's password:

Finally append Host1's new public key to hadoop@Host2: .ssh/authorized_keys and enter Host2's password one last time:

hadoop@Host1$ cat   /home/hadoop/.ssh/id_rsa.pub | ssh hadoop@Host2   ' cat >> .ssh/authorized_keys '
hadoop@Host2's password:

From now on you can log into Host2 as hadoop from Host1 without password:
hadoop@Host1$ ssh hadoop@Host2
Host2@hadoop$

NOTE: Do the following changes:
☐   Change the permissions of .ssh to 700
☐   Change the permissions of .ssh/authorized_keys to 640

Prepare for installation

Check for previous installed versions of java and hadoop on your machine

$ rpm –qa | grep java

It will display **fully qualified paths** of the version installed.

Remove all the previous version of Java and Hadoop installed on the machine.

$ rpm –e softwarename or path-name

NOTE: All the installations and extractions are being done in /home/hadoop/

Installing Java

Use the JDK bin file (jdk-6u25-linux-i586.bin) for installing java on your machine . Copy the .bin file in /home/hadoop/

Execute the command "**./jdk-6u25-linux-i586.bin**" in /home/hadoop/ (which will unzip the contents into folder **jdk1.6.0_25**)

Extract the hadoop package

Syntax

$ tar –xzvf < hadoop-tar-package>

$ tar –xzvf hadoop-0.20.0-cdh3u9.tar.gz

Configuring HADOOP_HOME

Check whether HADOOP_HOME is set up to the folder containing hadoop_core_VERSION.jar using

$ echo $HADOOP_HOME

If not set then set it

$ export HADOOP_HOME=/home/hadoop/hadoop-version

For e.g.

$ cd /home/hadoop/

$ export HADDOP_HOME= /home/hadoop/hadoop-0.20.2-cdh3u0/

Setting JAVA_HOME

$ cd /home/hadoop/hadoop-0.20.0-cdh3u0/conf/

$ vi hadoop-env.sh

hadoop-env.sh file for setting JAVA_HOME

Press :wq to save and exit the file

You need to change the bash file also .

$ vi ~/.bash_profile

```
# Set the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi


# User specific environment and startup programs
export HADOOP_HOME="/home/hadoop/hadoop-0.20.2-cdh3u0"
export JAVA_HOME="/home/hadoop/jdk1.6.0_25"
export PATH=$JAVA_HOME/bin:$HADOOP_HOME/bin:$PATH
export CLASSPATH=$CLASSPATH:$JAVA_HOME/lib
#Adding Flume path
export PATH=$PATH:/home/hadoop/flume/flume-0.9.2/bin
~
```

bash_profile file for setting environment variables and jar files

Check for hadoop installation confirmation

Run hadoop command to confirm whether the installation is successful.

$ cd <hadoop-home-directory>

Standard Path

$ cd /home/hadoop/ hadoop-0.20.0-cdh3u0/

$ bin/hadoop

On successful installation you should get the following message.

```
[hadoop@         hadoop-0.20.2-cdh3u0]$ bin/hadoop
Usage: hadoop [--config confdir] COMMAND
where COMMAND is one of:
  namenode -format     format the DFS filesystem
  secondarynamenode    run the DFS secondary namenode
  namenode             run the DFS namenode
  datanode             run a DFS datanode
  dfsadmin             run a DFS admin client
  mradmin              run a Map-Reduce admin client
  fsck                 run a DFS filesystem checking utility
  fs                   run a generic filesystem user client
  balancer             run a cluster balancing utility
  fetchdt              fetch a delegation token from the NameNode
  jobtracker           run the MapReduce job Tracker node
  pipes                run a Pipes job
  tasktracker          run a MapReduce task Tracker node
  job                  manipulate MapReduce jobs
  queue                get information regarding JobQueues
  version              print the version
  jar <jar>            run a jar file
  distcp <srcurl> <desturl> copy file or directories recursively
  archive -archiveName NAME -p <parent path> <src>* <dest> create a hadoop archive
  oiv                  apply the offline fsimage viewer to an fsimage
  classpath            prints the class path needed to get the
                       Hadoop jar and the required libraries
  daemonlog            get/set the log level for each daemon
 or
  CLASSNAME            run the class named CLASSNAME
Most commands print help when invoked w/o parameters.
[hadoop@hydlab3 hadoop-0.20.2-cdh3u0]$
```

CONFIGURING HADOOP IN FULLY DISTRIBUTED MODE

Create the dfs.name.dir local directories on namenode machine

$ cd /home/hadoop/

$ mkdir -p data/1/dfs/nn

Creating the directories for storing the Data blocks and the temporary directory for storing

process ids on datanode machines

$ cd /home/hadoop/

$ mkdir –p data/1/dfs/dn data/2/dfs/dn data/3/dfs/dn

$ mkdir –p /home/hadoop/ tmp

Creating the directories for storing the temporary data (Task Tracker) and the system files for

Map/Reduce jobs

$ cd /home/hadoop/

$ mkdir –p data/1/mapred/local data/2/mapred/local data/3/mapred/local

$ mkdir –p /home/hadoop/mapred/system

Give full permission to all folder under /home/hadoop/

$ cd /home/hadoop/

$ chmod 777 *

Navigate to /home/hadoop/hadoop-0.20.0-cdh3u0/conf directory

$ cd /home/hadoop/hadoop-0.20.0-cdh3u0/conf

Set up the configuration files under /home/hadoop/hadoop-0.20.0-cdh3u0/conf/

Core-Site.xml

$ vi core-site.xml

Parameters of core-site.xml

fs.default.name ☐ URL for the Name Node

hadoop.tmp.dir☐ URL for the temporary data

dfs.replication☐ This property specifies the number of times the file has to be replicated on

cluster.

```
<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://132.31.???.241:9020</value>
</property>
<property>
<name>hadoop.tmp.dir</name>
<value>/home/hadoop/tmp</value>
</property>
<property>
<name>mapred.job.tracker</name>
<value>???.31.129.24?:9010</value>
</property>
<property>
<name>dfs.replication</name>
<value>3</value>
</property>
</configuration>
```

core-site.xml

hdfs-site.xml

$ vi hdfs-site.xml

Parameters of hdfs-site.xml

dfs.name.dir☐ This property specifies the directories where the NameNode stores its metadata

and edit logs. Represented by the /home/hadoop/data/1/dfs/nn path examples.

dfs.data.dir☐ This property specifies the directories where the DataNode stores blocks.

Represented   by   the   /home/hadoop/data/1/dfs/dn,   /home/hadoop/data/2/dfs/dn   ,

/home/hadoop/data/3/dfs/dn

```
<configuration>
<property>
<name>dfs.name.dir</name>
<value>/home/hadoop/data/1/dfs/nn</value>
</property>
<property>
<name>dfs.data.dir</name>
<value>/home/hadoop/data/1/dfs/dn,/home/hadoop/data/2/dfs/dn,/home/hadoop/data/3/dfs/dn</value>
</property>
</configuration>
```

hdfs-site.xml

Press :wq to save and exit the file

mapred-site.xml

$ vi mapred-site.xml

Parameters of mapred-site.xml

30

mapred.local.dir ☐ This property specifies the directories where the TaskTracker will store temporary data and intermediate map output files while running Map Reduce jobs.

Eg./home/hadoop/data/1/mapred/local,/home/hadoop/data/2/mapred/local,
/home/hadoop/data/3/mapred/local.

mapred.system.dir ☐Path on the HDFS where the Map Reduce framework stores system files e.g./home/hadoop/mapred/system/.

mapred.job.tracker ☐ Host or IP and port of Job Tracker.

```
<configuration>
<property>
<name>mapred.local.dir</name>
<value>/home/hadoop/data/1/mapred/local,home/hadoop/data/2/mapred/local,/home/hadoop/data/3/mapred/local</value>
</property>
<property>
    <name>mapred.job.tracker</name>
    <value>172.21.129.241:9010</value>
</property>
<property>
    <name>mapred.system.dir</name>
    <value>/home/hadoop/mapred/system</value>
</property>
</property>
</configuration>
```

mapred-site.xml

Press :wq to save and exit the file

Set the correct owner and permissions of the local directories:

Directory	Owner	Permissions
dfs.name.dir	hdfs:hadoop	drwx------
dfs.data.dir	hdfs:hadoop	drwx------
mapred.local.dir	mapred:hadoop	drwxr-xr-x

$ chmod 700 /home/hadoop/data/1/dfs/nn/

$ chmod   700   /home/hadoop/data/1/dfs/dn/   /home/hadoop/data/2/dfs/dn/

/home/hadoop/data/3/dfs/dn/
$ chmod 755 /home/hadoop/data/1/mapred/local/
/home/hadoop/data/2/mapred/local/ /home/hadoop/data/3/mapred/local/

Setting up the masters and slaves

vi conf/masters

hostname of machine acting as a SecondaryNamenode

vi slaves

hostname of machines acting as a Datanode & TaskTrackers

Formatting the namenode

You need to format the namenode every time you start the dfs services. This is because every time you start the services it causes some files to be written in the namenode folder which may get duplicated when you run the services for the second time. Do not format a running Hadoop namenode, otherwise it will cause all your data in the HDFS filesytem to be erased.

$ cd /home/hadoop/hadoop-0.20.0-cdh3u0/

$ bin/hadoop namenode -format

Note : Give **"Y"** when it asks for re-format

Starting dfs service

Run the command

$ /bin/start-dfs.sh on the machine you want the namenode to run on. This will bring up HDFS with the namenode running on the machine you ran the previous command on, and datanodes on the machines listed in the conf/slaves file.

$ cd /home/hadoop/hadoop-0.20.0-cdh3u0/

$ ./bin/start-dfs.sh

NOTE: For any problems check the log files under all the machines under /home/hadoop/hadoop-0.20.2-cdh3u0/logs/ and refer to the troubleshooting guide for the same.

Starting mapred service

32

For mapred services run the following command on the machine you want jobtracker to run on
(in my case it was namenode machine)
you can choose other machine also.

$ ./bin/start-mapred.sh
Checking the DFS service report

$ ./bin/hadoop dfsadmin –report
Checking on web interface **DFS SERVICE**
http://ip-address of namenode machine:50070/
Checking on web interface **Mapred Job**
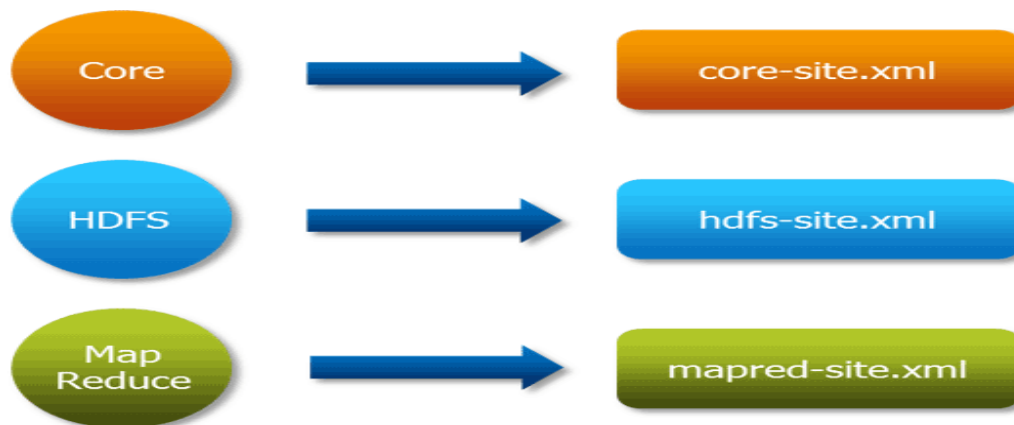**http://ip of namenode machine:50030/**
Stopping dfs and mapred services
 cd /home/hadoop/hadoop-0.20.0-cdh3u0/
$ ./bin/stop-all.sh

    4   Configuring XML files.

 Hadoop Cluster Configuration Files

Configuration Filenames	Description of Log Files
hadoop-env.sh	Environment variables that are used in the scripts to run Hadoop.
core-site.xml	Configuration settings for Hadoop Core such as I/O settings that are common to HDFS and MapReduce.
hdfs-site.xml	Configuration settings for HDFS daemons, the namenode, the secondary namenode and the data nodes.
mapred-site.xml	Configuration settings for MapReduce daemons : the job-tracker and the task-trackers.
masters	A list of machines (one per line) that each run a secondary namenode.
slaves	A list of machines (one per line) that each run a datanode and a task-tracker.

All these files are available under '**conf**' directory of Hadoop installation directory.

Here is a listing of these files in the File System:

```
ubuntu@ip-10-251-81-223:~$ ls
hadoop-1.2.0   hadoop-1.2.0.tar.gz   keypairs
ubuntu@ip-10-251-81-223:~$ unalias -a
ubuntu@ip-10-251-81-223:~$ cd
ubuntu@ip-10-251-81-223:~$ ls
hadoop-1.2.0  hadoop-1.2.0.tar.gz  keypairs
ubuntu@ip-10-251-81-223:~$ cd hadoop-1.2.0/
ubuntu@ip-10-251-81-223:~/hadoop-1.2.0$ ls
bin             hadoop-ant-1.2.0.jar            ivy           sbin
build.xml       hadoop-client-1.2.0.jar         ivy.xml       share
c++             hadoop-core-1.2.0.jar           lib           src
CHANGES.txt     hadoop-examples-1.2.0.jar       libexec       webapps
conf            hadoop-minicluster-1.2.0.jar    LICENSE.txt
contrib         hadoop-test-1.2.0.jar           NOTICE.txt
docs            hadoop-tools-1.2.0.jar          README.txt
ubuntu@ip-10-251-81-223:~/hadoop-1.2.0$ cd conf/
ubuntu@ip-10-251-81-223:~/hadoop-1.2.0/conf$ ls
capacity-scheduler.xml      hadoop-policy.xml       slaves
configuration.xsl           hdfs-site.xml           ssl-client.xml.example
core-site.xml               log4j.properties        ssl-server.xml.example
fair-scheduler.xml          mapred-queue-acls.xml   taskcontroller.cfg
hadoop-env.sh               mapred-site.xml         task-log4j.properties
hadoop-metrics2.properties  masters
ubuntu@ip-10-251-81-223:~/hadoop-1.2.0/conf$
```

Let's look at the files and their usage one by one!

hadoop-env.sh

This file specifies environment variables that affect the JDK used by **Hadoop Daemon** (bin/hadoop).

As Hadoop framework is written in *Java* and uses *Java Runtime environment*, one of the important environment variables for Hadoop daemon is **$JAVA_HOME** in **hadoop-env.sh**. This variable directs Hadoop daemon to the *Java path* in the system.

34

```
# The java implementation to use.  Required.
export JAVA_HOME=/usr/lib/jvm/java-1.6.0-openjdk-amd64
```

This file is also used for setting another *Hadoop daemon execution environment* such as **heap size (HADOOP_HEAP)**, **hadoop home (HADOOP_HOME),** **log file location (HADOOP_LOG_DIR),** etc.

