UNIT-1

1. DISTRIBUTED PROGRAMMING USING JAVA:

Anatomy of a Distributed Application

A distributed application is built upon several layers. At the lowest level, a network connects a group of host computers together so that they can talk to each other. Network protocols like TCP/IP let the computers send data to each other over the network by providing the ability to package and address data for delivery to another machine. Higher-level services can be defined on top of the network protocol, such as directory services and security protocols. Finally, the distributed application itself runs on top of these layers, using the mid-level services and network protocols as well as the computer operating systems to perform coordinated tasks across the network.

At the application level, a distributed application can be broken down into the following parts:

Processes

A process is created by describing a sequence of steps in a programming language, compiling the program into an executable form, and running the executable in the operating system.

Threads

Every process has at least one thread of control. Some operating systems support the creation of multiple threads of control within a single process. Each thread in a process can run independently from the other threads, although there is usually some synchronization between them.

Objects

An object is a group of related data, with methods available for querying or altering the data (getName(), set-Name()), or for taking some action based on the data (sendName(Out-putStreamo)). Objects can be accessed by one or more threads within the process. And with the introduction of distributed object technology like RMI and CORBA, an object can also be logically spread across multiple processes, on multiple computers.

Agents

The term "agent" as a general way to refer to significant functional elements of a distributed application. an agent is a higher-level system component, defined around a particular function, or utility, or role in the overall system.

Example: A remote banking application, for example, might be broken down into a customer agent, a transaction agent and an information brokerage agent. Agents can be distributed across multiple processes, and can be made up of multiple objects and threads in these processes. Our customer agent might be made up of an object in a process running on a client desktop that's listening for data and updating the local display, along with an object in a process running on the bank server, issuing queries and sending the data back to the client.

The Client/Server Model

The client/server model is a form of distributed computing in which one program (the client) communicates with another program (the server) for the purpose of exchanging information. In this model, both the client and server usually speak the same language -- a protocol that both the client and server understand -- so they are able to communicate.

While the client/server model can be implemented in various ways, it is typically done using low-level sockets. Using sockets to develop client/server systems means that we must design a protocol, which is a set of commands agreed upon by the client and server through which they will be able to communicate.

The Distributed Objects Model

A distributed object-based system is a collection of objects that isolates the requesters of services (clients) from the providers of services (servers) by a well-defined encapsulating interface. In other words, clients are isolated from the implementation of services as data representations and executable code. This is one of the main differences that distinguishes the distributed object-based model from the pure client/server model.

In the distributed object-based model, a client sends a message to an object, which in turns interprets the message to decide what service to perform. This service, or method, selection could be performed by either the object or a broker. The Java Remote Method Invocation (RMI) and the Common Object Request Broker Architecture (CORBA) are examples of this model.

RMI

RMI is a distributed object system that enables you to easily develop distributed Java applications. Developing distributed applications in RMI is simpler than developing with sockets since there is no need to design a protocol, which is an error-prone task. In RMI, the developer has the illusion of calling a local method from a local class file, when in fact the arguments are shipped to the remote target and interpreted, and the results are sent back to the callers.

2. ADVANCED JAVA PROGRAMMING: GENERICS

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

- Collections can store Objects of any Type
- Generics restricts the Objects to be put in a collection
- Generics ease identification of runtime errors at compile time

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
1. List list = new ArrayList();
2. list.add("hello");
3. String s = (String) list.get(0);//typecasting After Generics, we don't need to typecast the object.
```

```
1. List<String> list = new ArrayList<String>();
2. list.add("hello");
3. String s = list.get(0);
```

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
1. List<String> list = new ArrayList<String>();
2. list.add("hello");
3. list.add(32);//Compile Time Error
```

Syntax to use generic collection

```
1. ClassOrInterface<Type>
```

Example to use Generics in java

```
1. ArrayList<String>
```

Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
```

```
        ArrayList<String> list=new ArrayList<String>();
        list.add("rahul");
        list.add("jai");
         //list.add(32);//compile time error
         String s=list.get(1);//type casting is not required
    System.out.println("element is: "+s);
        Iterator<String> itr=list.iterator();
        while(itr.hasNext()){
        System.out.println(itr.next());
    }
    }
    }
```

Output:elemen

t is: jai

rahul

jai

Generic class

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

Let's see the simple example to create and use the generic class.

Creating generic class:

```
        class MyGen<T>{
        T obj;
        void add(T obj){this.obj=obj;}
    T get(){return obj;
    }
     }
```

The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

Using generic class:

Let's see the code to use the generic class.

```
        class TestGenerics3{
        public static void main(String args[]){
    MyGen<Integer> m=new MyGen<Integer>();
    m.add(2);
    //m.add("vivek");//Compile time error
    System.out.println(m.get());
    }
```

```
}
```

Output:2

Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The commonly type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

Generic Method

Like generic class, we can create generic method that can accept any type of argument.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4{
      public static < E > void printArray(E[] elements) {
         for ( E element : elements){
            System.out.println(element );
          }
       System.out.println();         }
       public static void main( String args[] )
{
          Integer[] intArray = { 10, 20, 30, 40, 50 };
          Character[] charArray = { 'J', 'A', 'V', 'A','P','O','I','N','T' };
           System.out.println( "Printing Integer Array" );
          printArray( intArray );
         System.out.println( "Printing Character Array" );
        printArray( charArray );
}
}
```

Output: Printing Integer Array

```
10
20
30
40
50
```

Printing Character Array

J

A

V

A

P

O

I

N

T

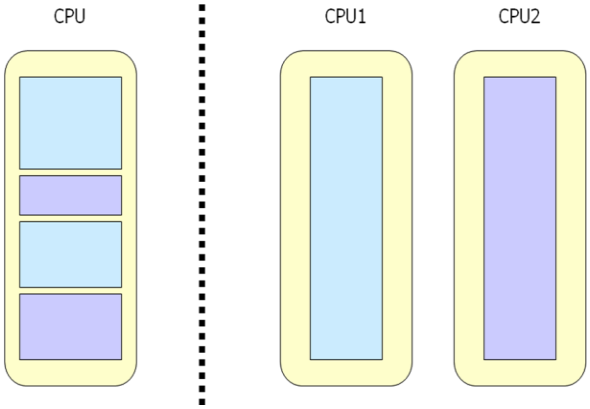
3. THREADS

Multitasking refers to a computer's ability to perform multiple jobs concurrently more than one program are running concurrently, e.g., UNIX

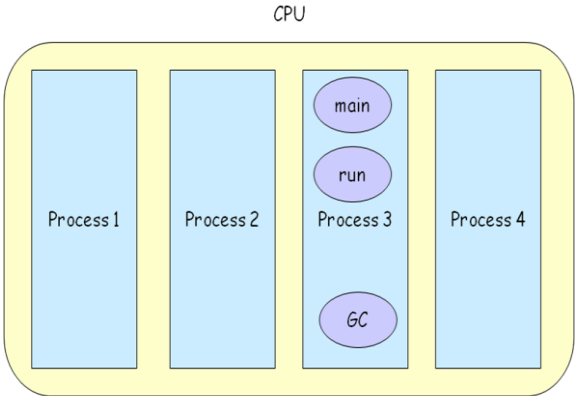
A **thread** is a single sequence of execution within a program

Multithreading refers to multiple threads of control within a single program each program can run multiple threads of control within it, e.g., Web Browser

Concurrency vs. Parallelism



Threads and Processes



What are Threads Good For?

- To maintain responsiveness of an application during a long running task.
- To enable cancellation of separable tasks.
- Some problems are intrinsically parallel.
- To monitor status of some resource (DB).
- Some APIs and systems demand it: Swing.

Application Thread

- When we execute an application:
  - The JVM creates a Thread object whose task is defined by the **main()** method
  - It starts the thread
  - The thread executes the statements of the program one by one until the method returns and the thread dies

Multiple Threads in an Application

- Each thread has its private run-time stack
- If two threads execute the same method, each will have its own copy of the local variables the methods uses
- However, all threads see the same dynamic memory (heap)
- Two different threads can act on the same object and same static fields concurrently

Creating Threads

- There are two ways to create our own **Thread** object
  1. Subclassing the **Thread** class and instantiating a new object of that class
  2. Implementing the **Runnable** interface
- In both cases the **run()** method should be implemented

Extending Thread

```
public class ThreadExample extends
  Thread { public void run () {
    for (int i = 1; i <= 100; i++) {
      System.out.println("Thread: " + i);
    }
  }
```

```
}
```

Implementing Runnable

```
public class RunnableExample implements
    Runnable { public void run () {
        for (int i = 1; i <= 100; i++) {
        System.out.println ("Runnable:
        " + i);
    }
    }
}
```

- The Thread object's **run()** method calls the Runnable object's **run()** method
- Allows threads to run inside any object, regardless of inheritance

Thread Methods

void start()	Creates a new thread and makes it runnable This method can be called only once
void run()	**The new thread begins its life inside this method**
void stop() (deprecated)	The thread is being terminated
yield()	Causes the currently executing thread object to temporarily pause and allow other threads to execute Allow only threads of the same priority to run.

sleep(int *m*)/sleep(int *m*,int *n*)

The thread sleeps for *m* milliseconds, plus *n* nanoseconds

Starting the Threads

```
public class ThreadsStartExample
{
    public static void main (String argv[])
    {
        new ThreadExample ().start ();
        new Thread(new RunnableExample ()).start ();
    }
}
```

Example

```
public class PrintThread1 extends Thread
{
        String name;
        public PrintThread1(String name)
        {
          this.name = name;
        }
        public void run()
      {
            for (int i=1; i<500 ; i++)
            {
           try {
                  sleep((long)(Math.random() * 100));
                } catch (InterruptedException ie) { }
          System.out.print(name);
            }
      }
      public static void main(String args[])
      {
          PrintThread1 a = new PrintThread1("*");
          PrintThread1 b = new PrintThread1("-");
          PrintThread1 c = new PrintThread1("="); a.start();
          b.start();
          c.start();
      }
  }
```

- **Thread scheduling** is the mechanism used to determine how runnable threads are allocated CPU time
- A thread-scheduling mechanism is either **preemptive** or **nonpreemptive**

Thread Priority

- Every thread has a priority
- When a thread is created, it inherits the priority of the thread that created it
- The priority values range from 1 to 10, in increasing priority
- The priority can be adjusted subsequently using the **setPriority()** method

- The priority of a thread may be obtained using **getPriority()**
- Priority constants are defined:
    - MIN_PRIORITY=1
    - MAX_PRIORITY=10
    - NORM_PRIORITY=5
- Thread implementation in Java is actually based on operating system support

Some Windows operating systems support only 7 priority levels, so different levels in Java may actually be mapped to the same operating system level

Daemon Threads

- Daemon threads are "***background***" threads, that provide services to other threads, e.g., the garbage collection thread
- The Java VM will not exit if non-Daemon threads are executing
- The Java VM will exit if only Daemon threads are executing
- Daemon threads die when the Java VM exits

Multithreading

- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.
- One thread can pause without stopping other parts of your program.
- For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.
- Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.
- When a thread blocks in a Java program, only the single thread that is blocked pauses.
- All other threads continue to run.