Note: For the simplicity of understanding the cluster setup, we have configured only necessary parameters to start a cluster.

The following three files are the important configuration files for the runtime environment settings of a Hadoop cluster.

core-site.sh

This file informs Hadoop daemon where NameNode runs in the cluster. It contains the configuration settings for Hadoop Core such as I/O settings that are common to *HDFS* and *MapReduce*.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://ec2-54-214-206-65.us-west-2.compute.amazonaws.com:8020</value>
</property>
</configuration>
```
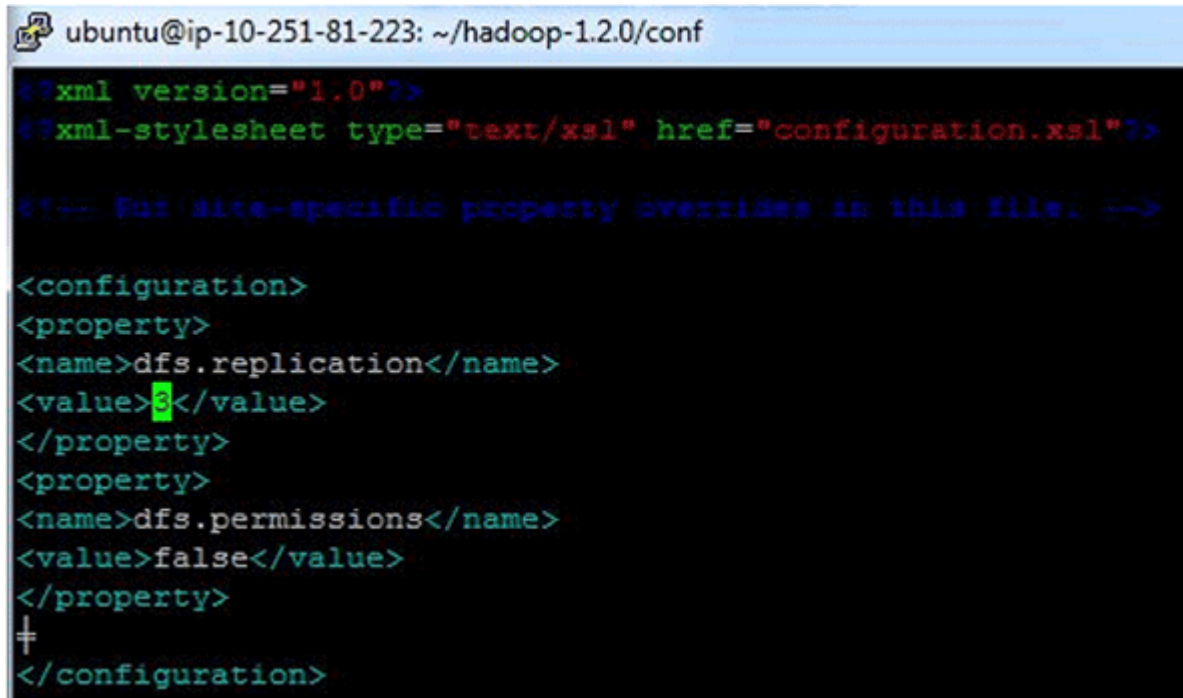
Where *hostname* and *port* are the machine and port on which NameNode daemon runs and listens. It also informs the Name Node as to which IP and port it should bind. The commonly used port is **8020** and you can also specify IP address rather than hostname.

hdfs-site.sh

This file contains the configuration settings for *HDFS daemons; the Name Node, the Secondary Name Node, and the data nodes.*

You can also configure **hdfs-site.xml** to specify default block replication and permission checking on HDFS. The actual number of replications can also be specified when the file is created. The default is used if replication is not specified in create time.

35

The value "true" for property **'dfs.permissions'** enables permission checking in HDFS and the value "false" turns off the permission checking. Switching from one parameter value to the other does not change the mode, owner or group of files or directories.
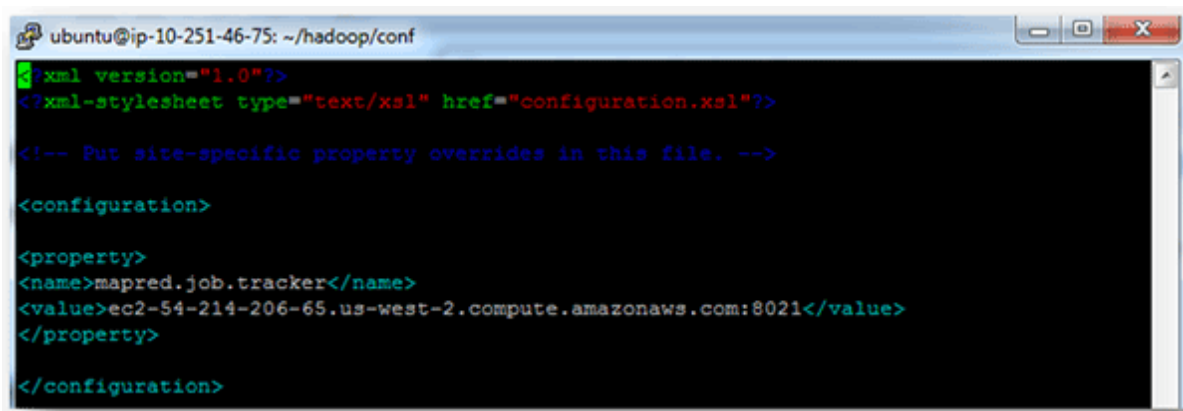
```
ubuntu@ip-10-251-81-223: ~/hadoop-1.2.0/conf

<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
<name>dfs.replication</name>
<value>3</value>
</property>
<property>
<name>dfs.permissions</name>
<value>false</value>
</property>

</configuration>
```

mapred-site.sh

This file contains the configuration settings for MapReduce daemons; *the job tracker and the task-trackers*. The **mapred.job.tracker** parameter is a *hostname* (or IP address) and *port* pair on which the Job Tracker listens for RPC communication. This parameter specify the location of the Job Tracker to Task Trackers and MapReduce clients.

```
ubuntu@ip-10-251-46-75: ~/hadoop/conf

<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>
<name>mapred.job.tracker</name>
<value>ec2-54-214-206-65.us-west-2.compute.amazonaws.com:8021</value>
</property>

</configuration>
```

36

You can replicate all of the four files explained above to all the Data Nodes and Secondary Namenode. These files can then be configured for any node specific configuration e.g. in case of a different **JAVA HOME** on one of the Datanodes.

The following two file 'masters' and 'slaves' determine the master and salve Nodes in Hadoop cluster.

Masters

This file informs about the Secondary Namenode location to hadoop daemon. The '**masters**' file at Master server contains a hostname Secondary Name Node servers.

The 'masters' file on **Slave Nodes** is blank.

Slaves

The '**slaves**' file at Master node contains a list of hosts, one per line, that are to host Data Node and Task Tracker servers.

The '**slaves**' file on Slave server contains the IP address of the slave node. Notice that the 'slaves' file at Slave node contains only its own IP address and not of any other Data Nodes in the cluster.

UNIT-III

MapReduce is a programming model for data processing. The model is simple, yet not too simple to express useful programs in. Hadoop can run MapReduce programs written in various languages; Most important, MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal. MapReduce comes into its own for large datasets, so let's start by looking at one.

A Weather Dataset

For our example, we will write a program that mines weather data. Weather sensors collecting data every hour at many locations across the globe gather a large volume of log data, which is a good candidate for analysis with MapReduce, since it is semistructured and record-oriented.

Data Format

The data we will use is from the National Climatic Data Center (NCDC, *http://www .ncdc.noaa.gov/*). The data is stored using a line-oriented ASCII format, in which each line is a record. The format supports a rich set of meteorological elements, many of which are optional or with variable data lengths. For simplicity, we shall focus on the basic elements, such as temperature, which are always present and are of fixed width.

Example shows a sample line with some of the salient fields highlighted. The line has been split into multiple lines to show each field: in the real file, fields are packed into one line with no delimiters.

Data files are organized by date and weather station. There is a directory for each year from 1901 to 2001, each containing a gzipped file for each weather station with its readings for that year. For example, here are the first entries for 1990:

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
```

Since there are tens of thousands of weather stations, the whole dataset is made up of a large number of relatively small files. It's generally easier and more efficient to process a smaller number of relatively large files, so the data was preprocessed so that each year's readings were concatenated into a single file.

Analyzing the Data with Hadoop

To take advantage of the parallel processing that Hadoop provides, we need to express our query as a MapReduce job. After some local, small-scale testing, we will be able to run it on a cluster of machines.

Map and Reduce

MapReduce works by breaking the processing into **two phases**: the **map phase** and the **reduce phase**. Each phase has **key-value pairs as input and output**, the types of which

---

may be chosen by the programmer. The programmer also specifies two functions: the **map function** and the **reduce function**.

The input to our map phase is the raw NCDC data. We choose a text input format that gives us each line in the dataset as a text value. The key is the offset of the beginning of the line from the beginning of the file, but as we have no need for this, we ignore it.

Our map function is simple. We pull out the year and the air temperature, since these are the only fields we are interested in. In this case, the map function is just a data preparation phase, setting up the data in such a way that the reducer function can do its work on it: finding the maximum temperature for each year. The map function is also a good place to drop bad records: here we filter out temperatures that are missing, suspect, or erroneous.

To visualize the way the map works, consider the following sample lines of input data (some unused columns have been dropped to fit the page, indicated by ellipses):
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...

These lines are presented to the map function as the key-value pairs:
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)

The keys are the line offsets within the file, which we ignore in our map function. The map function merely extracts the year and the air temperature (indicated in bold text), and emits them as its output (the temperature values have been interpreted as integers):

(1950, 0)
(1950, 22)
(1950, −11)
(1949, 111)
(1949, 78)

The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing **sorts and groups the key-value pairs** by key. So, continuing the example, our reduce function sees the following input:

(1949, [111, 78])
(1950, [0, 22, −11])

Each year appears with a list of all its air temperature readings. All the reduce function has to do now is iterate through the list and pick up the maximum reading:

(1949, 111)
(1950, 22)

This is the final output: the maximum global temperature recorded in each year.

The whole data flow is illustrated in the bellow Figure. At the bottom of the diagram is a Unix pipeline
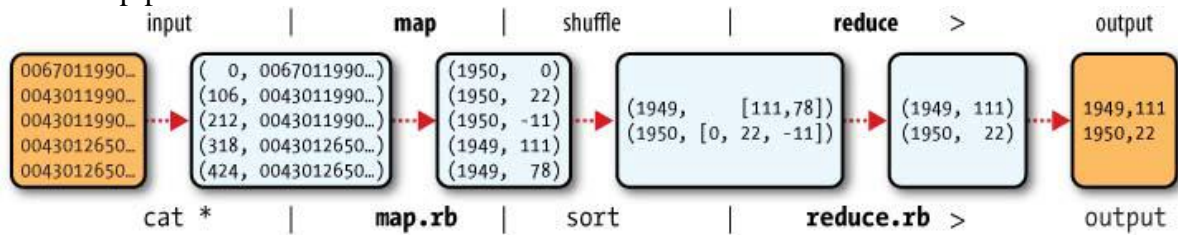
Figure 2-1. MapReduce logical data flow

Java MapReduce

Having run through how the MapReduce program works, the next step is to express it in code. We **need three things: a map function, a reduce function, and some code to run the job**. The map function is represented by an implementation of the Mapper interface, which declares a map() method. Example -1 shows the implementation of our map function.

Example - 1. Mapper for maximum temperature example

```java
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 MaxTemperatureMapper extends MapReduceBase implements
                                        Mapper <LongWritable, Text, Text, IntWritable>
{
private static final int MISSING = 9999;
public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
Reporter reporter) throws IOException
{
String line = value.toString();
String year = line.substring(15, 19);
int airTemperature;
if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
airTemperature = Integer.parseInt(line.substring(88, 92));
} else {
airTemperature = Integer.parseInt(line.substring(87, 92));
}
String quality = line.substring(92, 93);
if (airTemperature != MISSING && quality.matches("[01459]")) {
output.collect(new Text(year), new IntWritable(airTemperature));
}
}
}
```

www.Jntufastupdates.com

The Mapper interface is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function. For the present example, the input key is a long integer offset, the input value is a line of text, the output key is a year, and the output value is an air temperature (an integer). Rather than use built-in Java types, Hadoop provides its own set of basic types that are optimized for network serialization. These are found in the **org.apache.hadoop.io package**.

Here we use **LongWritable**, which corresponds to a Java Long, **Text** (like Java String), **and IntWritable** (like Java Integer).

The map() method is passed a key and a value. We convert the Text value containing the line of input into a Java String, then use its substring() method to extract the columns we are interested in.

The map() method also provides an instance of OutputCollector to write the output to. In this case, we write the year as a Text object (since we are just using it as a key), and the temperature is wrapped in an IntWritable. We write an output record only if the temperature is present and the quality code indicates the temperature reading is OK.

The reduce function is similarly defined using a Reducer, as illustrated in Example - 2.

Example - 2. Reducer for maximum temperature example

```
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 MaxTemperatureReducer extends MapReduceBase
                                    implements Reducer<Text, IntWritable, Text, IntWritable>
{
public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException
{
int maxValue = Integer.MIN_VALUE;
while (values.hasNext()) {
maxValue = Math.max(maxValue, values.next().get());
}
output.collect(key, new IntWritable(maxValue));
}
}
```

Again, four formal type parameters are used to specify the input and output types, this time for the reduce function. The input types of the reduce function must match the output types of the map function: Text and IntWritable. And in this case, the output types of the reduce function are Text and IntWritable, for a year and its maximum temperature, which we find by iterating through the temperatures and comparing each with a record of the highest found so far.

The third piece of code runs the MapReduce job (see Example -3).

Example -3. Application to find the maximum temperature in the weather dataset

```
import java.io.IOException;
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;
public class MaxTemperature
{
public static void main(String[] args) throws IOException
{
if (args.length != 2)
{
System.err.println("Usage: MaxTemperature <input path> <output path>");
System.exit(-1);
}
JobConf conf = new JobConf(MaxTemperature.class);
conf.setJobName("Max temperature");
FileInputFormat.addInputPath(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
conf.setMapperClass(MaxTemperatureMapper.class);
conf.setReducerClass(MaxTemperatureReducer.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
JobClient.runJob(conf);
}
}
```

A **JobConf** object forms the specification of the job. It gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster). Rather than explicitly specify the name of the JAR file, we can pass a class in the JobConf constructor, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.

Having constructed a JobConf object, we specify the input and output paths. An input path is specified by calling the static **addInputPath()** method on **FileInputFormat**, and it can be a single file, a directory (in which case, the input forms all the files in that directory), or a file pattern. As the name suggests, **addInputPath()** can be called more than once to use input from multiple paths.