Concurrency

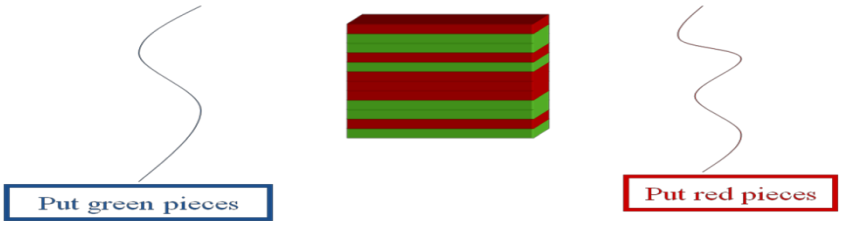
- An object in a program can be changed by more than one thread

Race Condition

- A race condition – the outcome of a program is affected by the order in which the program's threads are allocated CPU time
- Two threads are simultaneously modifying a single object

Both threads "race" to store their value

Race Condition Example



Thread Synchronization

- We need to synchronized between transactions, for example, the consumer-producer scenario

- Allows two threads to cooperate
- Based on a single shared lock object
  - Marge put a cookie wait and notify Homer
  - Homer eat a cookie wait and notify Marge
    - Marge put a cookie wait and notify Homer
    - Homer eat a cookie wait and notify Marge

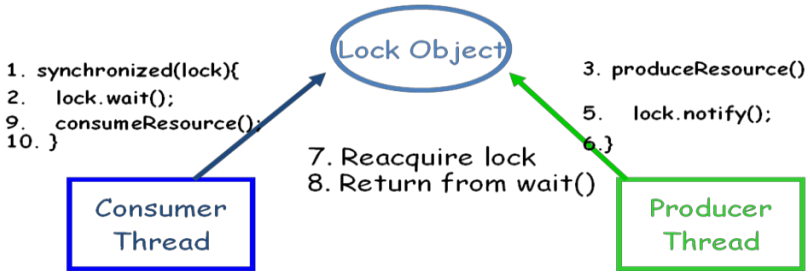
Consumer

```
synchronized (lock) {
  while (!resourceAvailable()) {
   lock.wait();
  }
  consumeResource();
  }
```

Producer produceResource();

```
 synchronized (lock)
 {
 lock.notifyAll();
```

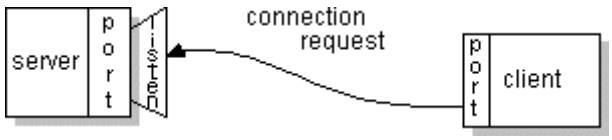
Wait/Notify Sequence



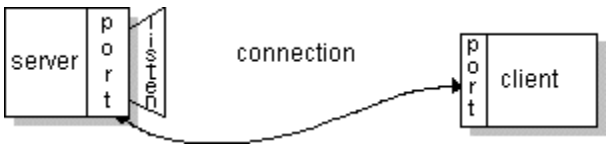
SOCKETS:

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Definition:

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

An endpoint is a combination of an **IP address** and a **port number**. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

Socket class

The Socket class can be used to create a socket.

Important methods

Method	Description
1) public InputStream getInputStream()	returns the InputStream attached with this socket.
2)public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
3) public synchronized void close()	closes this socket

ServerSocket class

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

Important methods

Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

Example of Java Socket Programming

- Refer next section **5. Simple Client Server Programming Using Java**

SIMPLE CLIENT SERVER PROGRAMMING USING JAVA

Let's see a simple of java socket programming in which client sends a text and server receives it.

File: MyServer.java

```
import java.io.*;
import java.net.*;
public class MyServer {
public static void main(String[] args){
try{
ServerSocket ss=new ServerSocket(6666);
Socket s=ss.accept();//establishes connection
DataInputStream dis=new DataInputStream(s.getInputStream());
String  str=(String)dis.readUTF();
System.out.println("message= "+str);
ss.close();
}catch(Exception e){System.out.println(e);}
    }
}
```

File: MyClient.java

```
import java.io.*;
import java.net.*;
public class MyClient {
public static void main(String[] args) {
try{
Socket s=new Socket("localhost",6666);
DataOutputStream dout=new DataOutputStream(s.getOutputStream());
dout.writeUTF("Hello Server");
dout.flush();
dout.close();
s.close();
}catch(Exception e){System.out.println(e);}
    }
}
```

To execute this program open two command prompts and execute each program at each command prompt as displayed in the below figure.

After running the client application, a message will be displayed on the server console.

Example-2: Client Server Multithread programming

Server

```
import java.net.*;

import java.io.*;

class HelloServer {

public static void main(String[] args)

{

int port = Integer.parseInt(args[0]);

try{

ServerSocket server = new ServerSocket(port);

}

catch (IOException ioe)

{


}

while(true)

{  System.err.println("Couldn't run " + "server on port " + port);

return;
try {

Socket connection = server.accept();

ConnectionHandler handler = new ConnectionHandler(connection);

 new Thread(handler).start();

} catch (IOException ioe1) { }

}
```

Connection Handler

```
// Handles a connection of a client to an HelloServer.
```

```
// Talks with the client in the 'hello'
protocol class ConnectionHandler
implements Runnable
{
// The connection with the client private Socket connection;
    public ConnectionHandler(Socket connection)
    {
      this.connection = connection;
    }
  public void run()
  {
      try {
BufferedReader reader = new BufferedReader(new InputStreamReader( connection.getInputStream()));
PrintWriter writer = new PrintWriter(new OutputStreamWriter( connection.getOutputStream()));
String clientName = reader.readLine();
writer.println("Hello " + clientName);
writer.flush();
      } catch (IOException ioe) {}
    }
}
```

Client side

```
import java.net.*; import java.io.*;
// A client of an HelloServer class
HelloClient {
public static void main(String[] args)
{
String hostname = args[0];
int port = Integer.parseInt(args[1]);
Socket connection = null;
        try
        {
          connection = new Socket(hostname, port);
        }
        catch (IOException ioe)
        {  }
```

```
try
{System.err.println("Connection failed"); return;
BufferedReader reader = new BufferedReader(new InputStreamReader( connection.getInputStream()));
PrintWriter writer = new PrintWriter(new OutputStreamWriter( connection.getOutputStream()));
        writer.println(args[2]); // client name
        String reply = reader.readLine();
        System.out.println("Server reply: "+reply);
        writer.flush();
                                }
                        catch (IOException ioe1) { }
                }
```

6. Difficulties In Developing Distributed Programs For Large Scale Clusters

Designing and implementing a distributed program for the Large Scale Clusters involves more than just sending and receiving messages and deciding upon the computational and architectural models. While all these are extremely important, they do not reflect the whole story of developing programs for the Distributed Programs For Large Scale Clusters.

Some Difficulties In Developing Distributed Programs For Large Scale Clusters are:

1. Heterogeneity
2. Scalability
3. Communication
4. Synchronization
5. fault-tolerance and
6. Security and privacy:
7. scheduling
8. Openness and Extensibility
9. Transparency

1 Heterogeneity

- distributed programs must be designed in a way that masks the heterogeneity of the underlying hardware, networks, OSs, and the programming languages
- Another serious problem that requires a great deal of attention from distributed programmers is performance variation.
- Performance variation entails that running the same distributed program on the same cluster twice can result in largely different execution times.
- Clearly, this can create tricky load-imbalance and subsequently degrade overall performance

2. Scalability

- A distributed program is said to be scalable if it remains effective when the quantities of users, data and resources are increased significantly
- Requires tens and hundreds of thousands of machines to maintain performance and load

3. Communication

- Distributed systems are composed of networked computers that can communicate by explicitly passing messages or implicitly accessing shared memories.
- Even with distributed shared memory systems, messages are internally passed between machines, yet in a manner that is totally transparent to users.
- Distributed systems such as the Big Data rely heavily on the underlying network to deliver messages rapidly enough to destination entities for three main reasons, performance, cost and quality of service (QoS).
- Specifically, faster delivery of messages entails minimized execution times, reduced costs and higher QoS, especially for audio and video applications.
- Communication is at the heart of the **Large Scale Clusters** and is one of its major bottlenecks.

4. Synchronization

- Distributed tasks should be allowed to simultaneously operate on shared data without corrupting data or causing any inconsistency
- Race-conditions whereby two tasks might try to modify data on a shared edge at the same time, resulting in a corrupted value.
- Wide use of semaphores, locks and barriers
- Avoiding the deadlock and practicing mutual exclusions are need to apply for synchronizing the data

5. Fault-tolerance

- The ability to tolerate faults in software system is required in applications like nuclear plant, Space missions, medical equipments etc.
- Different fault injection techniques are used for fault tolerance by injecting faults in the system under test.

6. Security and privacy:

- How to apply the security policies to the interdependent system is a great issue in distributed system. Since distributed systems deal with sensitive data and information so the system must have a strong security and privacy measurement.
- Protection of distributed system assets, including base resources, storage, communications and user-interface I/O as well as higher-level composites of these resources, like processes, files, messages, display windows and more complex objects, are important issues in distributed system

7.Scheduling:

- Focuses on Scheduling problems in homogeneous and heterogeneous parallel distributed
- systems. The performance of distributed systems are affected by Broadcast/multicast processing and required to develop a delivering procedure that completes the processing in minimum time.

8.Openness and Extensibility:

- Interfaces should be separated and publicly available to enable easy extensions to existing components and add new components

9.Transparency:

- Transparency means up to what extent the distributed system program should appear to the user as a single system? Distributed system must be designed to hide the complexity of the system to a greater extent.

Introduction to cloud computing

Cloud computing is a form of Internet-based computing that provides shared computer processing resources and data to computers and other devices on demand.

Characteristics

Cloud computing has a variety of characteristics, with the main ones being:

Shared Infrastructure

- Uses a virtualized software model, enabling the sharing of physical services, storage, and networking capabilities.
- The cloud infrastructure, regardless of deployment model, seeks to make the most of the available infrastructure across a number of users.

Dynamic Provisioning

- Allows for the provision of services based on current demand requirements. This is done automatically using software automation, enabling the expansion and contraction of service capability, as needed. This dynamic scaling needs to be done while maintaining high levels of reliability and security

Network Access

- Needs to be accessed across the internet from a broad range of devices such as PCs, laptops, and mobile devices, using standards-based APIs (for example, ones based on HTTP).
- Deployments of services in the cloud include everything from using business applications to the latest application on the newest smartphones.

Managed Metering

- Uses metering for managing and optimizing the service and to provide reporting and billing information.
- In this way, consumers are billed for services according to how much they have actually used during the billing period.

Service Models

Once a cloud is established, how its cloud computing services are deployed in terms of business models can differ depending on requirements. The primary service models being deployed (see Figure 1) are commonly known as:

• Software as a Service (SaaS)

- Consumers purchase the ability to access and use an application or service that is hosted in the cloud.
- A benchmark example of this is Salesforce.com, as discussed previously, where necessary information for the interaction between the consumer and the service is hosted as part of the service in the cloud.
- Also, Microsoft is expanding its involvement in this area, and as part of the cloud computing option for Microsoft Office 2010, its Office Web Apps are available to Office volume licensing customers and Office Web App subscriptions through its cloud-based Online Services.

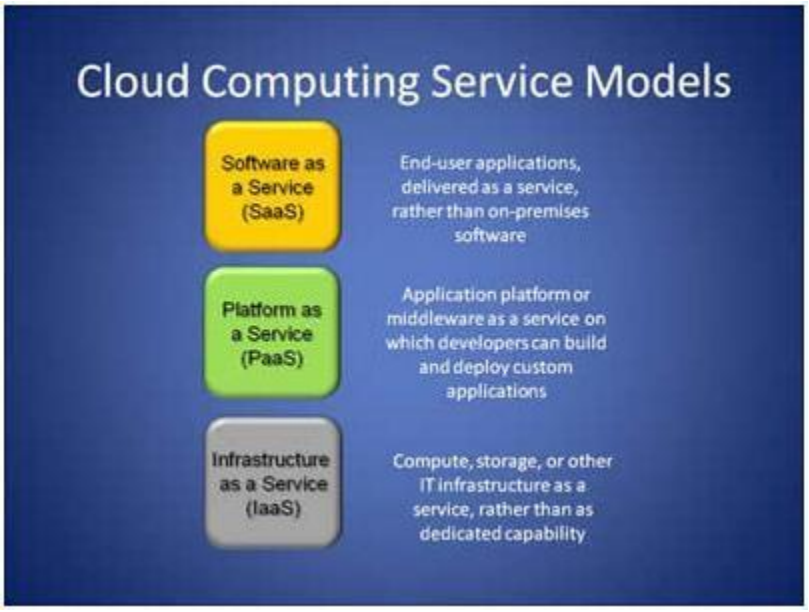
• Platform as a Service (PaaS)

- Consumers purchase access to the platforms, enabling them to deploy their own software and applications in the cloud. The operating systems and network access are not managed by the consumer, and there might be constraints as to which applications can be deployed.

• Infrastructure as a Service (IaaS)

- Consumers control and manage the systems in terms of the operating systems, applications, storage, and network connectivity, but do not themselves control the cloud infrastructure.

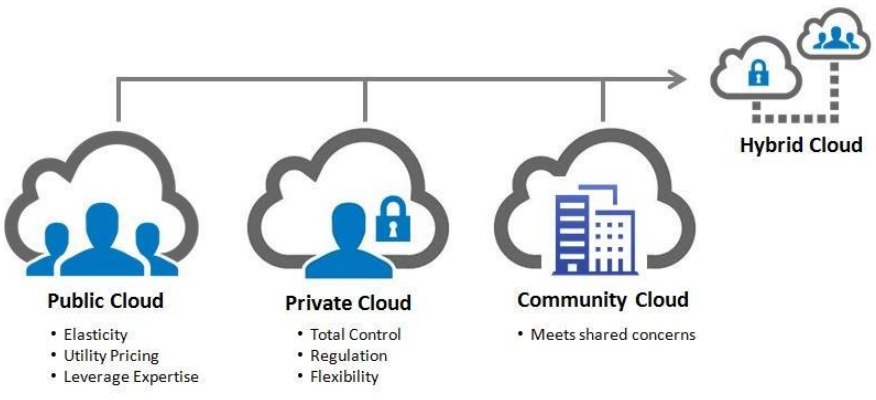
Also known are the various subsets of these models that may be related to a particular industry or market. Communications as a Service (CaaS) is one such subset model used to describe hosted IP telephony services. Along with the move to CaaS is a shift to more IP-centric communications and more SIP trunking deployments. With IP and SIP in place, it can be as easy to have the PBX in the cloud as it is to have it on the premise. In this context, CaaS could be seen as a subset of SaaS.



Deployment Models

Deploying cloud computing can differ depending on requirements, and the following four deployment models have been identified, each with specific characteristics that support the needs of the services and users of the clouds in particular ways (see Figure 2).

- **Private Cloud**
  - The cloud infrastructure has been deployed, and is maintained and operated for a specific organization. The operation may be in-house or with a third party on the premises.
- **Community Cloud**
  - The cloud infrastructure is shared among a number of organizations with similar interests and requirements.
  - This may help limit the capital expenditure costs for its establishment as the costs are shared among the organizations. The operation may be in-house or with a third party on the premises.
- **Public Cloud**
  - The cloud infrastructure is available to the public on a commercial basis by a cloud service provider. This enables a consumer to develop and deploy a service in the cloud



with very little financial outlay compared to the capital expenditure requirements normally associated with other deployment options.

- **Hybrid Cloud**
  - The cloud infrastructure consists of a number of clouds of any type, but the clouds have the ability through their interfaces to allow data and/or applications to be moved from one cloud to another. This can be a combination of private and public clouds that support the requirement to retain some data in an organization, and also the need to offer services in the cloud

Benefits

The following are some of the possible benefits for those who offer cloud computing-based services and applications:

- **Cost Savings**
  — Companies can reduce their capital expenditures and use operational expenditures for increasing their computing capabilities. This is a lower barrier to entry and also requires fewer in- house IT resources to provide system support.
- **Scalability/Flexibility**
  — Companies can start with a small deployment and grow to a large deployment fairly rapidly, and then scale back if necessary. Also, the flexibility of cloud computing allows companies to use extra resources at peak times, enabling them to satisfy consumer demands.
- **Reliability**
  — Services using multiple redundant sites can support business continuity and disaster recovery.
- **Maintenance**
  — Cloud service providers do the system maintenance, and access is through APIs that do not require application installations onto PCs, thus further reducing maintenance requirements.
- **Mobile Accessible**
  — Mobile workers have increased productivity due to systems accessible in an infrastructure available from anywhere.

Challenges

The following are some of the notable challenges associated with cloud computing, and although some of these may cause a slowdown when delivering more services in the cloud, most also can provide opportunities, if resolved with due care and attention in the planning stages.

- **Security and Privacy**
  — Perhaps two of the more "hot button" issues surrounding cloud computing relate to storing and securing data, and monitoring the use of the cloud by the service providers. These issues

are generally attributed to slowing the deployment of cloud services. These challenges can be addressed, for example, by storing the information internal to the organization, but allowing it to be used in the cloud. For this to occur, though, the security mechanisms between organization and the cloud need to be robust and a Hybrid cloud could support such a deployment.

- **Lack of Standards**

  — Clouds have documented interfaces; however, no standards are associated with these, and thus it is unlikely that most clouds will be interoperable. The Open Grid Forum is developing an Open Cloud Computing Interface to resolve this issue and the Open Cloud Consortium is working on cloud computing standards and practices. The findings of these groups will need to mature, but it is not known whether they will address the needs of the people deploying the services and the specific interfaces these services need. However, keeping up to date on the latest standards as they evolve will allow them to be leveraged, if applicable.

- **Continuously Evolving**

  — User requirements are continuously evolving, as are the requirements for interfaces, networking, and storage. This means that a "cloud," especially a public one, does not remain static and is also continuously evolving.

UNIT-2

1. Distributed File systems leading to Hadoop file system

A distributed file system is mainly designed to hold a large amount of data and provide access to this data to many clients distributed across a network. But a distributed file system has got many limitations.

1. The files reside on on single machine.
2. It does not provide any reliability guarantees if that machine goes down, this means that it will only store as much information as can be stored in one machine.
3. Finally, as all the data is stored on a single machine, all the clients must go to this machine to retrieve their data. This can overload the server if a large number of clients must be handled. Clients must also always copy the data to their local machines before they can operate on it.

To overcome above drawbacks, there came a file system — HDFS (Hadoop Distributed File System.)

1. HDFS is designed to store a very large amount of information (terabytes or petabytes). This requires spreading the data across a large number of machines. It also supports much larger file sizes than DFS.
2. HDFS should store data reliably. If individual machines in the cluster malfunction, data should still be available.
3. HDFS should provide fast, scalable access to this information. It should be possible to serve a larger number of clients by simply adding more machines to the cluster.
4. HDFS should integrate well with Hadoop MapReduce, allowing data to be read and computed upon locally when possible.

But, HDFS has also got some limitations.

1. HDFS is optimized to provide streaming read performance; this comes at the expense of random seek times to arbitrary positions in files.
2. Data will be written to the HDFS once and then read several times; updates to files after they have already been closed are not supported.
3. Due to the large size of files, and the sequential nature of reads, the system does not provide a mechanism for local caching of data.
4. Individual machines are assumed to fail on a frequent basis, both permanently and intermittently. The cluster must be able to withstand the complete failure of several machines, possibly many happening at the same time.

2. Introduction

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS also makes applications available to parallel processing.

HDFS Goals

1. Hardware Failure

   Hardware failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. The fact that there are a huge number of components and that each component has a non-trivial probability of failure means that some component of HDFS is always non-functional. Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

2. Streaming Data Access

   Applications that run on HDFS need streaming access to their data sets. They are not general purpose applications that typically run on general purpose file systems. HDFS is designed more for batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access. POSIX imposes many hard requirements that are not needed for applications that are targeted for HDFS. POSIX semantics in a few key areas has been traded to increase data throughput rates.

3. Large Data Sets

   Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.

4. Simple Coherency Model

   HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access. A MapReduce application or a web crawler application fits perfectly with this model. There is a plan to support appending-writes to files in the future.

5. "Moving Computation is Cheaper than Moving Data"

A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system. The assumption is that it is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running. HDFS provides interfaces for applications to move themselves closer to where the data is located.

6. Portability Across Heterogeneous Hardware and Software Platforms

HDFS has been designed to be easily portable from one platform to another. This facilitates widespread adoption of HDFS as a platform of choice for a large set of applications.

3. Using HDFS

Features of Hadoop HDFS

1. Fault Tolerance

Fault tolerance in HDFS refers to the working strength of a system in unfavorable conditions and how that system can handle such situations. HDFS is highly fault-tolerant, in HDFS data is divided into blocks and multiple copies of blocks are created on different machines in the cluster (this replica creation is configurable). So whenever if any machine in the cluster goes down, then a client can easily access their data from the other machine which contains the same copy of data blocks. HDFS also maintains the replication factor by creating a replica of blocks of data on another rack. Hence if suddenly a machine fails, then a user can access data from other slaves present in another rack.

2. High Availability

HDFS is a highly available file system, data gets replicated among the nodes in the HDFS cluster by creating a replica of the blocks on the other slaves present in HDFS cluster. Hence whenever a user wants to access his data, they can access their data from the slaves which contains its blocks and which is available on the nearest node in the cluster. And during unfavorable situations like a failure of a node, a user can easily access their data from the other nodes. Because duplicate copies of blocks which contain user data are created on the other nodes present in the HDFS cluster.

3. Data Reliability

HDFS is a distributed file system which provides reliable data storage. HDFS can store data in the range of 100s of petabytes. It stores data reliably on a cluster of nodes. HDFS divides the data into blocks and these blocks are stored on nodes present in HDFS cluster. It stores data reliably by creating a replica of each and every block present on the nodes present in the cluster and hence provides fault tolerance facility. If node containing data goes down, then a user can easily access that data from the other nodes which contain a copy of same data in the HDFS cluster. HDFS by default creates 3 copies of blocks containing data

present in the nodes in HDFS cluster. Hence data is quickly available to the users and hence user does not face the problem of data loss. Hence HDFS is highly reliable.

4. Replication

Data Replication is one of the most important and unique features of Hadoop HDFS. In HDFS replication of data is done to solve the problem of data loss in unfavorable conditions like crashing of a node, hardware failure, and so on. As data is replicated across a number of machines in the cluster by creating blocks. The process of replication is maintained at regular intervals of time by HDFS and HDFS keeps creating replicas of user data on different machines present in the cluster. So whenever any machine in the cluster gets crashed, the user can access their data from other machines which contain the blocks of that data. Hence there is no possibility of losing of user data.