The output path (of which there is only one) is specified by the static **setOutput Path()** method on **FileOutputFormat**. It specifies a directory where the output files from the reducer functions are written. **The directory shouldn't exist before running the job**, as Hadoop will complain and not run the job. This precaution is to prevent data loss (it can be very annoying to accidentally overwrite the output of a long job with another).

Next, we specify the map and reduce types to use via the **setMapperClass()** and **setReducerClass()** methods. The **setOutputKeyClass()** and **setOutputValueClass()** methods control the output types for the map and the reduce functions, which are often the same, as they are in our case.

If they are different, then the map output types can be set using the methods setMapOutputKeyClass() and setMapOutputValueClass(). The input types are controlled via the input format, which we have not explicitly set since we are using the default TextInputFormat. After setting the classes that define the map and reduce functions, we are ready to run the job.

The static **runJob()** method on **JobClient** submits the job and waits for it to finish, writing information about its progress to the console.

The output was written to the *output* directory, which contains one output file per reducer. The job had a single reducer, so we find a single file, named ***part-00000***:

% **cat output/part-00000**
1949 111
1950 22

This result is the same as when we went through it by hand earlier. We interpret this as saying that the maximum temperature recorded in 1949 was 11.1°C, and in 1950 it was 2.2°C.

The new Java MapReduce API:

Release 0.20.0 of Hadoop included a new Java MapReduce API, sometimes referred to as "Context Objects," designed to make the API easier to evolve in the future. The new API is type-incompatible with the old, however, so applications need to be rewritten to take advantage of it.

There are several notable differences between the two APIs:
➢ The new API favours abstract classes over interfaces, since these are easier to evolve. For example, you can add a method (with a default implementation) to an abstract class without breaking old implementations of the class. In the new API, the Mapper and Reducer interfaces are now abstract classes.
➢ The new API is in the **org.apache.hadoop.mapreduce** package (and subpackages). The old API can still be found in **org.apache.hadoop.mapred**.
➢ The new API makes extensive use of context objects that allow the user code to communicate with the MapReduce system. The MapContext, for example, essentially unifies the role of the JobConf, the OutputCollector, and the Reporter.
➢ The new API supports both a "push" and a "pull" style of iteration. In both APIs, key-value record pairs are pushed to the mapper, but in addition, the new API allows a mapper to pull records from within the map() method. The same goes for the reducer. An example of how the "pull" style can be useful is processing records in batches, rather than one by one.
➢ Configuration has been unified. The old API has a special JobConf object for job configuration, which is an extension of Hadoop's vanilla Configuration object. In the new API, this distinction is dropped, so job configuration is done through a Configuration.
➢ Job control is performed through the Job class, rather than JobClient, which no longer exists in the new API.

➢ Output files are named slightly differently: ***part-m-nnnnn*** for map outputs, and ***partr-nnnnn*** for reduce outputs (where ***nnnnn*** is an integer designating the part number, starting from **zero**).

When converting your Mapper and Reducer classes to the new API, don't forget to change the signature of the map() and reduce() methods to the new form. Just changing your class to extend the new Mapper or Reducer classes will *not* produce a compilation error or warning, since these classes provide an identity form of the map() or reduce() method (respectively). Your mapper or reducer code, however, will not be invoked, which can lead to some hard-to-diagnose errors.

Combiner

A **Combiner**, also known as a **semi-reducer,** is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

The main function of a Combiner is to summarize the map output records with the same key. The output (key-value collection) of the combiner will be sent over the network to the actual Reducer task as input.

Combiner class

The Combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Map and Reduce. Usually, the output of the map task is large and the data transferred to the reduce task is high.

The following MapReduce task diagram shows the COMBINER PHASE.

www.Jntufastupdates.com
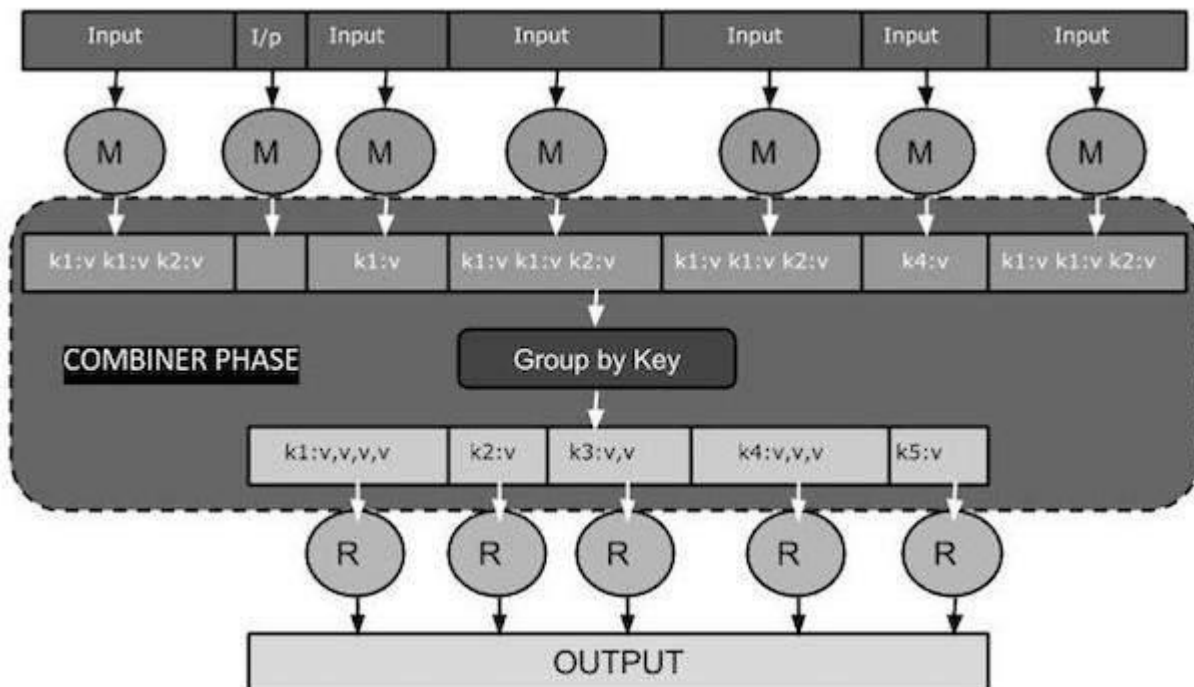
How Combiner Works?

Here is a brief summary on how MapReduce Combiner works −

- A combiner does not have a predefined interface and it must implement the Reducer interface's reduce() method.

- A combiner operates on each map output key. It must have the same output key-value types as the Reducer class.

- A combiner can produce summary information from a large dataset because it replaces the original Map output.

Although, Combiner is optional yet it helps segregating data into multiple groups for Reduce phase, which makes it easier to process.

MapReduce Combiner Implementation

The following example provides a theoretical idea about combiners. Let us assume we have the following input text file named **input.txt** for MapReduce.

What do you mean by Object
What do you know about Java
What is Java Virtual Machine
How Java enabled High Performance

The important phases of the MapReduce program with Combiner are discussed below.

Record Reader

This is the first phase of MapReduce where the Record Reader reads every line from the input text file as text and yields **output as key-value pairs**.

Input − Line by line text from the input file.

Output − Forms the key-value pairs. The following is the set of expected key-value pairs.

<1, What do you mean by Object>
<2, What do you know about Java>
<3, What is Java Virtual Machine>
<4, How Java enabled High Performance>

Map Phase

The Map phase takes input from the Record Reader, processes it, and produces the output as another set of key-value pairs.

Input − The following key-value pair is the input taken from the Record Reader.

<1, What do you mean by Object>
<2, What do you know about Java>
<3, What is Java Virtual Machine>
<4, How Java enabled High Performance>

The Map phase reads each key-value pair, divides each word from the value using **StringTokenizer**, and treats each word as key and the count of that word as value. The following code snippet shows the Mapper class and the map function.

```java
public static class WordCountMapper extends Mapper<Object, Text, Text, IntWritable>
{
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(Object key, Text value, Context context) throws IOException,
InterruptedException
  {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens())
    {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

Output − The expected output is as follows −

```
<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>
<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>
<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>
<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>
```

Combiner Phase

The Combiner phase takes each key-value pair from the Map phase, processes it, and produces the output as **key-value collection** pairs.

Input − The following key-value pair is the input taken from the Map phase.

```
<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>
<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>
<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>
<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>
```

The Combiner phase reads each key-value pair, **combines the common words as key** and values as collection. Usually, the code and operation for a **Combiner is similar to that of a Reducer**. Following is the code snippet for Mapper, Combiner and Reducer class declaration.

```java
job.setMapperClass(WordCountMapper.class);
job.setCombinerClass(WordCountReducer.class);
job.setReducerClass(WordCountReducer.class);
```

Output − The expected output is as follows −

```
<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,1,1,1>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>
```

Partitioner Phase

The partitioning phase takes place after the **map phase** and before the **reduce phase**. The number of **partitions is equal to the number of reducers**. The data gets partitioned across the reducers according to the partitioning function.

The difference between a partitioner and a combiner is that **the partitioner** divides the data according to the number of reducers so that all the data in a single partition gets executed by a single reducer. However, **the combiner** functions similar to the reducer and processes the data in each partition. The combiner is an optimization to the reducer.

The default partitioning function is the hash partitioning function where the hashing is done on the key. However it might be useful to partition the data according to some other function of the key or the value.

Reducer Phase

The Reducer phase takes each key-value collection pair from the Combiner phase, processes it, and passes the output as key-value pairs. Note that the Combiner functionality is same as the Reducer.

Input − The following key-value pair is the input taken from the Combiner phase.

```
<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,1,1,1>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>
```

The Reducer phase reads each key-value pair. Following is the code snippet for the Combiner.

```java
public static class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable>
{
  private IntWritable result = new IntWritable();
  public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException
  {
    int sum = 0;
    for (IntWritable val : values)
    {
```

```
    sum += val.get();
  }
  result.set(sum);
  context.write(key, result);
 }
}
```

Output − The expected output from the Reducer phase is as follows –

```
<What,3> <do,2> <you,2> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,3>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>
```

Record Writer

This is the last phase of MapReduce where the Record Writer writes every key-value pair from the Reducer phase and sends the output as text.

Input − Each key-value pair from the Reducer phase along with the Output format.

Output − It gives you the key-value pairs in text format. Following is the expected output.

```
What        3
do          2
you         2
mean        1
by          1
Object      1
know        1
about       1
Java        3
is          1
Virtual     1
Machine     1
How         1
enabled     1
High        1
Performance 1
```
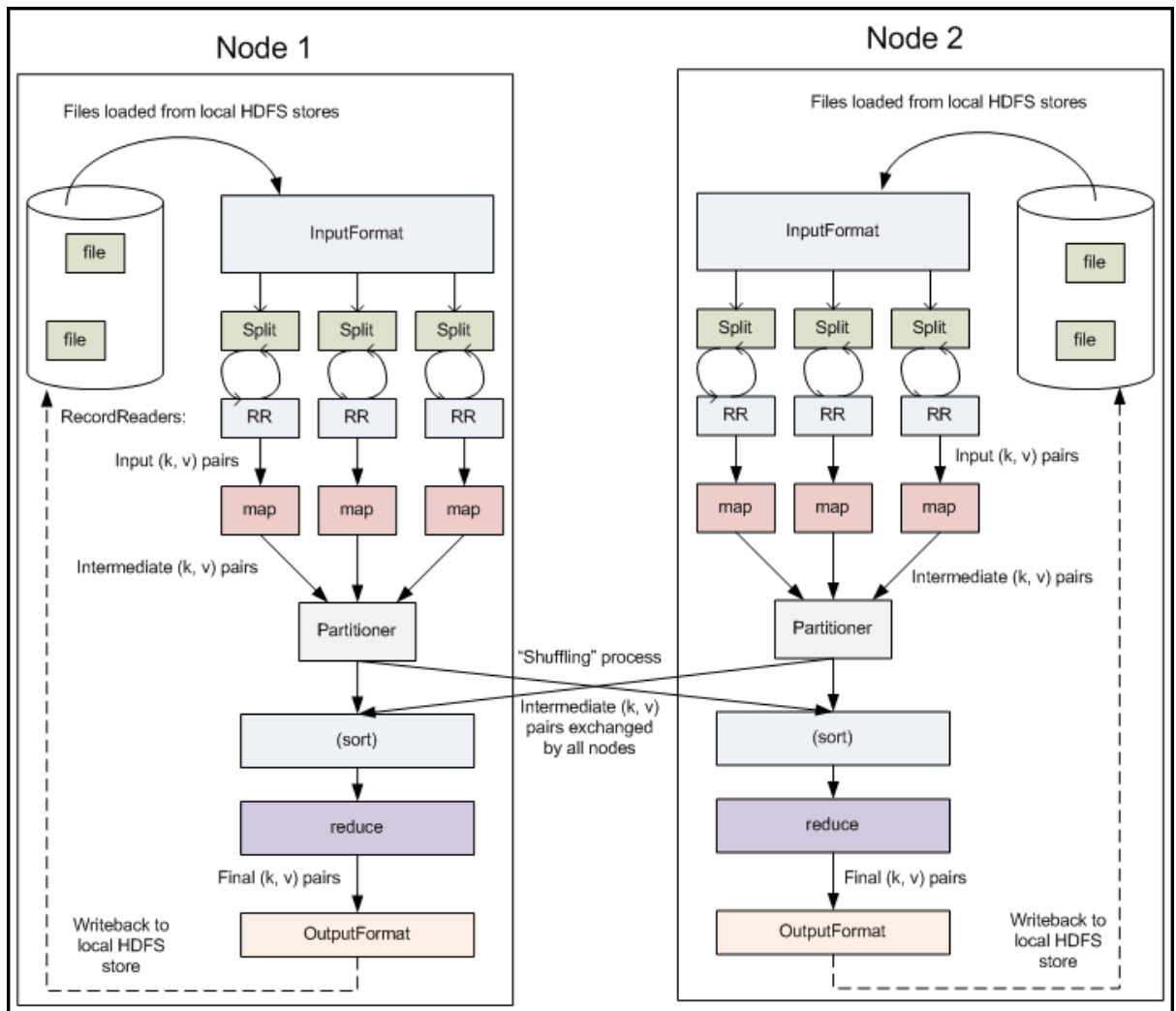
Unit 4: Hadoop I/O: The Writable Interface, WritableComparable and comparators, Writable Classes: Writable wrappers for Java primitives, Text, BytesWritable, NullWritable, ObjectWritable and GenericWritable, Writable collections, Implementing a Custom Writable: Implementing a RawComparator for speed, Custom comparators

Reference: Hadoop: The Definitive Guide by Tom White, 3rd Edition, O'reilly

Unit 4

1. Hadoop I/O:

   - The Writable Interface,

   - WritableComparable and comparators,

2. Writable Classes:

   - Writable wrappers for Java primitives,

   - Text,

   - BytesWritable,

   - NullWritable,

   - ObjectWritable

   - GenericWritable,

   - Writable collections,

3. Implementing a Custom Writable:

   - Implementing a Raw Comparator for speed,

   - Custom comparators

1. Hadoop I/O

The Writable Interface

Writable in an interface in Hadoop and types in Hadoop must implement this interface. Hadoop provides these writable wrappers for almost all Java primitive types and some other types,but sometimes we need to pass custom objects and these custom objects should implement Hadoop's Writable interface. Hadoop MapReduce uses implementations of Writables for interacting with user-provided Mappers and Reducers.

To implement the Writable interface we require two methods:

```
public interface Writable {

void readFields(DataInput in);

void write(DataOutput out);
}
```

Why use Hadoop Writable(s)?

As we already know, data needs to be transmitted between different nodes in a distributed computing environment. This requires serialization and de-serialization of data to convert the data that is in structured format to byte stream and vice-versa. Hadoop therefore uses simple and efficient serialization protocol to serialize data between map and reduce phase and these are called Writable(s). Some of the examples of writables as already mentioned before are IntWritable, LongWritable, BooleanWritable and FloatWritable.

WritableComparable interface is just a sub interface of the Writable and java.lang.Comparable interfaces. For implementing a WritableComparable we must have compareTo method apart from readFields and write methods, as shown below:

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

Comparison of types is crucial for MapReduce, where there is a sorting phase during which keys are compared with one another.

Implementing a comparator for WritableComparables like the org.apache.hadoop.io.RawComparator interface will definitely help speed up your Map/Reduce

(MR) Jobs. As you may recall, a MR Job is composed of receiving and sending key-value pairs. The process looks like the following.

(K1,V1) –> Map –> (K2,V2)
(K2,List[V2]) –> Reduce –> (K3,V3)

The key-value pairs (K2,V2) are called the intermediary key-value pairs. They are passed from the mapper to the reducer. Before these intermediary key-value pairs reach the reducer, a shuffle and sort step is performed.

The shuffle is the assignment of the intermediary keys (K2) to reducers and the sort is the sorting of these keys. In this blog, by implementing the RawComparator to compare the intermediary keys, this extra effort will greatly improve sorting. Sorting is improved because the RawComparator will compare the keys by byte. If we did not use RawComparator, the intermediary keys would have to be completely deserialized to perform a comparison.

Note (In Short):
1)WritableComparables can be compared to each other, typically via Comparators. Any type which is to be used as a key in the Hadoop Map-Reduce framework should implement this interface.

2) Any type which is to be used as a value in the Hadoop Map-Reduce framework should implement the Writable interface.

Writables and its Importance in Hadoop

Writable is an interface in Hadoop. Writable in Hadoop acts as a wrapper class to almost all the primitive data type of Java. That is how int of java has become IntWritable in Hadoop and String of Java has become Text in Hadoop.

Writables are used for creating serialized data types in Hadoop. So, let us start by understanding what data type, interface and serilization is.

Data Type

A data type is a set of data with values having predefined characteristics. There are several kinds of data types in Java. For example- int, short, byte, long, char etc. These are called as primitive data types. All these primitive data types are bound to classes called as wrapper class. For example int, short, byte, long are grouped under INTEGER which is a wrapper class. These wrapper classes are predefined in the Java.

Interface in Java

An interface in Java is a complete abstract class. The methods within an interface are abstract methods which do not accept body and the fields within the interface are public, static and final, which means that the fields cannot be modified.

The structure of an interface is most likely to be a class. We cannot create an object for an interface and the only way to use the interface is to implement it in other class by using 'implements' keyword.

Serialization

Serialization is nothing but converting the raw data into a stream of bytes which can travel along different networks and can reside in different systems. Serialization is not the only concern of Writable interface; it also has to perform compare and sorting operation in Hadoop.

Why are Writables Introduced in Hadoop?

Now the question is whether Writables are necessary for Hadoop. Hadoop frame work definitely needs Writable type of interface in order to perform the following tasks:

Implement serialization, Transfer data between clusters and networks

Store the de-serialized data in the local disk of the system

Implementation of writable is similar to implementation of interface in Java. It can be done by simply writing the keyword 'implements' and overriding the default writable method.

Writable is a strong interface in Hadoop which while serializing the data, reduces the data size enormously, so that data can be exchanged easily within the networks. It has separate read and write fields to read data from network and write data into local disk respectively. Every data inside Hadoop should accept writable and comparable interface properties.

We have seen how Writables reduces the data size overhead and make the data transfer easier in the network.

What if Writable were not there in Hadoop?

Let us now understand what happens if Writable is not present in Hadoop.

Serialization is important in Hadoop because it enables easy transfer of data. If Writable is not present in Hadoop, then it uses the serialization of Java which increases the data over-head in the network.

smallInt serialized value using Java serializer

aced0005737200116a6176612e6c616e672e496e74656765 7212e2a0a4f781873802000149000576616c7565787200106a6176612e 6c616e672e4e756d62657286ac951d0b94e08b020000787000000064

smallInt serialized value using IntWritable

00000064

This shows the clear difference between serialization in Java and Hadoop and also the difference between ObjectInputStream and Writable interface. If the size of serialized data in Hadoop is like that of Java, then it will definitely become an overhead in the network.

Also the core part of Hadoop framework i.e., shuffle and sort phase won't be executed without using Writable.

How can Writables be Implemneted in Hadoop?

Writable variables in Hadoop have the default properties of Comparable. For example:

When we write a key as IntWritable in the Mapper class and send it to the reducer class, there is an intermediate phase between the Mapper and Reducer class i.e., shuffle and sort, where each key has to be compared with many other keys. If the keys are not comparable, then shuffle and sort phase won't be executed or may be executed with high amount of overhead.

If a key is taken asIntWritable by default, then it has comparable feature because of RawComparator acting on that variable. It will compare the key taken with the other keys in the network. This cannot take place in the absence of Writable.

Can we make custom Writables? The answer is definitely 'yes'. We can make our own custom Writable type.

Let us now see how to make a custom type in Java.

The steps to make a custom type in Java is as follows:

```
public class add {

    int a;

    int b;

    public add() {

        this.a = a;

        this.b = b;

    }

}
```

Similarly we can make a custom type in Hadoop using Writables.

For implementing Writables, we need few more methods in Hadoop:

public interface Writable {

void readFields(DataInput in);

void write(DataOutput out);

}

Here, readFields, reads the data from network and write will write the data into local disk. Both are necessary for transferring data through clusters. DataInput and DataOutput classes (part of java.io) contain methods to serialize the most basic types of data.

Suppose we want to make a composite key in Hadoop by combining two Writables then follow the steps below:

```
public class add implements Writable{
public int a;

public int b;

public add(){

this.a=a;

this.b=b;

}

public void write(DataOutput out) throws IOException {

  out.writeInt(a);

  out.writeInt(b);

 }

public void readFields(DataInput in) throws IOException {

  a = in.readInt();
```

```
    b = in.readInt();

  }

 public String toString() {

    return Integer.toString(a) + ", " + Integer.toString(b)

 }

}
```

Thus we can create our custom Writables in a way similar to custom types in Java but with two additional methods, write and readFields. The custom writable can travel through networks and can reside in other systems.

This custom type cannot be compared with each other by default, so again we need to make them comparable with each other.

Let us now discuss what is WritableComparable and the solution to the above problem.

As explained above, if a key is taken as IntWritable, by default it has comparable feature because of RawComparator acting on that variable and it will compare the key taken with the other keys in network and If Writable is not there it won't be executed.

By default, IntWritable, LongWritable and Text have a RawComparator which can execute this comparable phase for them. Then, will RawComparator help the custom Writable? The answer is no. So, we need to have WritableComparable.

WritableComparable can be defined as a sub interface of Writable, which has the feature of Comparable too. If we have created our custom type writable, then

why do we need WritableComparable?

We need to make our custom type, comparable if we want to compare this type with the other.

we want to make our custom type as a key, then we should definitely make our key type as WritableComparable rather than simply Writable. This enables the custom type to be compared with other types and it is also sorted accordingly. Otherwise, the keys won't be compared with each other and they are just passed through the network.

What happens if WritableComparable is not present?

If we have made our custom type Writable rather than WritableComparable our data won't be compared with other data types. There is no compulsion that our custom types need to be

WritableComparable until unless if it is a key. Because values don't need to be compared with each other as keys.

If our custom type is a key then we should have WritableComparable or else the data won't be sorted.

How can WritableComparable be implemented in Hadoop?

The implementation of WritableComparable is similar to Writable but with an additional 'CompareTo' method inside it.

public interface WritableComparable extends Writable, Comparable

{

  void readFields(DataInput in);

  void write(DataOutput out);

  int compareTo(WritableComparable o)

}

How to make our custom type, WritableComparable?

We can make custom type a WritableComparable by following the method below:

public class add implements WritableComparable{

public int a;

public int b;

public add(){

this.a=a;

this.b=b;

}

public void write(DataOutput out) throws IOException {
  out.writeint(a);

  out.writeint(b);

 }

```
public void readFields(DataInput in) throws IOException {

    a = in.readint();

    b = in.readint();

  }

public int CompareTo(add c){
 int presentValue=this.value;
 int CompareValue=c.value;
 return (presentValue < CompareValue ? -1 : (presentValue==CompareValue ? 0 : 1));
 }

public int hashCode() {
    return Integer.IntToIntBits(a)^ Integer.IntToIntBits(b);
  }

}
```

These read fields and write make the comparison of data faster in the network.

With the use of these Writable and WritableComparables in Hadoop, we can make our serialized custom type with less difficulty. This gives the ease for developers to make their custom types based on their requirement.

Writable Classes – Hadoop Data Types

Hadoop provides classes that wrap the Java primitive types and implement the *WritableComparable* and *Writable* Interfaces. They are provided in the org.apache.hadoop.io package.

All the Writable wrapper classes have a get() and a set() method for retrieving and storing the wrapped value.

Primitive Writable Classes

These are Writable Wrappers for Java primitive data types and they hold a single primitive value that can be set either at construction or via a setter method.

All these primitive writable wrappers have get() and set() methods to read or write the wrapped value. Below is the list of primitive writable data types available in Hadoop.

- BooleanWritable
- ByteWritable
- IntWritable
- VIntWritable
- FloatWritable
- LongWritable
- VLongWritable
- DoubleWritable

In the above list VIntWritable and VLongWritable are used for variable length Integer types and variable length long types respectively.

Serialized sizes of the above primitive writable data types are same as the size of actual java data type. So, the size of IntWritable is 4 bytes and LongWritable is 8 bytes.

Array Writable Classes

Hadoop provided two types of array writable classes, one for *single-dimensional* and another for *two-dimensional* arrays. But theelements of these arrays must be other writable objects like IntWritable or LongWritable only but not the java native data types like int or float.

- ArrayWritable
- TwoDArrayWritable

Map Writable Classes

Hadoop provided below MapWritable data types which implement java.util.Map interface

o   AbstractMapWritable – This is abstract or base class for other MapWritable classes.

- o MapWritable – This is a general purpose map mapping Writable keys to Writable values.
- o SortedMapWritable – This is a specialization of the MapWritable class that also implements the SortedMap interface.

Other Writable Classes

- **NullWritable**

NullWritable is a special type of Writable representing a null value. No bytes are read or written when a data type is specified as NullWritable. So, in Mapreduce, a key or a value can be declared as a NullWritable when we don't need to use that field.

- **ObjectWritable**

This is a general-purpose generic object wrapper which can store any objects like Java primitives, String, Enum, Writable, null, or arrays.

- **Text**

Text can be used as the Writable equivalent of java.lang.String and It's max size is 2 GB. Unlike java's String data type, Text is mutable in Hadoop.

- **BytesWritable**

BytesWritable is a wrapper for an array of binary data.

- **GenericWritable**

It is similar to ObjectWritable but supports only a few types. User need to subclass this GenericWritable class and need to specify the types to support.

Example Program to Test Writables

Lets write a WritablesTest.java program to test most of the data types mentioned above in this post with get(), set(), getBytes(), getLength(), put(), containsKey(), keySet() methods.

WritablesTest.java

```java
import org.apache.hadoop.io.* ;
import java.util.* ;
 public class WritablesTest
{
  public static class TextArrayWritable extends ArrayWritable
  {
   public TextArrayWritable()
   {
    super(Text.class) ;
   }
  }

  public static class IntArrayWritable extends ArrayWritable
  {
   public IntArrayWritable()
   {
    super(IntWritable.class) ;
   }
  }

  public static void main(String[] args)
  {

  IntWritable i1 = new IntWritable(2) ;
  IntWritable i2 = new IntWritable() ;
  i2.set(5);

  IntWritable i3 = new IntWritable();
  i3.set(i2.get());

  System.out.printf("Int Writables Test I1:%d , I2:%d , I3:%d", i1.get(), i2.get(), i3.get()) ;

  BooleanWritable bool1 = new BooleanWritable() ;
  bool1.set(true);

  ByteWritable byte1 = new ByteWritable( (byte)7) ;

  System.out.printf("\n Boolean Value:%s Byte Value:%d", bool1.get(), byte1.get()) ;

  Text t = new Text("hadoop");
  Text t2 = new Text();
  t2.set("pig");

  System.out.printf("\n t: %s, t.legth: %d, t2: %s, t2.length: %d \n", t.toString(), t.getLength(),
 t2.getBytes(), t2.getBytes().length);
```

```
ArrayWritable a = new ArrayWritable(IntWritable.class) ;
a.set( new IntWritable[]{ new IntWritable(10), new IntWritable(20), new IntWritable(30)}) ;

ArrayWritable b = new ArrayWritable(Text.class) ;

b.set( new Text[]{ new Text("Hello"), new Text("Writables"), new Text("World !!!")}) ;

for (IntWritable i: (IntWritable[])a.get())
System.out.println(i) ;

for (Text i: (Text[])b.get())
System.out.println(i) ;

IntArrayWritable ia = new IntArrayWritable() ;
ia.set( new IntWritable[]{ new IntWritable(100), new IntWritable(300), new IntWritable(500)}) ;

IntWritable[] ivalues = (IntWritable[])ia.get() ;

for (IntWritable i : ivalues)
System.out.println(i);

MapWritable m = new MapWritable() ;
IntWritable key1 = new IntWritable(1) ;
NullWritable value1 = NullWritable.get() ;

m.put(key1, value1) ;
m.put(new VIntWritable(2), new LongWritable(163));
m.put(new VIntWritable(3), new Text("Mapreduce"));