5. Scalability

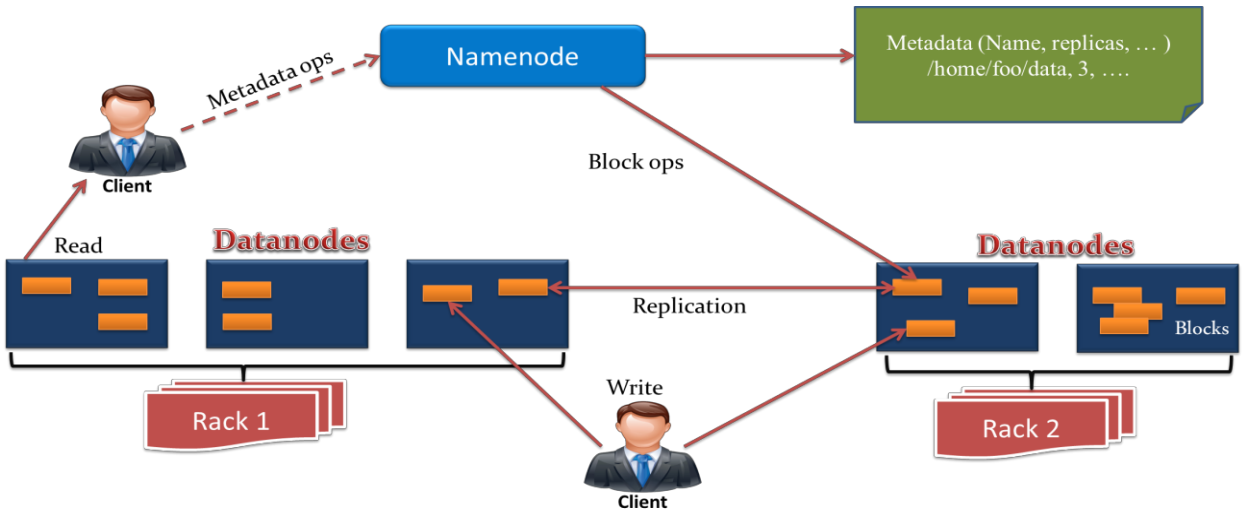
As HDFS stores data on multiple nodes in the cluster, when requirements increase we can scale the cluster. There is two scalability mechanisms available: Vertical scalability – add more resources (CPU, Memory, Disk) on the existing nodes of the cluster. Another way is horizontal scalability – Add more machines in the cluster. The horizontal way is preferred as we can scale the cluster from 10s of nodes to 100s of nodes on the fly without any downtime.

6. Distributed Storage

In HDFS all the features are achieved via distributed storage and replication. In HDFS data is stored in distributed manner across the nodes in HDFS cluster. In HDFS data is divided into blocks and is stored on the nodes present in HDFS cluster. And then replicas of each and every block are created and stored on other nodes present in the cluster. So if a single machine in the cluster gets crashed we can easily access our data from the other nodes which contain its replica.

Hadoop Architecture

Hadoop HDFS has a **Master/Slave** architecture in which *Master* is **NameNode** and *Slave* is



DataNode. HDFS Architecture consists of single NameNode and all the other nodes are DataNodes.

1. HDFS NameNode

It is also known as *Master node*. HDFS Namenode stores *meta-data* i.e. number of **data blocks**, replicas and other details. This meta-data is available in memory in the master for faster retrieval of data. NameNode maintains and manages the slave nodes, and assigns tasks to them. It should deploy on reliable hardware as it is the centerpiece of HDFS.

Task of NameNode

- Manage file system namespace.
- Regulates client's access to files.
- It also executes file system execution such as naming, closing, opening files/directories.
- All DataNodes sends a Heartbeat and block report to the NameNode in the **Hadoop cluster.** It ensures that the DataNodes are alive. A block report contains a list of all blocks on a datanode.
- NameNode is also responsible for taking care of the *Replication Factor* of all the blocks.

Files present in the NameNode metadata are as follows-

FsImage –

It is an "Image file". **FsImage** contains the entire filesystem namespace and stored as a file in the namenode's local file system. It also contains a serialized form of all the directories and file inodes in the filesystem. Each *inode* is an internal representation of file or directory's metadata.

EditLogs –

It contains all the recent modifications made to the file system on the most recent FsImage. Namenode receives a create/update/delete request from the client. After that this request is first recorded to edits file.

2. HDFS DataNode

It is also known as *Slave*. In **Hadoop HDFS Architecture**, DataNode stores actual data in HDFS. It performs **read and write operation** as per the request of the client. DataNodes can deploy on commodity hardware.

Task of DataNode

- Block replica creation, deletion, and replication according to the instruction of Namenode.
- DataNode manages data storage of the system.
- DataNodes send heartbeat to the NameNode to report the health of HDFS. By default, this frequency is set to 3 seconds.

3. Secondary NameNode

In HDFS, when NameNode starts, first it reads HDFS state from an image file, FsImage. After that, it applies edits from the edits log file. NameNode then writes new HDFS state to the FsImage. Then it starts normal operation with an empty edits file. At the time of start-up, NameNode merges FsImage and edits files, so the edit log file could get very large over time. A side effect of a larger edits file is that next restart of Namenode takes longer.

Secondary Namenode solves this issue. Secondary NameNode downloads the FsImage and EditLogs from the NameNode. And then merges EditLogs with the FsImage (FileSystem Image). It keeps edits log size within a limit. It stores the modified FsImage into persistent storage. And we can use it in the case of NameNode failure.

Secondary NameNode performs a regular checkpoint in HDFS.

4. Checkpoint Node

The **Checkpoint node** is a node which periodically creates checkpoints of the namespace. Checkpoint Node in Hadoop first downloads FsImage and edits from the Active Namenode. Then it merges them (FsImage and edits) locally, and at last, it uploads the new image back to the active NameNode. It stores the latest checkpoint in a directory that has the same structure as the Namenode's directory. This permits the checkpointed image to be always available for reading by the namenode if necessary.

5. Backup Node

A **Backup node** provides the same checkpointing functionality as the Checkpoint node. In Hadoop, Backup node keeps an in-memory, up-to-date copy of the file system namespace. It is always synchronized with the active NameNode state. The backup node in HDFS Architecture does not need to download FsImage and edits files from the active NameNode to create a checkpoint. It already has an up-to-date state of the namespace state in memory. The Backup node checkpoint process is more efficient as it only needs to save the namespace into the local FsImage file and reset edits. NameNode supports one Backup node at a time.

6. The File System Namespace

HDFS supports a traditional hierarchical file organization. A user or an application can create directories and store files inside these directories. The file system namespace hierarchy is similar to most other existing file systems; one can create and remove files, move a file from one directory to another, or rename a file. HDFS does not yet implement user quotas. HDFS does not support hard links or soft links. However, the HDFS architecture does not preclude implementing these features.

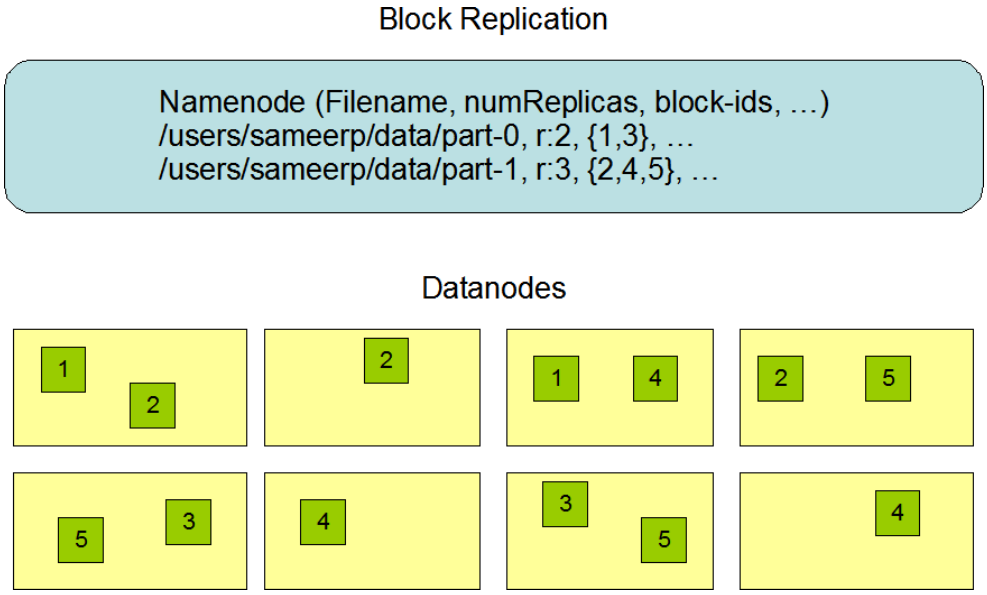
The NameNode maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the NameNode. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file. This information is stored by the NameNode.

5. Internals of Hadoop File Systems

Data Replication

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.



Replica Placement: The First Baby Steps

The placement of replicas is critical to HDFS reliability and performance. Optimizing replica placement distinguishes HDFS from most other distributed file systems. This is a feature that needs lots of tuning and experience. The purpose of a rack-aware replica placement policy is to improve data reliability, availability, and network bandwidth utilization. The current implementation for the replica placement policy is a first effort in this direction. The short-term goals of implementing this policy are to validate it on production systems, learn more about its behavior, and build a foundation to test and research more sophisticated policies.

Large HDFS instances run on a cluster of computers that commonly spread across many racks. Communication between two nodes in different racks has to go through switches. In

most cases, network bandwidth between machines in the same rack is greater than network bandwidth between machines in different racks.

The NameNode determines the rack id each DataNode belongs to via the process outlined in Hadoop Rack Awareness. A simple but non-optimal policy is to place replicas on unique racks. This prevents losing data when an entire rack fails and allows use of bandwidth from multiple racks when reading data. This policy evenly distributes replicas in the cluster which makes it easy to balance load on component failure. However, this policy increases the cost of writes because a write needs to transfer blocks to multiple racks.

For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a node in a different (remote) rack, and the last on a different node in the same remote rack. This policy cuts the inter-rack write traffic which generally improves write performance. The chance of rack failure is far less than that of node failure; this policy does not impact data reliability and availability guarantees. However, it does reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three. With this policy, the replicas of a file do not evenly distribute across the racks. One third of replicas are on one node, two thirds of replicas are on one rack, and the other third are evenly distributed across the remaining racks. This policy improves write performance without compromising data reliability or read performance.

The current, default replica placement policy described here is a work in progress.

Replica Selection

To minimize global bandwidth consumption and read latency, HDFS tries to satisfy a read request from a replica that is closest to the reader. If there exists a replica on the same rack as the reader node, then that replica is preferred to satisfy the read request. If angg/ HDFS cluster spans multiple data centers, then a replica that is resident in the local data center is preferred over any remote replica.

Safemode

On startup, the NameNode enters a special state called Safemode. Replication of data blocks does not occur when the NameNode is in the Safemode state. The NameNode receives Heartbeat and Blockreport messages from the DataNodes. A Blockreport contains the list of data blocks that a DataNode is hosting. Each block has a specified minimum number of replicas. A block is considered safely replicated when the minimum number of replicas of that data block has checked in with the NameNode. After a configurable percentage of safely replicated data blocks checks in with the NameNode (plus an additional 30 seconds), the NameNode exits the Safemode state. It then determines the list of data blocks (if any) that still have fewer than the specified number of replicas. The

NameNode then replicates these blocks to other DataNodes.

The Persistence of File System Metadata

The HDFS namespace is stored by the NameNode. The NameNode uses a transaction log called the EditLog to persistently record every change that occurs to file system metadata. For example, creating a new file in HDFS causes the NameNode to insert a record into the EditLog indicating this. Similarly, changing the replication factor of a file causes a new record to be inserted into the EditLog. The NameNode uses a file in its local host OS file system to store the EditLog. The entire file system namespace, including the mapping of blocks to files and file system properties, is stored in a file called the FsImage. The FsImage is stored as a file in the NameNode's local file system too.

The NameNode keeps an image of the entire file system namespace and file Blockmap in memory. This key metadata item is designed to be compact, such that a NameNode with 4 GB of RAM is plenty to support a huge number of files and directories. When the NameNode starts up, it reads the FsImage and EditLog from disk, applies all the transactions from the EditLog to the in-memory representation of the FsImage, and flushes out this new version into a new FsImage on disk. It can then truncate the old EditLog because its transactions have been applied to the persistent FsImage. This process is called a checkpoint. In the current implementation, a checkpoint only occurs when the NameNode starts up. Work is in progress to support periodic checkpointing in the near future.

The DataNode stores HDFS data in files in its local file system. The DataNode has no knowledge about HDFS files. It stores each block of HDFS data in a separate file in its local file system. The DataNode does not create all files in the same directory. Instead, it uses a heuristic to determine the optimal number of files per directory and creates subdirectories appropriately. It is not optimal to create all local files in the same directory because the local file system might not be able to efficiently support a huge number of files in a single directory. When a DataNode starts up, it scans through its local file system, generates a list of all HDFS data blocks that correspond to each of these local files and sends this report to the NameNode: this is the Blockreport.

The Communication Protocols

All HDFS communication protocols are layered on top of the TCP/IP protocol. A client establishes a connection to a configurable TCP port on the NameNode machine. It talks the ClientProtocol with the NameNode. The DataNodes talk to the NameNode using the DataNode Protocol. A Remote Procedure Call (RPC) abstraction wraps both the Client Protocol and the DataNode Protocol. By design, the NameNode never initiates any RPCs. Instead, it only responds to RPC requests issued by DataNodes or clients.

Robustness

The primary objective of HDFS is to store data reliably even in the presence of failures. The three common types of failures are NameNode failures, DataNode failures and network

partitions.

Data Disk Failure, Heartbeats and Re-Replication

Each DataNode sends a Heartbeat message to the NameNode periodically. A network partition can cause a subset of DataNodes to lose connectivity with the NameNode. The NameNode detects this condition by the absence of a Heartbeat message. The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them. Any data that was registered to a dead DataNode is not available to HDFS any more. DataNode death may cause the replication factor of some blocks to fall below their specified value. The NameNode constantly tracks which blocks need to be replicated and initiates replication whenever necessary. The necessity for re-replication may arise due to many reasons: a DataNode may become unavailable, a replica may become corrupted, a hard disk on a DataNode may fail, or the replication factor of a file may be increased.

The time-out to mark DataNodes dead is conservatively long (over 10 minutes by default) in order to avoid replication storm caused by state flapping of DataNodes. Users can set shorter interval to mark DataNodes as stale and avoid stale nodes on reading and/or writing by configuration for performance sensitive workloads.

Cluster Rebalancing

The HDFS architecture is compatible with data rebalancing schemes. A scheme might automatically move data from one DataNode to another if the free space on a DataNode falls below a certain threshold. In the event of a sudden high demand for a particular file, a scheme might dynamically create additional replicas and rebalance other data in the cluster. These types of data rebalancing schemes are not yet implemented.

Data Integrity

It is possible that a block of data fetched from a DataNode arrives corrupted. This corruption can occur because of faults in a storage device, network faults, or buggy software. The HDFS client software implements checksum checking on the contents of HDFS files. When a client creates an HDFS file, it computes a checksum of each block of the file and stores these checksums in a separate hidden file in the same HDFS namespace. When a client retrieves file contents it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file. If not, then the client can opt to retrieve that block from another DataNode that has a replica of that block.

Metadata Disk Failure

The FsImage and the EditLog are central data structures of HDFS. A corruption of these files can cause the HDFS instance to be non-functional. For this reason, the NameNode can be configured to support maintaining multiple copies of the FsImage and EditLog. Any update to either the FsImage or EditLog causes each of the FsImages and EditLogs to get updated synchronously. This synchronous updating of multiple copies of the FsImage and

EditLog may degrade the rate of namespace transactions per second that a NameNode can support. However, this degradation is acceptable because even though HDFS applications are very data intensive in nature, they are not metadata intensive. When a NameNode restarts, it selects the latest consistent FsImage and EditLog to use.

Another option to increase resilience against failures is to enable High Availability using multiple NameNodes either with a shared storage on NFS or using a distributed edit log (called Journal). The latter is the recommended approach.

Snapshots

Snapshots support storing a copy of data at a particular instant of time. One usage of the snapshot feature may be to roll back a corrupted HDFS instance to a previously known good point in time.

Data Organization

Data Blocks

HDFS is designed to support very large files. Applications that are compatible with HDFS are those that deal with large data sets. These applications write their data only once but they read it one or more times and require these reads to be satisfied at streaming speeds. HDFS supports write-once-read-many semantics on files. A typical block size used by HDFS is 128 MB. Thus, an HDFS file is chopped up into 128 MB chunks, and if possible, each chunk will reside on a different DataNode.

Staging

A client request to create a file does not reach the NameNode immediately. In fact, initially the HDFS client caches the file data into a local buffer. Application writes are transparently redirected to this local buffer. When the local file accumulates data worth over one chunk size, the client contacts the NameNode. The NameNode inserts the file name into the file system hierarchy and allocates a data block for it. The NameNode responds to the client request with the identity of the DataNode and the destination data block. Then the client flushes the chunk of data from the local buffer to the specified DataNode. When a file is closed, the remaining un-flushed data in the local buffer is transferred to the DataNode. The client then tells the NameNode that the file is closed. At this point, the NameNode commits the file creation operation into a persistent store. If the NameNode dies before the file is closed, the file is lost.

The above approach has been adopted after careful consideration of target applications that run on HDFS. These applications need streaming writes to files. If a client writes to a remote file directly without any client side buffering, the network speed and the congestion in the network impacts throughput considerably.

This approach is not without precedent. Earlier distributed file systems, e.g. AFS, have used client side caching to improve performance. A POSIX requirement has been relaxed to achieve higher performance of data uploads.

Replication Pipelining

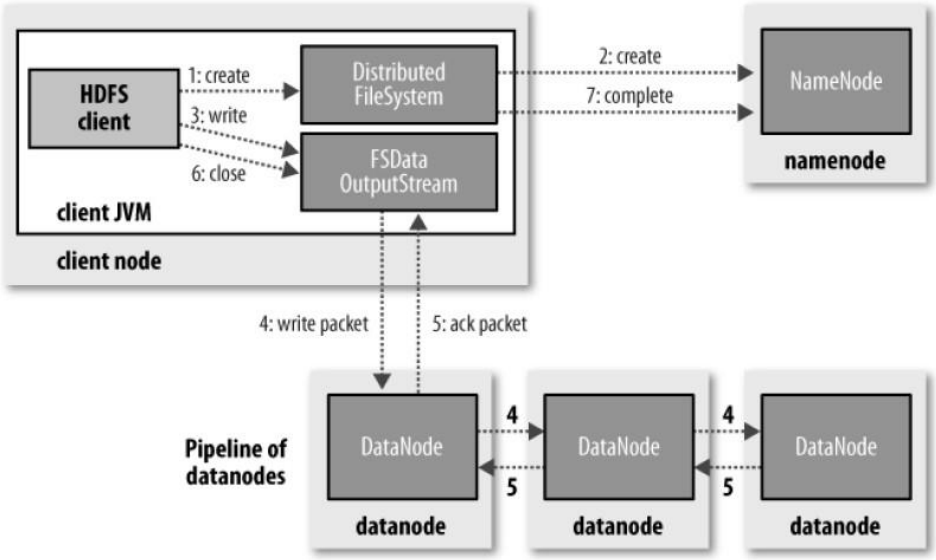
When a client is writing data to an HDFS file, its data is first written to a local buffer as explained in the previous section. Suppose the HDFS file has a replication factor of three. When the local buffer accumulates a chunk of user data, the client retrieves a list of DataNodes from the NameNode. This list contains the DataNodes that will host a replica of that block. The client then flushes the data chunk to the first DataNode. The first DataNode starts receiving the data in small portions, writes each portion to its local repository and transfers that portion to the second DataNode in the list. The second DataNode, in turn starts receiving each portion of the data block, writes that portion to its repository and then flushes that portion to the third DataNode. Finally, the third DataNode writes the data to its local repository. Thus, a DataNode can be receiving data from the previous one in the pipeline and at the same time forwarding data to the next one in the pipeline. Thus, the data is pipelined from one DataNode to the next.

Hadoop HDFS Data Write Operation

i) The HDFS client sends a **create** request on *DistributedFileSystem* APIs.

ii) *DistributedFileSystem* makes an RPC call to the namenode to create a new file in the file system's namespace. The namenode performs various checks to make sure that the file doesn't already exist and that the client has the permissions to create the file. When these checks pass, then only the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an *IOException.*

iii) The *DistributedFileSystem* returns a *FSDataOutputStream* for the client to start writing data to. As the client writes data, *DFSOutputStream* splits it into packets, which it writes to an internal queue, called the data queue. The data queue is consumed by the *DataStreamer, which*I is responsible for asking the namenode to allocate new **blocks** by picking a list of suitable datanodes to store the replicas.



iv) The list of datanodes form a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The *DataStreamer* streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline.

v) *DFSOutputStream* also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the *ack queue*. A packet is removed from the ack queue only when it has been acknowledged by the datanodes in the pipeline. Datanode sends the acknowledgment once required replicas are created (3 by default). Similarly, all the blocks are stored and replicated on the different datanodes, the data blocks are copied in parallel.

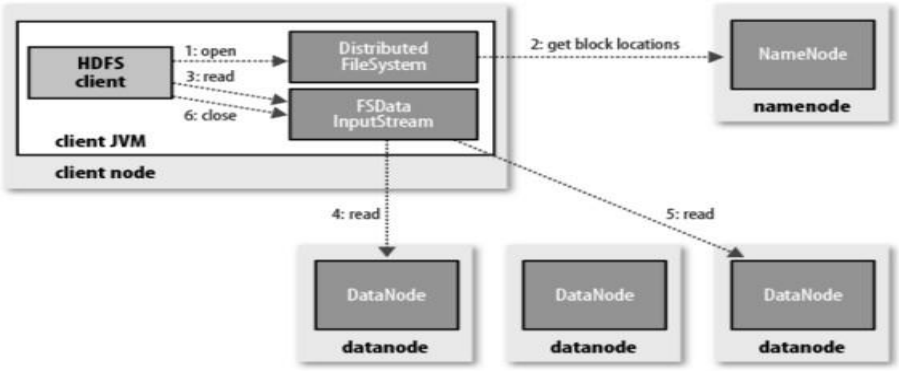
vi) When the client has finished writing data, it calls **close()** on the stream.

vii) This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete. The namenode already knows which blocks the file is made up of, so it only has to wait for blocks to be minimally replicated before returning successfully.

HDFS Data Read Operation

i) Client opens the file it wishes to read by calling **open()** on the *FileSystem* object, which for HDFS is an instance of *DistributedFileSystem*.

ii) *DistributedFileSystem* calls the namenode using RPC to determine the locations of the blocks for the first few blocks in the file. For each block, the namenode returns the addresses of the datanodes that have a copy of that block and datanode are sorted according to their proximity to the client.



iii) *DistributedFileSystem* returns a *FSDataInputStream* to the client for it to read data from. *FSDataInputStream*, thus, wraps the *DFSInputStream* which manages the datanode and namenode I/O. Client calls **read()** on the stream. DFSInputStream which has stored the datanode addresses then connects to the closest datanode for the first block in the file.

iv) Data is streamed from the datanode back to the client, as a result client can call **read()** repeatedly on the stream. When the block ends, DFSInputStream will close the connection to the datanode and then finds the best datanode for the next block.

v) If the *DFSInputStream* encounters an error while communicating with a datanode, it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The *DFSInputStream* also verifies checksums for the data transferred to it from the datanode. If it finds a corrupt block, it reports this to the namenode before the *DFSInputStream* attempts to read a replica of the block from another datanode.

vi) When the client has finished reading the data, it calls **close()** on the stream

Fault Tolerance in HDFS

As we have discussed HDFS data read and write operations in detail, Now, what happens when one of the machines part of the pipeline which has a datanode process running fails. Hadoop has an inbuilt functionality to handle this scenario (HDFS is fault tolerant). When a datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data.