System.out.println(m.containsKey(key1)) ;
System.out.println(m.get(new VIntWritable(3))) ;

m.put(new LongWritable(1000000000), key1) ;
Set<Writable> keys = m.keySet() ;

for(Writable w: keys)
 System.out.println(m.get(w)) ;
 }
}
```

```
hadoop1@ubuntu-1:~/project$ javac -classpath $CLASSPATH WritablesTest.java
hadoop1@ubuntu-1:~/project$ java WritablesTest
Int Writables Test I1:2 , I2:5 , I3:5
Boolean Value:true Byte Value:7
t: hadoop, t.legth: 6, t2: [B@1a93a7ca, t2.length: 3
-------Array variables-------
10
20
30
Hello
Writables
World !!!
100
300
500
-------Map Variables-------
true
Mapreduce
(null)
163
Mapreduce
1
hadoop1@ubuntu-1:~/project$
```

Implementing a Custom Writable:

IMPLEMENTING RAWCOMPARATOR WILL SPEED UP YOUR HADOOP MAP/REDUCE (MR) JOBS

Implementing the org.apache.hadoop.io.RawComparator interface will definitely help speed up your Map/Reduce (MR) Jobs. As you may recall, a MR Job is composed of receiving and sending key-value pairs. The process looks like the following.

(K1,V1) –> Map –> (K2,V2)

(K2,List[V2]) –> Reduce –> (K3,V3)

The key-value pairs (K2,V2) are called the intermediary key-value pairs. They are passed from the mapper to the reducer. Before these intermediary key-value pairs reach the reducer, a shuffle and sort step is performed. The shuffle is the assignment of the intermediary keys (K2) to reducers and the sort is the sorting of these keys. In this blog, by implementing the RawComparator to compare the intermediary keys, this extra effort will greatly improve sorting. Sorting is improved because the RawComparator will compare the keys by byte. If we did not use RawComparator, the intermediary keys would have to be completely deserialized to perform a comparison.

BACKGROUND

Two ways you may compare your keys is by implementing the org.apache.hadoop.io.WritableComparable interface or by implementing the RawComparator interface. In the former approach, you will compare (deserialized) objects, but in the latter approach, you will compare the keys using their corresponding raw bytes.

The empirical test to demonstrate the advantage of RawComparator over WritableComparable. Let's say we are processing a file that has a list of pairs of indexes {i,j}. These pairs of indexes could refer to the i-th and j-th matrix element. The input data (file) will look something like the following.

1, 2

3, 4

5, 6

...

...

...

0, 0

What we want to do is simply count the occurrences of the {i,j} pair of indexes. Our MR Job will look like the following.

(LongWritable,Text) –> Map –> ({i,j},IntWritable)

({i,j},List[IntWritable]) –> Reduce –> ({i,j},IntWritable)

METHOD

The first thing we have to do is model our intermediary key K2={i,j}. Below is a snippet of the IndexPair. As you can see, it implements WritableComparable. Also, we are sorting the keys ascendingly by the i-th and then j-th indexes.

public class IndexPair implements WritableComparable<IndexPair> {

  private IntWritable i;

  private IntWritable j;

```
public IndexPair(int i, int j) {

  this.i = new IntWritable(i);

  this.j = new IntWritable(j);

}

public int compareTo(IndexPair o) {

  int cmp = i.compareTo(o.i);

  if(0 != cmp)

    return cmp;

  return j.compareTo(o.j);

}

//....

}
```

Below is a snippet of the RawComparator. As you notice, it does not directly implement RawComparator. Rather, it extends WritableComparator (which implements RawComparator). We could have directly implemented RawComparator, but by extending WritableComparator, depending on the complexity of our intermediary key, we may use some of the utility methods of WritableComparator.

```
public class IndexPairComparator extends WritableComparator {

  protected IndexPairComparator() {

    super(IndexPair.class);

  }

  @Override

  public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {

    int i1 = readInt(b1, s1);

    int i2 = readInt(b2, s2);

    int comp = (i1 < i2) ? -1 : (i1 == i2) ? 0 : 1;
```

```
    if(0 != comp)

        return comp;

    int j1 = readInt(b1, s1+4);

    int j2 = readInt(b2, s2+4);

    comp = (j1 < j2) ? -1 : (j1 == j2) ? 0 : 1;


    return comp;

  }

}
```

As you can see the above code, for the two objects we are comparing, there are two corresponding byte arrays (b1 and b2), the starting positions of the objects in the byte arrays, and the length of the bytes they occupy. Please note that the byte arrays themselves represent other things and not only the objects we are comparing. That is why the starting position and length are also passed in as arguments. Since we want to sort ascendingly by i then j, we first compare the bytes representing the i-th indexes and if they are equal, then we compare the j-th indexes. You can also see that we use the util method, readInt(byte[], start), inherited from WritableComparator. This method simply converts the 4 consecutive bytes beginning at start into a primitive int (the primitive int in Java is 4 bytes). If the i-th indexes are equal, then we shift the starting point by 4, read in the j-th indexes and then compare them.

A snippet of the mapper is shown below.

```
public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {

    String[] tokens = value.toString().split(",");

    int i = Integer.parseInt(tokens[0].trim());

    int j = Integer.parseInt(tokens[1].trim());


    IndexPair indexPair = new IndexPair(i, j);

    context.write(indexPair, ONE);

}
```

A snippet of the reducer is shown below.

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

    int sum = 0;

    for(IntWritable value : values) {

        sum += value.get();

    }


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

}
```

The snippet of code below shows how I wired up the MR Job that does NOT use raw byte comparison.

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

    Configuration conf = getConf();

    Job job = new Job(conf, "raw comparator example");

      job.setJarByClass(RcJob1.class);

    job.setMapOutputKeyClass(IndexPair.class);

    job.setMapOutputValueClass(IntWritable.class);

    job.setOutputKeyClass(IndexPair.class);

    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(RcMapper.class);

    job.setReducerClass(RcReducer.class);

    job.waitForCompletion(true);

    return 0;

}
```

The snippet of code below shows how I wired up the MR Job using the raw byte comparator.

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

    Configuration conf = getConf();

    Job job = new Job(conf, "raw comparator example");


    job.setJarByClass(RcJob1.class);

    job.setSortComparatorClass(IndexPairComparator.class);


    job.setMapOutputKeyClass(IndexPair.class);

    job.setMapOutputValueClass(IntWritable.class);

    job.setOutputKeyClass(IndexPair.class);

    job.setOutputValueClass(IntWritable.class);


    job.setMapperClass(RcMapper.class);

    job.setReducerClass(RcReducer.class);


    job.waitForCompletion(true);
    return 0;

}
```

As you can see, the only difference is that in the MR Job using the raw comparator, we explicitly set its sort comparator class.

RESULTS

I ran the MR Jobs (without and with raw byte comparisons) 10 times on a dataset of 4 million rows of {i,j} pairs. The runs were against Hadoop v0.20 in standalone mode on Cygwin. The average running time for the MR Job without raw byte comparison is 60.6 seconds, and the average running time for the job with raw byte comparison is 31.1 seconds. A two-tail paired t-test showed $p < 0.001$, meaning, there is a statistically significant difference between the two implementations in terms of empirical running time.

I then ran each implementation on datasets of increasing record sizes from 1, 2, …, and 10 million records. At 10 million records, without using raw byte comparison took 127 seconds (over 2 minutes) to complete, while using raw byte comparison took 75 seconds (1 minute and 15 seconds) to complete. Below is a line graph.

Comparing empirical running times of a Map/Reduce job without (no-rc) and with (rc) raw byte comparison

www.Jntufastupdates.com
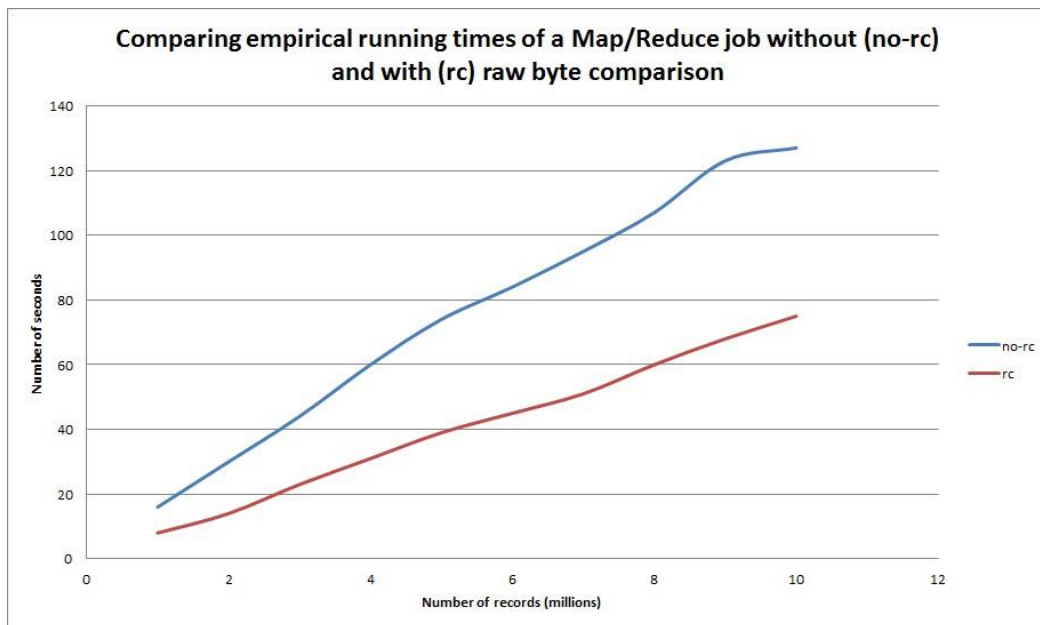
Custom comparators.

Frequently, objects in one Tuple are compared to objects in a second Tuple. This is especially true during the sort phase of GroupBy and CoGroup in Cascading Hadoop mode.

By default, Hadoop and Cascading use the native Object methods equals() and hashCode() to compare two values and get a consistent hash code for a given value, respectively.

To override this default behavior, you can create a custom java.util.Comparator class to perform comparisons on a given field in a Tuple. For instance, to secondary-sort a collection of custom Person objects in a GroupBy, use the Fields.setComparator() method to designate the custom Comparator to the Fields instance that specifies the sort fields.

Alternatively, you can set a default Comparator to be used by a Flow, or used locally on a given Pipe instance. There are two ways to do this. Call FlowProps.setDefaultTupleElementComparator() on a Properties instance, or use the property key cascading.flow.tuple.element.comparator.

If the hash code must also be customized, the custom Comparator can implement the interface cascading.tuple.Hasher

```
public class CustomTextComparator extends

{

private Collator collator;

public CustomTextComparator()
 super(Text.class);

 final Locale locale = new Locale("pl");

 collator = Collator.getInstance(locale);

 }

public int compare(WritableComparable a, WritableComparable b) {

 synchronized (collator) {

 return collator.compare(((Text) a).toString(), ((Text) b).toString());

 }

 }

 }
```

Applying Structure to Hadoop Data with Hive: Saying Hello to Hive, Seeing How the Hive is Put Together, Getting Started with Apache Hive, Examining the Hive Clients, Working with Hive Data Types, Creating and Managing Databases and Tables, Seeing How the Hive Data Manipulation Language Works, Querying and Analyzing Data.

HIVE Introduction

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

The term 'Big Data' is used for collections of large datasets that include huge volume, high velocity, and a variety of data that is increasing day by day. Using traditional data management systems, it is difficult to process Big Data. Therefore, the Apache Software Foundation introduced a framework called Hadoop to solve Big Data management and processing challenges.

Hadoop

Hadoop is an open-source framework to store and process Big Data in a distributed environment. It contains two modules, one is MapReduce and another is Hadoop Distributed File System (HDFS).

☐ **MapReduce:** It is a parallel programming model for processing large amounts of structured, semi-structured, and unstructured data on large clusters of commodity hardware.

☐ **HDFS:** Hadoop Distributed File System is a part of Hadoop framework, used to store and process the datasets. It provides a fault-tolerant file system to run on commodity hardware.

The Hadoop ecosystem contains different sub-projects (tools) such as Sqoop, Pig, and Hive that are used to help Hadoop modules.

☐ **Sqoop:** It is used to import and export data to and from between HDFS and RDBMS.

☐ **Pig:** It is a procedural language platform used to develop a script for MapReduce operations.

☐ **Hive:** It is a platform used to develop SQL type scripts to do MapReduce operations.

Note: There are various ways to execute MapReduce operations:

☐ The traditional approach using Java MapReduce program for structured, semi- structured, and unstructured data.

☐ The scripting approach for MapReduce to process structured and semi structured data using Pig.

☐ The Hive Query Language (HiveQL or HQL) for MapReduce to process structured data using Hive.

What is Hive

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

Hive is not

☐   A relational database

☐   A design for OnLine Transaction Processing (OLTP)

☐   A language for real-time queries and row-level updates

Features of Hive

➢ It stores schema in a database and processed data into HDFS.
➢   It is designed for OLAP.
➢ It provides SQL type language for querying called HiveQL or HQL.
➢ It is familiar, fast, scalable, and extensible.