- First, the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanode that are downstream from the failed node will not miss any packets.
- The current block on the good datanode is given a new identity, which is communicated to the namenode so that the partial block on the failed datanode will be deleted if the failed datanode recovery later on.
- The datanode that fails is removed from the pipeline, and then the remainder of the block's data is written to the two good datanodes in the pipeline.
- The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Then it treats the subsequent blocks as normal.

Unit-3

DEVELOPING DISTRIBUTED PROGRAMS AND ISSUES

Scalability: Scaling is one of the major issues of Developing Distributed Programs. The scaling issue consists of dimensions like communication capacity. The program should be designed such that the capacity may be increased with the increasing demand on the system.

Heterogeneity: It is an important design issue for the distributed programming. The communications infrastructure consists of channels of different capacities. End-Systems will possess a wide variety of presentation techniques.

Objects representation and translation: Selecting the best programming models for distributed objects like CORBA, Java etc. is an important issue.

Resource management: In Developing Distributed Programs, objects consisting of resources are located on different places. Routing is an issue at the network layer of the distributed system and at the application layer.

Security and privacy: How to apply the security policies to the interdependent system is a great issue in Developing Distributed Programs. Since Distributed Programs deal with sensitive data and information so the program must have a strong security and privacy measurement. Protection of distributed system assets, including base resources, storage, communications and user-interface I/O as well as higher-level composites of these resources, like processes, files, messages, display windows and more complex objects, are important issues in distributed program

Transparency: Transparency means up to what extent the distributed system should appear to the user as a single system? Distributed program must be designed to hide the complexity of the system to a greater extent.

Openness: Openness means up to what extent a system be designed using standard protocols to support Interoperability. It is desired for developers to add new features or replace subsystem in future. To accomplish this, distributed program must have well defined interfaces.

Quality of Service: How to specify the quality of service given to system users and acceptable level of quality of service delivered to the users. The quality of service is heavily dependent on the processes to be allocated to the processors in the system, resource distribution, hardware, adaptability of the program, network etc. A good performance, availability and reliability are required to reflect good quality of service.

Failure management: How can failure of the system be detected and repaired.

Synchronization: One of the most important issues that engineers of distributed programs are facing is synchronizing computations consisting of thousands of components. Current methods of synchronization like semaphores, monitors, barriers, remote procedure call, object method invocation, and message passing, do not scale well.

Resource identification: The resources are distributed across various computers and a proper naming scheme is to be designed for exact reference of the resources.

Communications: Distributed Systems have become more effective with the advent of Internet but there are certain requirements for performance, reliability etc. Effective approaches to communication should be used.

Software Architectures: It reflects the application functionality distributed over the logical components and across the processors. Selecting the right architecture for an application is required for better quality of service.

Performance analysis: The Performance analysis of Distributed software program is a great issue. It is expected that it should be high speed, fault tolerant and cost effective. It is also essential towards evaluating alternative design to meet the Quality of service. Moreover the ability to estimate the future performance of a large and complex distributed software program at design time can reduce the software cost and risk.

Generating a Test Data: Generating a test data to cover the respective test criteria for testing the component is a difficult task. It becomes more difficult in case of distributed program because the number of possible paths increases significantly. Test cases must cover the low level elements.

Component selection for testing: Testing distributed components require the services of other components. When a component is used with other there could be a possibility of deadlocks and race condition. There may be no error detected when only one client is used because only one thread is executed but in the case of multithreading the number of clients used for testing the components may detect the errors.

Test Sequence: The component is to be tested along with other components. What orders should be followed in testing components? If the components do not follow the layered architectural model, there could be chances of cycles among the components.

Testing for system scalability and performance: Scalability of conventional test criteria of data is a major issue in the context of testing. The concept of threading may be used in the components for improved performance while testing. But using multiple threads is a challenging task in testing.

Redundant testing during integration of component: components are first tested separately. When the entire program is tested, lots of retesting of component occurs.

Availability of source code: Software components may be developed in house or off- the-shelf. Depending upon the availability of source code various testing techniques are used for the system testing.

Heterogeneous languages, platform and Architecture: Different languages may be used for writing the components of the system. The components may be used on different hardware and software platform.

Monitoring and control mechanism in testing distributed software: Distributed software system involves multiple computers on the network. Testing monitoring and control mechanism in distributed environment is complex compared with centralized software system. Monitoring Distributed System services are also important for debugging during program development and required as part of the application itself like process control and automation.

Reproducibility of Events: Reproducibility of events is a challenging task because of concurrent processing and asynchronous communication occurring in the distributed environment. Moreover the lack of full control over the environment is another hurdle in this regards.

Deadlocks and Race Conditions: Deadlocks and race conditions are other great issues while developing distributed programs especially in the context of testing. It becomes more important issue especially in shared memory multiprocessor environment .

Testing for fault tolerance: The ability to tolerate faults in software system is required in applications like nuclear plant, Space missions, medical equipments etc. Testing for fault tolerance is challenging because the fault recovery code hardly gets executed while testing. Different fault injection techniques are used for fault tolerance by injecting faults in the program under test.

Scheduling issue for distributed program: Focuses on Scheduling problems in homogeneous and heterogeneous parallel distributed systems. The performance of distributed programs are affected by Broadcast/multicast processing and required to develop a delivering procedure that completes the processing in minimum time.

Controllability and Observability issues : Controllability and observability are two important issues in testing because they have an effect on the capability of the test system to check the conformance of an implementation

under test. Controllability is the capability of the Test System to force the Implementation under Test to receive inputs in a given order.

Distributed Task Allocation: Finding an optimal Task allocation in developing distributed program is a challenging job keeping in mind the concept of reliability and performance.

WHY MAP- REDUCE AND CONCEPTUAL UNDERSTANDING OF MAP-REDUCE PROGRAMMING

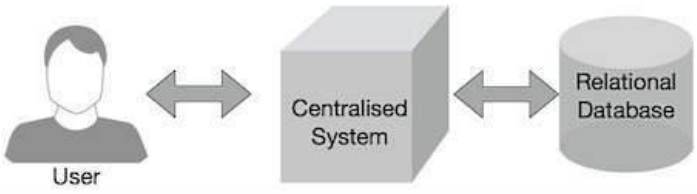
MapReduce is a programming model for writing applications that can process Big Data in parallel on multiple nodes. MapReduce provides analytical capabilities for analyzing huge volumes of complex data.

What is Big Data?

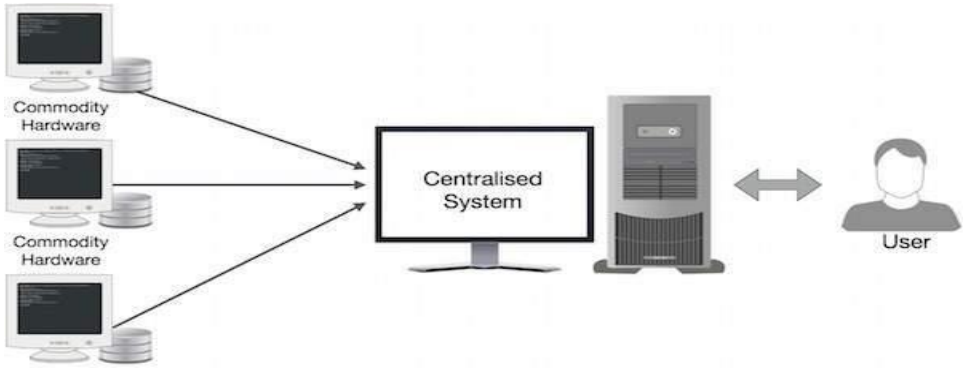
Big Data is a collection of large datasets that cannot be processed using traditional computing techniques. For example, the volume of data Facebook or YouTube need require it to collect and manage on a daily basis, can fall under the category of Big Data. However, Big Data is not only about scale and volume, it also involves one or more of the following aspects − Velocity, Variety, Volume, and Complexity.

Why MapReduce?

Traditional Enterprise Systems normally have a centralized server to store and process data. The following illustration depicts a schematic view of a traditional enterprise system. Traditional model is certainly not suitable to process huge volumes of scalable data and cannot be accommodated by standard database servers. Moreover, the centralized system creates too much of a bottleneck while processing multiple files simultaneously.



Google solved this bottleneck issue using an algorithm called MapReduce. MapReduce divides a task into small parts and assigns them to many computers. Later, the results are collected at one place and integrated to form the result dataset.



MapReduce programs work in two phases:

1. Map phase
2. Reduce phase.

Input to each phase are **key-value** pairs. In addition, every programmer needs to specify two functions: **map function** and **reduce function**.

The whole process goes through three phase of execution namely,

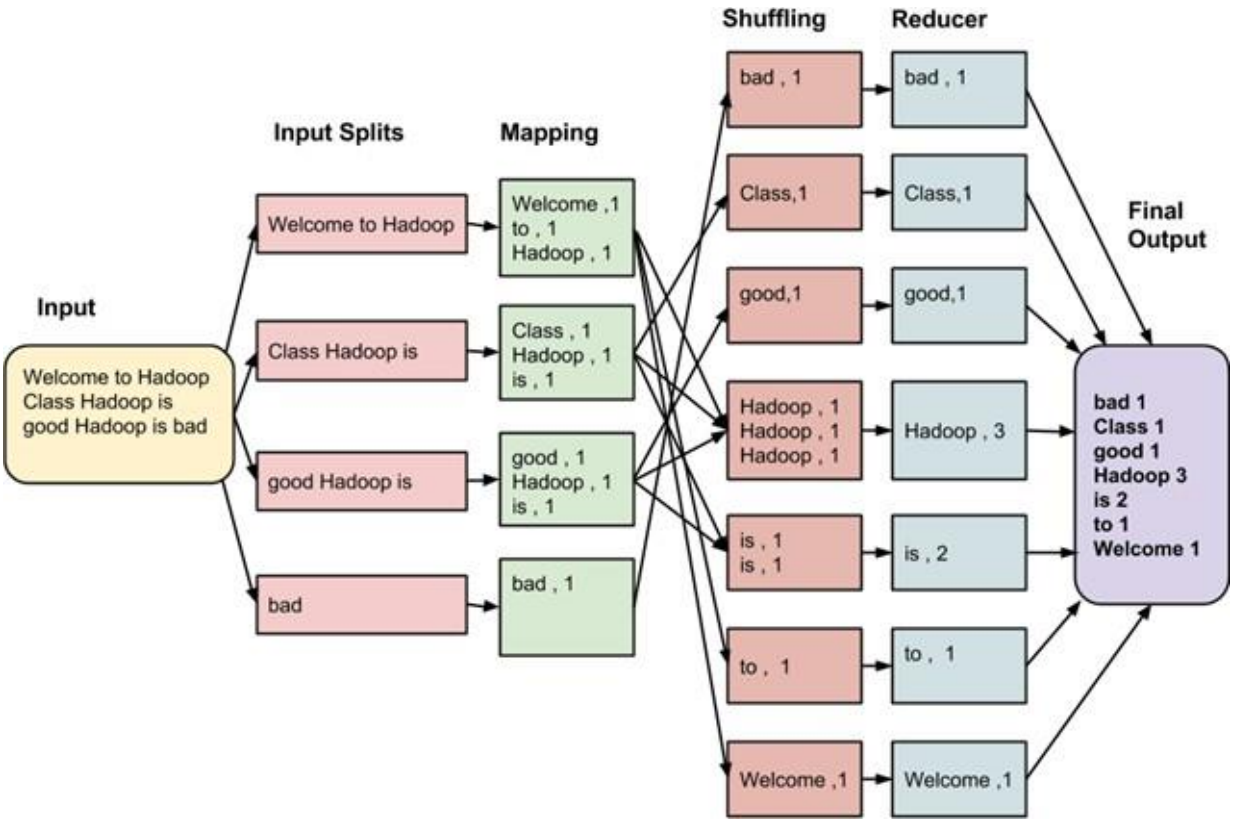
How MapReduce works

Lets understand this with an example –

Consider you have following input data for your MapReduce Program

Welcome to Hadoop Class Hadoop is good

Hadoop is bad



The final output of the MapReduce task is

bad	1
Class	1
good	1
Hadoop	3
is	2
to	1
Welcome	1

The data goes through following phases

Input Splits:

Input to a MapReduce job is divided into fixed-size pieces called **input splits** Input split is a chunk of the input that is consumed by a single map

Mapping

This is very first phase in the execution of map-reduce program. In this phase data in each split is passed to a mapping function to produce output values. In our example, job of mapping phase is to count number of occurrences of each word from input splits (more details about input-split is given below) and prepare a list in the form of <word, frequency>

Shuffling

This phase consumes output of Mapping phase. Its task is to consolidate the relevant records from Mapping phase output. In our example, same words are clubed together along with their respective frequency.

Reducing

In this phase, output values from Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset.

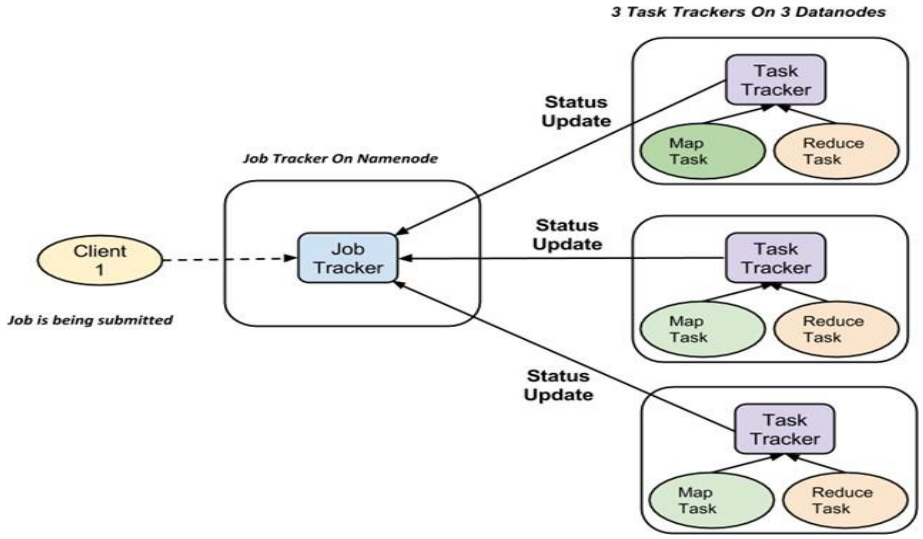
In our example, this phase aggregates the values from Shuffling phase i.e., calculates total occurrences of each words.

The overall process in detail

- One map task is created for each split which then executes map function for each record in the split.
- It is always beneficial to have multiple splits, because time taken to process a split is small as compared to the time taken for processing of the whole input. When the splits are smaller, the processing is better load balanced since we are processing the splits in parallel.
- However, it is also not desirable to have splits too small in size. When splits are too small, the overload of managing the splits and map task creation begins to dominate the total job execution time.
- For most jobs, it is better to make split size equal to the size of an HDFS block (which is 64 MB, by default).
- Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
- Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of HDFS store operation.
- Map output is intermediate output which is processed by reduce tasks to produce the final output.
- Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill.
- In the event of node failure before the map output is consumed by the reduce task, Hadoop

reruns the map task on another node and re-creates the map output.

- Reduce task don't work on the concept of data locality. Output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
- On this machine the output is merged and then passed to the user defined reduce function.
- Unlike to the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes). So, writing the reduce output



How MapReduce Organizes Work?

Hadoop divides the job into tasks. There are two types of tasks:

1. **Map tasks** (Spilts & Mapping)
2. **Reduce tasks** (Shuffling, Reducing) as mentioned above.

The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called a

1. **Jobtracker** : Acts like a **master** (responsible for complete execution of submitted job)
2. **Multiple Task Trackers** : Acts like **slaves,** each of them performing the job

For every job submitted for execution in the system, there is one **Jobtracker** that resides on **Namenode** and there are **multiple tasktrackers** which reside on **Datanode**.

- A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
- It is the responsibility of jobtracker to coordinate the activity by scheduling tasks to run on different data nodes.
- Execution of individual task is then look after by tasktracker, which resides on every data node executing part of the job.
- Tasktracker's responsibility is to send the progress report to the jobtracker.
- In addition, tasktracker periodically sends '**heartbeat**' signal to the Jobtracker so as to notify him of current state of the system.
- Thus jobtracker keeps track of overall progress of each job. In the event of task failure, the

jobtracker can reschedule it on a different tasktracker.

DEVELOPING MAP-REDUCE PROGRAMS IN JAVA

Given below is the data regarding the electrical consumption of an organization. It contains the monthly electrical consumption and the annual average for various years.

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Avg
1979	23	23	2	43	24	25	26	26	26	26	25	26	25
1980	26	27	28	28	28	30	31	31	31	30	30	30	29
1981	31	32	32	32	33	34	35	36	36	34	34	34	34
1984	39	38	39	39	39	41	42	43	40	39	38	38	40
1985	38	39	39	39	39	41	41	41	00	40	39	39	45

If the above data is given as input, we have to write applications to process it and produce results such as finding the year of maximum usage, year of minimum usage, and so on. This is a walkover for the programmers with finite number of records. They will simply write the logic to produce the required output, and pass the data to the application written.

But, think of the data representing the electrical consumption of all the largescale industries of a particular state, since its formation.

When we write applications to process such bulk data,

- They will take a lot of time to execute.
- There will be a heavy network traffic when we move data from source to network server and so on.

To solve these problems, we have the MapReduce framework.

Input Data

1979 23 23 2 43 24 25 26 26 26 26 25 26 25
1980 26 27 28 28 28 30 31 31 31 30 30 30 29
1981 31 32 32 32 33 34 35 36 36 34 34 34 34
1984 39 38 39 39 39 41 42 43 40 39 38 38 40
1985 38 39 39 39 39 41 41 41 00 40 39 39 45

Given below is the program to the sample data using MapReduce framework.

```
package hadoop; import java.util.*;
import java.io.IOException;
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
 import org.apache.hadoop.util.*;
public class ProcessUnits
{
```

//Mapper class

```
public static class E_EMapper extends MapReduceBase implements Mapper<LongWritable ,
Text,    /*Input value Type*/
Text,    /*Output key Type*/
IntWritable>       /*Output value Type*/
{
//Map function
public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
    Reporter reporter) throws IOException
{
String line = value.toString(); String lasttoken = null;
StringTokenizer s = new StringTokenizer(line,"\t");
String year = s.nextToken();
while(s.hasMoreTokens())
{
lasttoken=s.nextToken();
}
int avgprice = Integer.parseInt(lasttoken);
output.collect(new Text(year), new IntWritable(avgprice));
}
}
```

//Reducer class

```
public static class E_EReduce extends MapReduceBase implements Reducer< Text,
    IntWritable, Text, IntWritable >
{
//Reduce function
    public void reduce( Text key, Iterator <IntWritable> values, OutputCollector<Text,
    IntWritable> output, Reporter reporter) throws
IOException
{
int maxavg=30;
int val=Integer.MIN_VALUE;
while (values.hasNext())
{
if((val=values.next().get())>maxavg)
{
output.collect(key, new IntWritable(val));
}
}
```

```
}
}
//Main function
public static void main(String args[])throws Exception
{
JobConf conf = new JobConf(ProcessUnits.class);
conf.setJobName("max_eletricityunits");
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
conf.setMapperClass(E_EMapper.class);
conf.setCombinerClass(E_EReduce.class);
conf.setReducerClass(E_EReduce.class);
conf.setInputFormat(TextInputFormat.class);
 conf.setOutputFormat(TextOutputFormat.class);
FileInputFormat.setInputPaths(conf, new Path(args[0]));
 FileOutputFormat.setOutputPath(conf, new Path(args[1]));
JobClient.runJob(conf);
}
}
```

Save the above program as **ProcessUnits.java.** The compilation and execution of the program is explained below.

Compilation and Execution of Process Units Program

Let us assume we are in the home directory of a Hadoop user (e.g. /home/hadoop). Follow the steps given below to compile and execute the above program.

Step 1

The following command is to create a directory to store the compiled java classes.

$ mkdir units

Step 2

Download **Hadoop-core-2.6.5.jar,** which is used to compile and execute the MapReduce program. Visit the following link **http://www-us.apache.org/dist/hadoop/common/hadoop-2.6.5/hadoop-2.6.5.tar.gz** to download the jar. Let us assume the downloaded folder is **/home/hadoop/.**

Step 3

The following commands are used for compiling the **ProcessUnits.java** program and creating a jar for the program.

$ javac -classpath hadoop-core-1.2.1.jar -d units ProcessUnits.java

$ jar -cvf units.jar -C units/ .

Step 4

The following command is used to create an input directory in HDFS.

$HADOOP_HOME/bin/hadoop fs -mkdir input_dir

Step 5

The following command is used to copy the input file named **sample.txt**in the input directory of HDFS.

$HADOOP_HOME/bin/hadoop fs -put /home/hadoop/sample.txt input_dir

Step 6

The following command is used to verify the files in the input directory.

$HADOOP_HOME/bin/hadoop fs -ls input_dir/

Step 7

The following command is used to run the Eleunit_max application by taking the input files from the input directory.

$HADOOP_HOME/bin/hadoop jar units.jar hadoop.ProcessUnits input_dir output_dir

Wait for a while until the file is executed. After execution, as shown below, the output will contain the number of input splits, the number of Map tasks, the number of reducer tasks, etc.

```
INFO mapreduce.Job: Job
job_1414748220717_0002 completed
successfully
14/10/31 06:02:52
INFO mapreduce.Job:
Counters: 49 File System
Counters

FILE: Number of bytes read=61
FILE: Number of bytes
```

```
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2 Job Counters


          Launched map
          tasks=2 Launched
          reduce tasks=1 Data-
          local map tasks=2
          Total time spent by all maps in occupied slots
          (ms)=146137 Total time spent by all reduces in
          occupied slots (ms)=441 Total time spent by all
          map tasks (ms)=14613
          Total time spent by all reduce tasks
          (ms)=44120 Total vcore-seconds taken by
          all map tasks=146137

          Total vcore-seconds taken by all reduce tasks=44120
          Total megabyte-seconds taken by all map
          tasks=149644288 Total megabyte-seconds taken by
          all reduce tasks=45178880

        Map-Reduce

        Framework Map

        input records=5
          Map output
          records=5 Map
          output bytes=45
          Map output materialized
          bytes=67 Input split
          bytes=208
```

Step 8

The following command is used to verify the resultant files in the output folder.