Architecture of Hive

The following component diagram depicts the architecture of Hive:
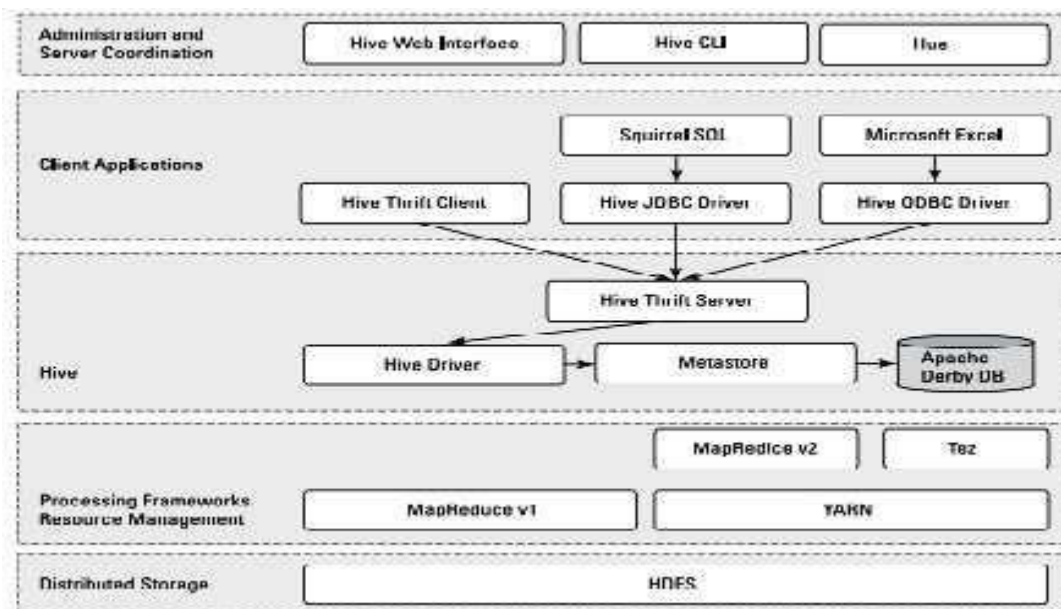
The apache Hive architecture

This   component   diagram   contains   different   units.   The   following table describes each unit:

Unit Name	Operation
User Interface	Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows
Meta Store	Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.
HiveQL Process Engine	HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it.
Execution Engine	The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce
HDFS or HBASE	Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Working of Hive

The following diagram depicts the workflow between Hive and Hadoop.
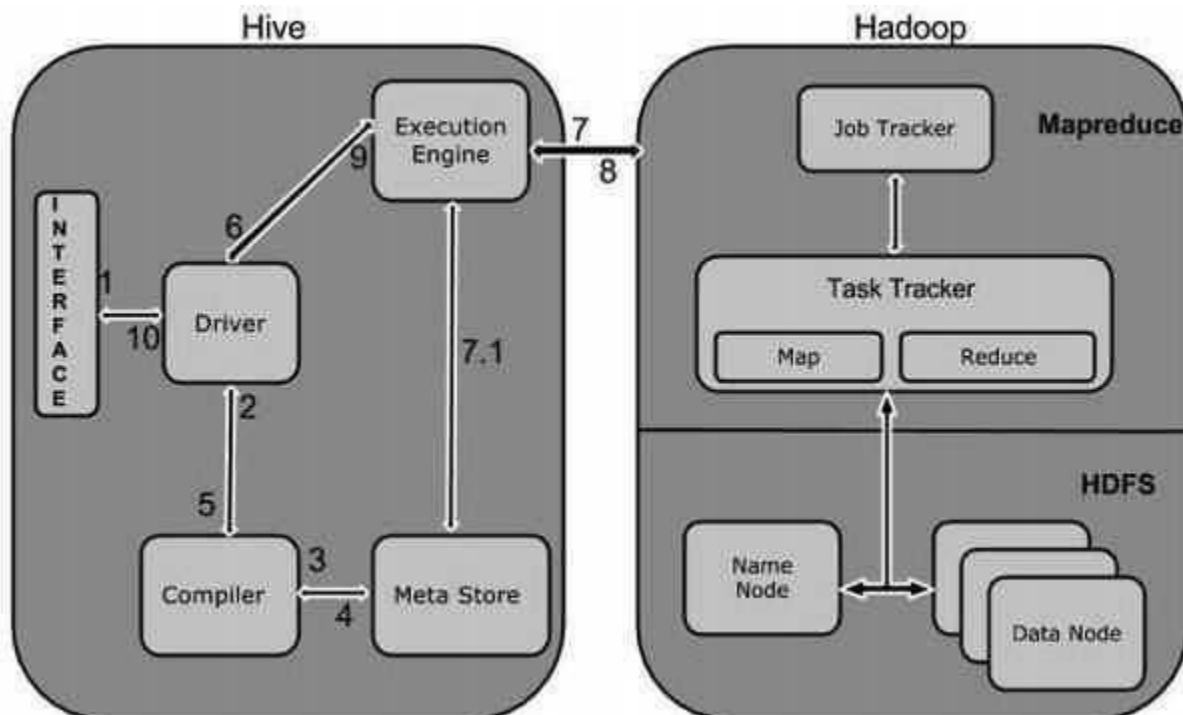
The following table defines how Hive interacts with Hadoop framework:

Step No.	Operation
1	**Execute Query** The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute.
2	**Get Plan** The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query.
3	**Get Metadata** The compiler sends metadata request to Metastore (any database).
4	**Send Metadata** Metastore sends metadata as a response to the compiler.
5	**Send Plan** The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete.
6	**Execute Plan** The driver sends the execute plan to the execution engine.
7	**Execute Job** Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Here, the query executes MapReduce job.
7.1	**Metadata Ops** Meanwhile in execution, the execution engine can execute metadata operations with Metastore.
8	**Fetch Result** The execution engine receives the results from Data nodes.
9	**Send Results** The execution engine sends those resultant values to the driver.
10	**Send Results** The driver sends the results to Hive Interfaces.

Hive - Data Types

All the data types in Hive are classified into four types, given as follows:

- Column Types
- Literals
- Null Values

- Complex Types

Column Types

Column type are used as column data types of Hive. They are as follows:

Integral Types

Integer type data can be specified using integral data types, INT. When the data range exceeds the range of INT, you need to use BIGINT and if the data range is smaller than the INT, you use SMALLINT. TINYINT is smaller than SMALLINT.

The following table depicts various INT data types:

Type	Postfix	Example
TINYINT	Y	10Y
SMALLINT	S	10S
INT	-	10
BIGINT	L	10L

String Types

String type data types can be specified using single quotes (' ') or double quotes (" "). It contains two data types: VARCHAR and CHAR. Hive follows C-types escape characters.

The following table depicts various CHAR data types:

Data Type	Length
VARCHAR	1 to 65355
CHAR	255

Timestamp

It supports traditional UNIX timestamp with optional nanosecond precision. It supports java.sql.Timestamp format "YYYY-MM-DD HH:MM:SS.fffffffff" and format "yyyy-mm-dd hh:mm:ss.fffffffff".

Dates

DATE values are described in year/month/day format in the form {{YYYY- MM-DD}}.

Decimals

The DECIMAL type in Hive is as same as Big Decimal format of Java. It is used for representing immutable arbitrary precision. The syntax and example is as follows:

```
DECIMAL(precision, scale)

decimal(10,0)
```

Union Types

Union is a collection of heterogeneous data types. You can create an instance using **create union**. The syntax and example is as follows:

```
UNIONTYPE<int, double, array<string>, struct<a:int,b:string>>

{0:1}

{1:2.0}

{2:["three","four"]}

{3:{"a":5,"b":"five"}}

{2:["six","seven"]}
{3:{"a":8,"b":"eight"}}
{0:9}
{1:10.0}
```

Literals

The following literals are used in Hive:

Floating Point Types

Floating point types are nothing but numbers with decimal points. Generally, this type of data is composed of DOUBLE data type.

Decimal Type

Decimal type data is nothing but floating point value with higher range than DOUBLE data type. The range of decimal type is approximately -10^{-308} to 10^{308}.

Null Value

Missing values are represented by the special value NULL.

Complex Types

The Hive complex data types are as follows:

Arrays

Arrays in Hive are used the same way they are used in Java.

```
Syntax: ARRAY<data_type>
```

Maps

Maps in Hive are similar to Java Maps.

```
Syntax: MAP<primitive_type, data_type>
```

Structs

Structs in Hive is similar to using complex data with comment.

```
Syntax: STRUCT<col_name : data_type [COMMENT col_comment], ...>
```

Hive - Create Database

☐   Hive is a database technology that can define databases and tables to analyze structured data. The theme for structured data analysis is to store the data in a tabular manner, and pass queries to analyze it. This chapter explains how to create Hive database. Hive contains a default database named **default**.

Create Database Statement

☐   Create Database is a statement used to create a database in Hive. A database in Hive is a **namespace** or a collection of tables. The **syntax** for this statement is as follows:

```
☐    CREATE DATABASE|SCHEMA [IF NOT EXISTS] <database name>
```

☐Here, IF NOT EXISTS is an optional clause, which notifies the user that a database with the same name already exists. We can use SCHEMA in place of DATABASE in this command. The following query is executed to create a database named **userdb**:

    hive> CREATE DATABASE [IF NOT EXISTS] userdb;

or

    hive> CREATE SCHEMA userdb;

The following query is used to verify a databases list:

    hive> SHOW DATABASES;
    default
    userdb

Drop Database Statement

Drop Database is a statement that drops all the tables and deletes the database. Its syntax is as follows:

DROP DATABASE StatementDROP (DATABASE|SCHEMA) [IF EXISTS] database_name

[RESTRICT|CASCADE];

The following queries are used to drop a database. Let us assume that the database name is **userdb**.

hive> DROP DATABASE IF EXISTS userdb;

The following query drops the database using **CASCADE**. It means dropping respective tables before dropping the database.

hive> DROP DATABASE IF EXISTS userdb CASCADE;

The following query drops the database using **SCHEMA**.

hive> DROP SCHEMA userdb;

create Table Statement

Create Table is a statement used to create a table in Hive. The syntax and example are as follows:

Syntax

CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.] table_name

[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[ROW FORMAT row_format] [STORED AS file_format]

Example

Let us assume you need to create a table named **employee** using **CREATE TABLE** statement. The following table lists the fields and their data types in employee table:

Sr.No	Field Name	Data Type
1	Eid	int
2	Name	String
3	Salary	Float
4	Designation	string

The following data is a Comment, Row formatted fields such as Field terminator, Lines terminator, and Stored File type.

```
COMMENT ʈEmployee detailsŨ FIELDS TERMINATED BY ʈ\tŨ LINES TERMINATED BY ʈ\nŨ STORED IN TEXT FILE
```

The following query creates a table named **employee** using the above data.

```
hive> CREATE TABLE IF NOT EXISTS employee ( eid int, name String, salary String, destination String)
COMMENT ʈEmployee detailsŨ
ROW FORMAT DELIMITED FIELDS TERMINATED BY ʈ\tŨ LINES TERMINATED BY ʈ\nŨ STORED AS TEXTFILE;
```

If you add the option IF NOT EXISTS, Hive ignores the statement in case the table already exists.

On successful creation of table, you get to see the following response:

```
OK
Time taken: 5.905 seconds
hive>
```

Load Data Statement

Generally, after creating a table in SQL, we can insert data using the Insert statement. But in Hive, we can insert data using the LOAD DATA statement.

While inserting data into Hive, it is better to use LOAD DATA to store bulk records. There are two ways to load data: one is from local file system and second is from Hadoop file system.

Syntax

The   syntax   for   load   data   is   as follows:

```
LOAD  DATA  [LOCAL]  INPATH  'filepath'  [OVERWRITE]  INTO  TABLE  tablename
[PARTITION (partcol1=val1, partcol2=val2 ...)]
```

- ☐   LOCAL is identifier to specify the local path. It is optional.

- ☐   OVERWRITE is optional to overwrite the data in the table.

- ☐   PARTITION is optional.

Example

We   will   insert   the   following   data   into   the   table.   It   is   a   text
file named**sample.txt** in **/home/user** directory.

```
1201    Gopal            45000        Technical  manager
1202    Manisha          45000        Proof  reader
1203    Masthanvali  40000           Technical  writer
1204    Kiran            40000        Hr   Admin
1205    Kranthi          30000        Op   Admin
```

The following query loads the given text into the table.

```
hive> LOAD  DATA  LOCAL  INPATH
'/home/user/sample.txt' OVERWRITE  INTO  TABLE  employee;
```

On successful download, you get to see the following response:

```
OK
Time  taken: 15.905  seconds
hive>
```

Alter Table Statement

It is used to alter a table in Hive.

Syntax

The statement takes any of the following syntaxes based on what attributes we wish to
modify in a table.

```
ALTER  TABLE  name  RENAME  TO  new_name
ALTER  TABLE  name  ADD COLUMNS  (col_spec[, col_spec ...])
ALTER  TABLE  name  DROP  [COLUMN]  column_name
ALTER  TABLE  name  CHANGE  column_name  new_name  new_type
ALTER  TABLE  name  REPLACE  COLUMNS  (col_spec[, col_spec ...])
```

Rename To…Statement

The following query renames the table from **employee** to **emp**.

```
hive>  ALTER  TABLE  employee  RENAME  TO  emp;
```

Change Statement

The following table contains the fields of **employee** table and it shows the fields to be changed (in bold).

Field Name	Convert from Data Type	Change Field Name	Convert to Data Type
eid	int	eid	int
name	String	**ename**	String
salary	**Float**	salary	**Double**
designation	String	designation	String

The following queries rename the column name and column data type using the above data:

```
hive> ALTER TABLE employee CHANGE name ename String;
hive> ALTER TABLE employee CHANGE salary salary Double;
```

Add Columns Statement

The following query adds a column named dept to the employee table.

```
hive> ALTER TABLE employee ADD COLUMNS (
dept STRING COMMENT 'Department name');
```

Replace Statement

The following query deletes all the columns from the **employee** table and replaces it with **emp** and **name** columns:

```
hive> ALTER TABLE employee REPLACE COLUMNS (
eid INT empid Int,
ename STRING name String);
```

Drop Table Statement

Hive Metastore, it removes the table/column data and their metadata. It can be a normal table (stored in Metastore) or an external table (stored in local file system); Hive treats both in the same manner, irrespective of their types.

The syntax is as follows:

```
DROP TABLE [IF EXISTS] table_name;
```

The following query drops a table named **employee**:

```
hive> DROP  TABLE  IF  EXISTS  employee;
```

                                                              see   the   following

On   successful   execution   of   the   query,   you   get
to response:

```
OK
Time taken: 5.3 seconds

hive>
```

The following query is used to verify the list of tables:

```
hive> SHOW TABLES;
emp ok
Time taken: 2.1 seconds

hive>
```

Operators in HIVE:

There are four types of operators in Hive:

➢ Relational Operators

➢ Arithmetic Operators

➢ Logical Operators

➢ Complex Operators

Relational Operators: These operators are used to compare two operands. The following table describes the relational operators available in Hive:

Operator	Operand	Description
A = B	all primitive types	TRUE if expression A is equivalent to expression B otherwise FALSE.
A != B	all primitive types	TRUE if expression A is not equivalent to expression B otherwise FALSE.
A < B	all primitive types	TRUE if expression A is less than expression B otherwise FALSE.
A <= B	all primitive types	TRUE if expression A is less than or equal to expression B otherwise FALSE.
A > B	all primitive types	TRUE if expression A is greater than expression B otherwise FALSE.
A >= B	all primitive types	TRUE if expression A is greater than or equal to expression B otherwise FALSE.

A IS NULL	all types	TRUE if expression A evaluates to NULL otherwise FALSE.
A IS NOT NULL	all types	FALSE if expression A evaluates to NULL otherwise TRUE.
A LIKE B	Strings	TRUE if string pattern A matches to B otherwise FALSE.
A RLIKE B	Strings	NULL if A or B is NULL, TRUE if any substring of A matches the Java regular expression B , otherwise FALSE.
A REGEXP B	Strings	Same as RLIKE.

Example

Let us assume the **employee** table is composed of fields named Id, Name, Salary, Designation, and Dept as shown below. Generate a query to retrieve the employee details whose Id is 1205.

```
+-----+--------------+--------+-------------------------+------+
| Id  | Name         | Salary | Designation             | Dept |
+-----+--------------+--------+-------------------------+------+
|1201 | Gopal        | 45000  | Technical  manager      | TP    |
|1202 | Manisha      | 45000  | Proofreader             | PR    |
|1203 | Masthanvali  | 40000  | Technical  writer       | TP    |
|1204 | Krian        | 40000  | Hr Admin                | HR    |
|1205 | Kranthi      | 30000  | Op Admin                | Admin|
+-----+--------------+--------+-------------------------+------+
```

The following query is executed to retrieve the employee details using the above table:

```
hive> SELECT * FROM employee WHERE Id=1205;
```

On successful execution of query, you get to see the following response:

```
+-----+-----------+-----------+-------------------------------+
| ID  | Name      | Salary    | Designation                   | Dept   |
+-----+--------------+-------+-------------------------------+
|1205 | Kranthi   | 30000     | Op Admin                      | Admin |
+-----+----------+-----------+-------------------------------+
```

The following query is executed to retrieve the employee details whose salary is more than or equal to Rs 40000.

```
hive> SELECT * FROM employee WHERE Salary>=40000;
```

On successful execution of query, you get to see the following response:

```
+-----+-----------+--------+--------------------------+------+
| ID  | Name      | Salary | Designation              | Dept |
+-----+-----------+--------+--------------------------+------+
|120  | Gopal     | 45000  | Technical  manager       | TP   |
|120  | Manisha   | 45000  | Proofreader              | PR   |
|1203 | Masthanvali| 40000 | Technical  writer        | TP   |
|1204 | Krian     | 40000  | Hr Admin                 | HR   |
+-----+-----------+--------+--------------------------+------+
```

Arithmetic Operators

These operators support various common arithmetic operations on the operands. All of them return number types. The following table describes the arithmetic operators available in Hive:

Operators	Operand	Description
A + B	all number types	Gives the result of adding A and B.
A - B	all number types	Gives the result of subtracting B from A.
A * B	all number types	Gives the result of multiplying A and B.
A / B	all number types	Gives the result of dividing B from A.
A % B	all number types	Gives the reminder resulting from dividing A by B.
A & B	all number types	Gives the result of bitwise AND of A and B.
A | B	all number types	Gives the result of bitwise OR of A and B.
A ^ B	all number types	Gives the result of bitwise XOR of A and B.
~A	all number types	Gives the result of bitwise NOT of A.

Example

The following query adds two numbers, 20 and 30.

```
hive> SELECT 20+30 ADD FROM temp;
```

On successful execution of the query, you get to see the following response:

```
+--------+
|   ADD  |
+--------+
|   50   |
+--------+
```

Logical Operators

The operators are logical expressions. All of them return either TRUE or

FALSE.

Operators	Operands	Description
A AND B	boolean	TRUE if both A and B are TRUE, otherwise FALSE.
A && B	boolean	Same as A AND B.
A OR B	boolean	TRUE if either A or B or both are TRUE, otherwise FALSE.
A || B	boolean	Same as A OR B.
NOT A	boolean	TRUE if A is FALSE, otherwise FALSE.
!A	boolean	Same as NOT A.

Example

The following query is used to retrieve employee details whose Department is TP and Salary is more than Rs 40000.

```
hive> SELECT * FROM employee WHERE Salary>40000 && Dept=TP;
```

On successful execution of the query, you get to see the following response:

```
+------+-------------+------------+------------------+--------+
| ID   | Name        | Salary     | Designation      | Dept   |
+------+-------------+------------+------------------+--------+
|1201  | Gopal       | 45000      | Technical manager | TP    |
+------+-------------+------------+------------------+--------+
```

Complex Operators

These operators provide an expression to access the elements of Complex Types.

Operator	Operand	Description
A[n]	A is an Array and n is an int	It returns the nth element in the array A. The first element has index 0.
M[key]	M is a Map<K, V> and key has type K	It returns the value corresponding to the key in the map.
S.x	S is a struct	It returns the x field of S.

HiveQL - Select-Where

☐ The Hive Query Language (HiveQL) is a query language for Hive to process and analyze structured data in a Metastore. This chapter explains how to use the SELECT statement with WHERE clause.

☐ SELECT statement is used to retrieve the data from a table. WHERE clause works similar to a condition. It filters the data using the condition and gives you a finite result. The built-in operators and functions generate an expression, which fulfils the condition.

Syntax

Given below is the syntax of the SELECT query:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
 FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
 [CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY col_list]]
[LIMIT number];
```

Example

Let us take an example for SELECT…WHERE clause. Assume we have the employee table as given below, with fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details who earn a salary of more than Rs 30000.

```
+------+-------------+------------+------------------+--------+
| ID | Name           | Salary       | Designation        | Dept      |
+------+-------------+------------+------------------+--------+
      Gopal         | 45000        | Technical manager | TP
      Manisha       | 45000        | Proofreader        | PR
|1203 | Masthanvali | 40000        | Technical writer   | TP      |
|1204 | Krian       | 40000        | Hr Admin           | HR      |
|1205 | Kranthi     | 30000        | Op                 | Admin |
+------+-------------+------------+------------------+--------+
```

The following query retrieves the employee details using the above scenario:

```
hive> SELECT * FROM employee WHERE salary>30000;
```

On successful execution of the query, you get to see the following response:

```
+------+-------------+------------+------------------+--------+
| ID   | Name        | Salary     | Designation        | Dept   |
+------+-------------+------------+------------------+--------+
|1201 | Gopal       | 45000      | Technical manager | TP     |
|1202 | Manisha     | 45000      | Proofreader        | PR     |
|1203 | Masthanvali | 40000      | Technical writer   | TP     |
|1204 | Krian       | 40000      | Hr Admin           | HR     |
+------+-------------+------------+------------------+--------+
```

HiveQL - Select-Order By

□ This chapter explains how to use the ORDER BY clause in a SELECT statement. The ORDER BY clause is used to retrieve the details based on one column and sort the result set by ascending or descending order.

Syntax

      ☐   Given below is the syntax of the ORDER BY clause:

SELECT [ALL | DISTINCT] select_expr, select_expr, ... FROM table_reference

[WHERE where_condition] [GROUP BY col_list]

[HAVING having_condition]
[ORDER BY col_list]] [LIMIT
number];

Example

☐   Let us take an example for SELECT...ORDER BY clause. Assume employee table as given below, with the fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details in order by using Department name.

```
+------+-------------+------------+------------------+--------+
| ID   | Name        | Salary     | Designation      | Dept   |
+------+-------------+------------+------------------+--------+
|1201  | Gopal       | 45000      | Technical manager| TP     |
|1202  | Manisha     | 45000      | Proofreader      | PR     |
|1203  | Masthanvali | 40000      | Technical writer | TP     |
|1204  | Krian       | 40000      | Hr Admin         | HR     |
|1205  | Kranthi     | 30000      | Op Admin         | Admin  |
+------+-------------+------------+------------------+--------+
```

☐   The following query retrieves the employee details using the above scenario:

```
hive> SELECT Id, Name, Dept FROM employee ORDER BY DEPT;
```

HiveQL - Select-Group By

☐   This chapter explains the details of GROUP BY clause in a SELECT statement. The GROUP BY clause is used to group all the records in a result set using a particular collection column. It is used to query a group of records.

The syntax of GROUP BY clause is as follows:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...

FROM table_reference

[WHERE where_condition]

[GROUP BY col_list]

[HAVING having_condition]

[ORDER BY col_list]]

[LIMIT number];
```

HiveQL - Select-Joins

JOIN is a clause that is used for combining specific fields from two tables by using values common to each one. It is used to combine records from two or more tables in the database. It is more or less similar to SQL JOIN.

Syntax

join_table:

    table_reference JOIN table_factor [join_condition]

    | table_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table_reference
join_condition
    | table_reference LEFT SEMI JOIN table_reference join_condition

    | table_reference CROSS JOIN table_reference [join_condition]

Example

We will use the following two tables in this chapter. Consider the following table named CUSTOMERS..

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Consider another table ORDERS as follows:

```
+-----+--------------------+-------------+--------+
|OID  | DATE               | CUSTOMER_ID | AMOUNT |
+-----+--------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |        3   | 3000   |
| 100 | 2009-10-08 00:00:00 |        3   | 1500   |

| 101 | 2009-11-20 00:00:00 |        2   | 1560   |
| 103 | 2008-05-20 00:00:00 |        4   | 2060   |
+-----+--------------------+-------------+--------+
```

There are different types of joins given as follows:

☐   JOIN   ☐   LEFT OUTER JOIN   ☐   RIGHT OUTER JOIN
FULL OUTER JOIN
☐

JOIN

JOIN clause is used to combine and retrieve the records from multiple tables. JOIN is same as OUTER JOIN in SQL. A JOIN condition is to be raised using the primary keys and foreign keys of the tables.

The following query executes JOIN on the CUSTOMER and ORDER tables, and retrieves the records:

```
hive> SELECT c.ID, c.NAME, c.AGE,
o.AMOUNT FROM  CUSTOMERS  c  JOIN
ORDERS  o
ON  (c.ID  =  o.CUSTOMER_ID);
```

On successful execution of the query, you get to see the following response:

```
+----+---------+-----+--------+
| ID | NAME    | AGE |  AMOUNT |
+----+---------+-----+--------+
| 3    | kaushik    | 23       |
3000   |
| 3    | kaushik    | 23       |
1500   |
| 2    | Khilan     | 25       |
1560   |
| 4  | Chaitali | 25   | 2060
```

LEFT OUTER JOIN

The HiveQL LEFT OUTER JOIN returns all the rows from the left table, even if there are no matches in the right table. This means, if the ON clause matches 0 (zero) records in the right table, the JOIN still returns a row in the result, but with NULL in each column from the right table. A LEFT JOIN returns all the values from the left table, plus the matched values from the right table, or NULL in case of no matching JOIN predicate.

The following query demonstrates LEFT OUTER JOIN between CUSTOMER and ORDER tables:

```
hive>  SELECT c.ID, c.NAME, o.AMOUNT,
o.DATE FROM  CUSTOMERS  c
LEFT  OUTER  JOIN  ORDERS  o
ON  (c.ID  =  o.CUSTOMER_ID);
```

On successful execution of the query, you get to see the following response:

```
+----+---------+--------+--------------------+
| ID      | NAME          | AMOUNT | DATE
|
+----+---------+--------+--------------------+
| 1      | Ramesh        | NULL        | NULL                    |
| 2      | Khilan        | 1560        | 2009-11-20  00:00:00 |
| 3      | kaushik       | 3000        | 2009-10-08  00:00:00 |
| 3      | kaushik       | 1500        | 2009-10-08  00:00:00 |
| 4      | Chaitali      | 2060        | 2008-05-20  00:00:00 |

  5      | Hardik        | NULL        |    NULL
| 6      | Komal         | NULL        | NULL
| 7      | Muffy         | NULL        | NULL
|
```

RIGHT OUTER JOIN

The HiveQL RIGHT OUTER JOIN returns all the rows from the right table, even if there are no matches in the left table. If the ON clause matches 0 (zero) records in the left table, the JOIN still returns a row in the result, but with NULL in each column from the left table.

A RIGHT JOIN returns all the values from the right table, plus the matched values from the left table, or NULL in case of no matching join predicate.

The following query demonstrates RIGHT OUTER JOIN between the CUSTOMER and ORDER tables.

hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c RIGHT
OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER_ID);
On successful execution of the query, you get to see the following response:

```
+------+----------+--------+--------------------+
| ID   | NAME     | AMOUNT | DATE               |
+------+----------+--------+--------------------+
| 3    | kaushik  | 3000   | 2009-10-08         |
| 3    | kaushik  | 1500   | 2009-10-08         |
| 2    | Khilan   | 1560   | 2009-11-20         |
| 4    | Chaitali | 2060   | 2008-05-20         |
+------+----------+--------+--------------------+
```

FULL OUTER JOIN

The HiveQL FULL OUTER JOIN combines the records of both the left and the right outer tables that fulfil the JOIN condition. The joined table contains either all the records from both the tables, or fills in NULL values for missing matches on either side.

The following query demonstrates FULL OUTER JOIN between CUSTOMER and ORDER tables:

```
hive> SELECT c.ID, c.NAME, o.AMOUNT,
o.DATE FROM CUSTOMERS c
FULL OUTER JOIN ORDERS o
ON (c.ID = o.CUSTOMER_ID);
```

On successful execution of the query, you get to see the following response:

```
+------+----------+--------+--------------------+
| ID   | NAME      | AMOUNT | DATE               |
+------+----------+--------+--------------------+
| 1    | Ramesh    | NULL   | NULL               |
| 2    | Khilan    | 1560   | 2009-11-20  00:00:00 |
| 3    | kaushik   | 3000   | 2009-10-08  00:00:00 |
| 3    | kaushik   | 1500   | 2009-10-08  00:00:00 |
| 4    | Chaitali | 2060   | 2008-05-20  00:00:00 |
| 5    | Hardik    | NULL   | NULL               |
| 6    | Komal     | NULL   | NULL               |
| 7    | Muffy     | NULL   | NULL               |
| 3    | kaushik   | 3000   | 2009-10-08  00:00:00 |
| 3    | kaushik   | 1500   | 2009-10-08  00:00:00 |
| 2    | Khilan    | 1560   | 2009-11-20  00:00:00 |
| 4    | Chaitali | 2060   | 2008-05-20  00:00:00 |
```

IMP Questions

1. Explain Hive architecture in detail with neat diagram?

2. How the WorkFlow will happen Between Hive and Hadoop?

3. What is the difference between Hive and ApachePig?

4. Write the Hive Script for the following?
   - create database
   - select statement for
   - each statement order
   - by statement
   - alter table statement
   - Drop table statement

Unit 5:

Pig: Hadoop Programming Made Easier Admiring the Pig Architecture, Going with
the Pig Latin Application Flow, Working through the ABCs of Pig Latin,
Evaluating Local and Distributed Modes of Running Pig Scripts, Checking out
the Pig Script Interfaces, Scripting with Pig Latin

Pig:

Hadoop Programming Made Easier Admiring the Pig Architecture

Apache Pig is an abstraction over MapReduce. It is a tool/platform which is
used to analyze larger sets of data representing them as data flows. Pig is
generally used with **Hadoop**; we can perform all the data manipulation
operations in Hadoop using Pig.

To write data analysis programs, Pig provides a high-level language known as
Pig Latin. This language provides various operators using which programmers
can develop their own functions for reading, writing, and processing data.

To analyze data using **Apache Pig**, programmers need to write scripts
using Pig Latin language. All these scripts are internally converted to Map
and Reduce tasks. Apache Pig has a component known as **Pig Engine** that
accepts the Pig Latin scripts as input and converts those scripts
into MapReduce jobs.

Why Do We Need Apache Pig?

Programmers who are not so good at Java normally used to struggle
working with Hadoop, especially while performing any MapReduce tasks.
Apache Pig is a boon for all such programmers.

- Using **Pig Latin**, programmers can perform MapReduce tasks easily
  without having to type complex codes in Java.

- Apache Pig uses **multi-query approach**, thereby reducing the length of codes.
  For example, an operation that would require you to type 200 lines of code
  (LoC) in Java can be easily done by typing as less as just 10 LoC in Apache Pig.
  Ultimately Apache Pig reduces the development time by almost 16 times.

- Pig Latin is **SQL-like language** and it is easy to learn Apache Pig when you are
  familiar with SQL.

- Apache Pig provides many built-in operators to support data operations
  like joins, filters, ordering, etc. In addition, it also provides nested data types
  like tuples, bags, and maps that are missing from MapReduce.

Features of Pig

Apache Pig comes with the following features −

- **Rich set of operators** − It provides many operators to perform operations like join, sort, filer, etc.

- **Ease of programming** − Pig Latin is similar to SQL and it is easy to write a Pig script if you are good at SQL.

- **Optimization opportunities** − The tasks in Apache Pig optimize their execution automatically, so the programmers need to focus only on semantics of the language.

- **Extensibility** − Using the existing operators, users can develop their own functions to read, process, and write data.

- **UDF's** − Pig provides the facility to create **User-defined Functions** in other programming languages such as Java and invoke or embed them in Pig Scripts.

- **Handles all kinds of data** − Apache Pig analyzes all kinds of data, both structured as well as unstructured. It stores the results in HDFS.

Apache Pig Vs MapReduce

Listed below are the major differences between Apache Pig and MapReduce.

Apache Pig	MapReduce
Apache Pig is a data flow language.	MapReduce is a data processing paradigm.
It is a high level language.	MapReduce is low level and rigid.
Performing a Join operation in Apache Pig is	It is quite difficult in MapReduce to perform a Join operation between
pretty simple.	datasets.
Any novice programmer with a basic knowledge of SQL can work conveniently with Apache Pig.	Exposure to Java is must to work with MapReduce.
Apache Pig uses multi-query approach, thereby reducing the length of the codes to a great extent.	MapReduce will require almost 20 times more the number of lines to perform the same task.

There is no need for compilation. On execution, every Apache Pig operator is converted internally into a MapReduce job.	MapReduce jobs have a long compilation process.

Apache Pig Vs SQL

Listed below are the major differences between Apache Pig and SQL.

Pig	SQL
Pig Latin is a **procedural** language.	SQL is a **declarative** language.
In Apache Pig, **schema** is optional. We can store data without designing a schema (values are stored as $01, $02 etc.)	Schema is mandatory in SQL.
The data model in Apache Pig is **nested relational**.	The data model used in SQL **is flat relational**.
Apache Pig provides limited opportunity for **Query optimization**.	There is more opportunity for query optimization in SQL.

In addition to above differences, Apache Pig Latin −

☐   Allows splits in the pipeline.

☐   Allows developers to store data anywhere in the pipeline. ☐   Declares execution plans.

☐   Provides operators to perform ETL (Extract, Transform, and Load) functions.

Apache Pig - Architecture

The language used to analyze data in Hadoop using Pig is known as **Pig Latin**. It is a high level data processing language which provides a rich set of data types and operators to perform various operations on the data.

To perform a particular task Programmers using Pig, programmers need to write a Pig script using the Pig Latin language, and execute them using any of the execution mechanisms (Grunt Shell, UDFs, Embedded). After execution, these scripts will go through a series of transformations applied by the Pig Framework, to produce the desired output.

Internally, Apache Pig converts these scripts into a series of MapReduce jobs, and thus, it makes the programmer's job easy. The architecture of Apache Pig is shown below.
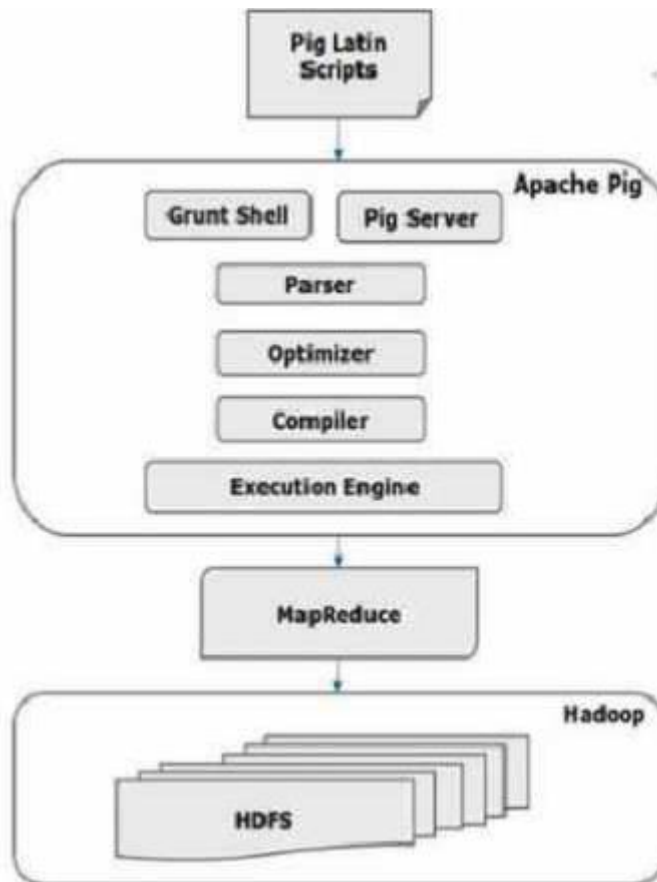

Admiring the pig architecture

Pig is made up of two components:

The language itself: As proof that programmers have a sense of humor, the Programming language for Pig is known as Pig Latin, a high-level language that allows you to write data processing and analysis programs.

The Pig Latin compiler: The Pig Latin compiler converts the Pig Latin code into executable code. The executable code is either in the form of MapReduce jobs or it can spawn a process where a virtual Hadoop instance is created to run the Pig node on a single node.

The sequence of MapReduce programs enables Pig programs to do data processing and analysis in parallel, leveraging Hadoop MapReduce and HDFS. Running the Pig job in the virtual Hadoop instance is a useful strategy for testing your Pig scripts.
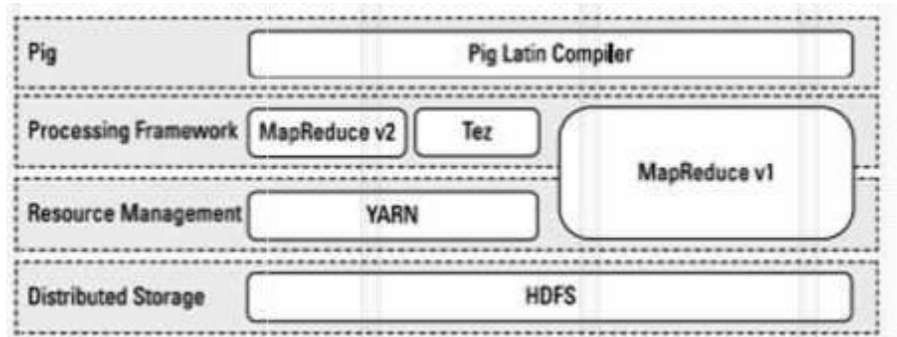
Pig relates to the hadoop ecosystem

Pig programs can run on MapReduce v1 or MapReduce v2 without any code changes, regardless of what mode your cluster is running. However, Pig scripts can also run using the Tez API instead. Apache Tez provides a more efficient execution framework than MapReduce. YARN enables application frameworks other than MapReduce (like Tez) to run on Hadoop. Hive can also run against the Tez framework.

Apache Pig Components

As shown in the figure, there are various components in the Apache Pig framework. Let us take a look at the major components.

Parser

Initially the Pig Scripts are handled by the Parser. It checks the syntax of the script, does type checking, and other miscellaneous checks. The output of the parser will be a DAG Latin statements and logical (directed acyclic graph), which represents the Pig operators.

In the DAG, the logical operators of the script are represented and the data flows are represented as edges.

Optimizer   as the nodes

The logical plan (DAG) is passed to the logical optimizer, which carries out the logical optimizations such as projection and pushdown.

Compiler

The compiler compiles the optimized logical plan into a series of MapReduce jobs.

Execution engine

Finally the MapReduce jobs are submitted to Hadoop in a sorted order. Finally, these MapReduce jobs are executed on Hadoop producing the desired results.

Going with the Pig Latin Application Flow

At its core, Pig Latin is a dataflow language, where we define a data stream and a series of transformations that are applied to the data as it flows through your application.

This is in contrast to a control flow language (like C or Java), where we write a series of instructions. In control flow languages, we use constructs like loops and conditional logic (like an if statement). You won't find loops and if statements in Pig Latin.

```
A = LOAD 'data_file.txt';
...
B = GROUP ... ;
...
C= FILTER ...;
...
DUMP B;
..
STORE C INTO 'Results';
```

Sample pig code

Looking at each line in turn, you can see the basic flow of a Pig program.

1. **Load:** we first load (LOAD) the data you want to manipulate. As in a typical MapReduce job, that data is stored in HDFS. For a Pig program to access the data, you first tell Pig what file or files to use. For that task, you use the LOAD 'data_file' command.
Here, 'data_file' can specify either an HDFS file or a directory. If a directory is specified, all files in that directory are loaded into the program.
If the data is stored in a file format that isn't natively accessible to Pig, you can optionally add the USING function to the LOAD statement to specify a user-defined function that can read in (and interpret) the data.

2. **Transform:** You run the data through a set of transformations that, way under the hood and far removed from anything you have to concern yourself with, are translated into a set of Map and Reduce tasks.
The transformation logic is where all the data manipulation happens. Here, you can FILTER out rows that aren't of interest, JOIN two sets of data files, GROUP data to build aggregations, ORDER results, and do much, much more.

3. **Dump:** Finally, you dump (DUMP) the results to the screen or Store (STORE) the results in a file somewhere.

Pig Latin Data Model(pig data types)

The data model of Pig Latin is fully nested and it allows complex non-atomic data types such as **map** and **tuple**. Given below is the diagrammatical representation of Pig Latin's data model.
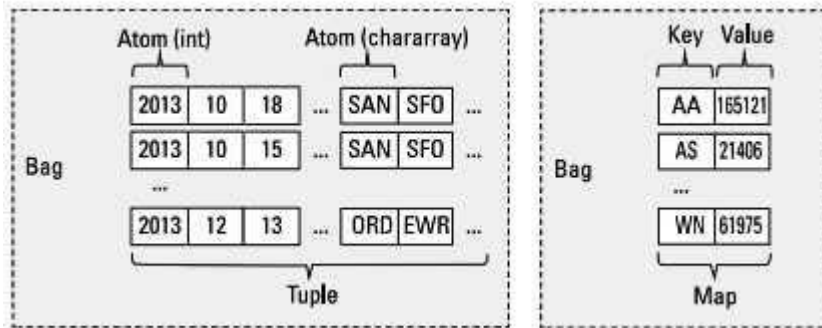
Atom: An atom is any single value, such as a string or a number — 'Diego', for example. Pig's atomic values are scalar types that appear in most programming languages — int, long, float, double chararray, and bytearray.

Tuple: A tuple is a record that consists of a sequence of fields. Each field can be of any type — 'Diego', 'Gomez', or 6, for example. Think of a tuple as a row in a table.

Bag: A bag is a collection of non-unique tuples. The schema of the bag is flexible — each tuple in the collection can contain an arbitrary number of fields, and each field can be of any type.

Map: A map is a collection of key value pairs. Any type can be stored in the value, and the key needs to be unique. The key of a map must be a chararray and the value can be of any type.

Pig latin opearations:

In a Hadoop context, accessing data means allowing developers to load, store, and stream data, whereas transforming data means taking advantage of Pig's ability to group, join, combine, split, filter, and sort data.

Pig latin operators

Operation	Operator	Explanation
Data Access	LOAD/STORE	Read and Write data to file system
	DUMP	Write output to standard output (stdout)
	STREAM	Send all records through external binary
	FOREACH	Apply expression to each record and output one or more records
	FILTER	Apply predicate and remove records that don't meet condition
	GROUP/ COGROUP	Aggregate records with the same key from one or more inputs
	JOIN	Join two or more records based on a condition
Transformations	CROSS	Cartesian product of two or more inputs
	ORDER	Sort records based on key
	DISTINCT	Remove duplicate records
	UNION	Merge two data sets
	SPLIT	Divide data into two or more bags based on predicate
	LIMIT	subset the number of records

Operators for Debugging and Trouble shooting

| Operation	Operator	Description
Debug	DESCRIBE	Return the schema of a relation.
	DUMP	Dump the contents of a relation to the screen.
	EXPLAIN	Display the MapReduce execution plans.

Apache Pig Execution Modes

we can run Apache Pig in two modes, namely, **Local Mode** and **HDFS mode**.

Local Mode

In this mode, all the files are installed and run from your local host and local file system. There is no need of Hadoop or HDFS. This mode is generally used for testing purpose.

MapReduce Mode

MapReduce mode is where we load or process the data that exists in the Hadoop File System (HDFS) using Apache Pig. In this mode, whenever we execute the Pig Latin statements to process the data, a MapReduce job is invoked in the back-end to perform a particular operation on the data that exists in the HDFS.
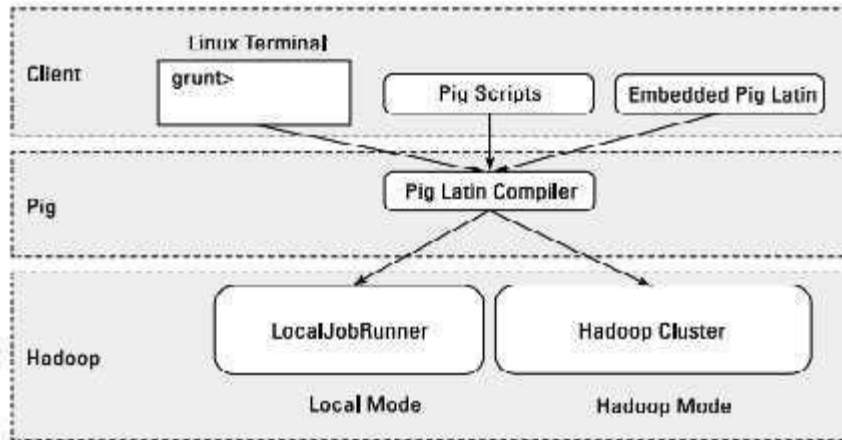
Apache Pig Execution Mechanisms

Apache Pig scripts can be executed in three ways, namely, interactive mode, batch mode, and embedded mode.

- **Interactive Mode** (Grunt shell) − You can run Apache Pig in interactive mode using the Grunt shell. In this shell, you can enter the Pig Latin statements and get the output (using Dump operator).

- **Batch Mode** (Script) − You can run Apache Pig in Batch mode by writing the Pig Latin script in a single file with **.pig** extension.

- **Embedded Mode** (UDF) − Apache Pig provides the provision of defining our own functions (**U**ser **D**efined **F**unctions) in programming languages such as Java, and using them in our script.

Important Questions

1. Explain Pig architecture in detail with neat diagram?
2. what is the difference between Hive and ApachePig?
3. what is the difference between MapReduce and ApachePig?
4. Apache Pig Scripting Statements and data type...?