$HADOOP_HOME/bin/hadoop fs -ls output_dir/

Step 9

The following command is used to see the output in **Part-00000** file. This file is generated by HDFS.

$HADOOP_HOME/bin/hadoop fs -cat output_dir/part-00000

Below is the output generated by the MapReduce program.

```
1981  34
1984  40
1985  45
```

Step 10

The following command is used to copy the output folder from HDFS to the local file system for analyzing.

$HADOOP_HOME/bin/hadoop fs -cat output_dir/part-00000/bin/hadoop dfs get output_dir /home/Hadoop

Important Commands

All Hadoop commands are invoked by the **$HADOOP_HOME/bin/hadoop** command.

Running the Hadoop script without any arguments prints the description for all commands.

Usage : hadoop [--config confdir] COMMAND

The following table lists the options available and their description.

Options	**Description**
namenode –format	Formats the DFS filesystem.
secondarynamenode	Runs the DFS secondary namenode.
namenode	Runs the DFS namenode.
datanode	Runs a DFS datanode.
dfsadmin	Runs a DFS admin client.
mradmin	Runs a Map-Reduce admin client.
fsck	Runs a DFS filesystem checking utility.
fs	Runs a generic filesystem user client.
balancer	Runs a cluster balancing utility.
oiv	Applies the offline fsimage viewer to an fsimage.
fetchdt	Fetches a delegation token from the NameNode.
jobtracker	Runs the MapReduce job Tracker node.
pipes	Runs a Pipes job.
tasktracker	Runs a MapReduce task Tracker node.
historyserver	Runs job history servers as a standalone daemon.
job	Manipulates the MapReduce jobs.
queue	Gets information regarding JobQueues.
version	Prints the version.
jar <jar>	Runs a jar file.
distcp <srcurl> <desturl>	Copies file or directories recursively.
distcp2 <srcurl> <desturl>	DistCp version 2.
archive -archiveName NAME -p <parent path> <src>* <dest>	Creates a hadoop archive.
Classpath	Prints the class path needed to get the Hadoop jar and the required libraries.
daemonlog	Get/Set the log level for each daemon

SETTING UP THE CLUSTER WITH HDFS AND UNDERSTANDING HOW MAP-REDUCE WORKS ON HDFS

a. Setting up SSH for a Hadoop cluster

The first step is to check whether SSH is installed on your nodes. We can easily do this by use of the "which" UNIX command:

```
[hadoop-user@master]$ which ssh
/usr/bin/ssh
[hadoop-user@master]$ which sshd
/usr/bin/sshd
```

[hadoop-user@master]$ which ssh-keygen

/usr/bin/ssh-keygen

If you instead receive an error message such as this,

/usr/bin/which: no ssh in (/usr/bin:/bin:/usr/sbin...

install OpenSSH (www.openssh.com) via a Linux package manager or by downloading the source directly. (Better yet, have your system administrator do it for you.)

Generate SSH key pair

Having verified that SSH is correctly installed on all nodes of the cluster, we use ssh-keygen on the master node to generate an RSA key pair. Be certain to avoid entering a passphrase, or you'll have to manually enter that phrase every time the master node attempts to access another node.

[hadoop-user@master]$ ssh-keygen -t rsa Generating public/private rsa key pair.

Enter file in which to save the key (/home/hadoop-user/.ssh/id_rsa): Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /home/hadoop-user/.ssh/id_rsa. Your public key has been saved in /home/hadoop-user/.ssh/id_rsa.pub.

After creating your key pair, your public key will be of the form

[hadoop-user@master]$ more /home/hadoop-user/.ssh/id_rsa.pub ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA1WS3RG8LrZH4zL2/1oYgkV1OmVclQ2OO5vRi0Nd51Sy3wWpBVHx82F3x3ddoZQjBK3uvLMaDhXvncJG31JPfU7CTAfmtgINYv0kdUbDJq4TKG/fuO5q9CqHV71thN2M310gcJ0Y9YCN6grmsiWb2iMcXpy2pqg8UM3ZKApyIPx99O1vREWm+4moFTgYwIl5be23ZCyxNjgZFWk5MRlT1p1TxB68jqNbPQtU7fIafS7Sasy7h4eyIy7cbLh8x0/V4/mcQsY5dvReitNvFVte6onl8YdmnMpAh6nwCvog3UeWWJjVZTEBFkTZuV1i9HeYHxpm1wAzcnf7az78jT IRQ== hadoop-user@master

and we next need to distribute this public key across your cluster.

Distribute public key and validate logins

Albeit a bit tedious, you'll next need to copy the public key to every slave node as well as the master node:

[hadoop-user@master]$ scp ~/.ssh/id_rsa.pub hadoop-user@target:~/master_key

Manually log in to the target node and set the master key as an authorized key (or append to the list of authorized keys if you have others defined).

[hadoop-user@target]$ mkdir ~/.ssh [hadoop-user@target]$ chmod 700 ~/.ssh

[hadoop-user@target]$ mv ~/master_key ~/.ssh/authorized_keys

[hadoop-user@target]$ chmod 600 ~/.ssh/authorized_keys

After generating the key, you can verify it's correctly defined by attempting to log in to the target node from the master:

[hadoop-user@master]$ ssh target

The authenticity of host 'target (xxx.xxx.xxx.xxx)' can't be established. RSA key fingerprint is

72:31:d8:1b:11:36:43:52:56:11:77:a4:ec:82:03:1d. Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'target' (RSA) to the list of known hosts. Last login: Sun Jan 4 15:32:22 2009 from master

After confirming the authenticity of a target node to the master node, you won't be prompted upon subsequent login attempts.

[hadoop-user@master]$ ssh target

Last login: Sun Jan 4 15:32:49 2009 from master

We've now set the groundwork for running Hadoop on your own cluster. Let's discuss the different Hadoop modes you might want to use for your projects.

Running Hadoop

We need to configure a few things before running Hadoop. Let's take a closer look at the Hadoop configuration directory:

[hadoop-user@master]$ cd $HADOOP_HOME [hadoop-user@master]$ ls -l conf/

total 100

-rw-rw-r-- 1 hadoop-user hadoop 2065 Dec 1 10:07 capacity-scheduler.xml

-rw-rw-r-- 1 hadoop-user hadoop 535 Dec 1 10:07 configuration.xsl

-rw-rw-r-- 1 hadoop-user hadoop 49456 Dec 1 10:07 hadoop-default.xml

-rwxrwxr-x 1 hadoop-user hadoop 2314 Jan 8 17:01 hadoop-env.sh

-rw-rw-r-- 1 hadoop-user hadoop 2234 Jan 2 15:29 hadoop-site.xml

-rw-rw-r-- 1 hadoop-user hadoop 2815 Dec 1 10:07 log4j.properties

-rw-rw-r-- 1 hadoop-user hadoop 28 Jan 2 15:29 masters

-rw-rw-r-- 1 hadoop-user hadoop 84 Jan 2 15:29 slaves

-rw-rw-r-- 1 hadoop-user hadoop 401 Dec 1 10:07 sslinfo.xml.example

The first thing you need to do is to specify the location of Java on all the nodes including the master. In hadoop-env.sh.

>$export JAVA_HOME=/usr/share/jdk

b. Operational Modes Of Hadoop

We have 3 operational modes for running Hadoop are,

1. *Local (standalone) mode*
2. *Pseudo-distributed mode*
3. *Fully distributed mode*

1. Local (standalone) mode

The standalone mode is the default mode for Hadoop. When you first uncompress the Hadoop source package, it's ignorant of your hardware setup. Hadoop chooses to be conservative and assumes a minimal configuration. All three XML files (or hadoop-site.xml before version 0.20) are empty under this default mode:

<?xml version="1.0"?>

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

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

<configuration>

</configuration>

With empty configuration files, Hadoop will run completely on the local machine. Because there's no need to communicate with other nodes, the standalone mode doesn't use HDFS, nor will it launch any of the Hadoop daemons. Its primary use is for developing and debugging the application logic of a MapReduce pro-gram without the additional complexity of interacting with the daemons.

2. ***Pseudo-distributed mode***

The pseudo-distributed mode is running Hadoop in a "cluster of one" with all daemons running on a single machine. This mode complements the standalone mode for debugging your code, allowing you to examine memory usage, HDFS input/out-put issues, and other daemon interactions. Listing 2.1 provides simple XML files to configure a single server in this mode.

core-site.xml

<?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://localhost:9000</value>

<description>The name of the default file system. A URI whose scheme and authority determine the FileSystem implementation. </description>

</property>

</configuration>

mapred-site.xml

<?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>localhost:9001</value>

<description>The host and port that the MapReduce job tracker runs at.</description>

</property>

</configuration>

hdfs-site.xml

<?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>1</value>
<description>The actual number of replications can be specified when the file is created.</description>
</property>
</configuration>

In core-site.xml and mapred-site.xml we specify the hostname and port of the NameNode and the JobTracker, respectively. In hdfs-site.xml we specify the default replication factor for HDFS, which should only be one because we're running on only one node. We must also specify the location of the Secondary NameNode in the mas-ters file and the slave nodes in the slaves file:

[hadoop-user@master]$ cat masters localhost

[hadoop-user@master]$ cat slaves localhost

While all the daemons are running on the same machine, they still communicate with each other using the same SSH protocol as if they were distributed over a cluster. For single-node operation simply check to see if your machine already allows you to ssh back to itself.

[hadoop-user@master]$ ssh localhost

If it does, then you're good. Otherwise setting up takes two lines. [hadoop-user@master]$ ssh-keygen -t dsa -P " -f ~/.ssh/id_dsa

[hadoop-user@master]$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys

You are almost ready to start Hadoop. But first you'll need to format your HDFS by using the command

[hadoop-user@master]$ bin/hadoop namenode -format

We can now launch the daemons by use of the start-all.sh script. The Java jps command will list all daemons to verify the setup was successful.

[hadoop-user@master]$ bin/start-all.sh [hadoop-user@master]$ jps

26893 Jps

26832 TaskTracker

26620 SecondaryNameNode

26333 NameNode

26484 DataNode

26703 JobTracker

3. *Fully distributed mode*

After continually emphasizing the benefits of distributed storage and distributed computation, it's time for us to set up a full cluster. In the discussion below we'll use the following server names:

- *master*—The master node of the cluster and host of the NameNode and JobTracker daemons
- *backup*—The server that hosts the Secondary NameNode daemon
- *hadoop1, hadoop2, hadoop3, ...*—The slave boxes of the cluster running both DataNode and TaskTracker daemons

Listing Example configuration files for fully

core-site.xml

```
<?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://master:9000</value>
 <description>The name of the default file system. A URI whose
 scheme and authority determine the FileSystem implementation.
 </description>
</property>

</configuration>
```

1 Locate NameNode for filesystem

mapred-site.xml

```
<?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>master:9001</value>
 <description>The host and port that the MapReduce job tracker runs
 at.</description>
</property>

</configuration>
```

2 Locate JobTracker master

hdfs-site.xml

```
<?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>
 <description>The actual number of replications can be specified when the
 file is created.</description>
</property>

</configuration>
```

3 Increase HDFS replication factor

The key differences are

- We explicitly stated the hostname for location of the NameNode **'1'** and JobTracker **'2'** daemons.

- We increased the HDFS replication factor to take advantage of distributed storage **'3'**. Recall that data is replicated across HDFS to increase availability and reliability.We also need to update the masters and slaves files to reflect the locations of the other daemons.

[hadoop-user@master]$ cat masters backup

[hadoop-user@master]$ cat slaves hadoop1

hadoop2 hadoop3

...

Once you have copied these files across all the nodes in your cluster, be sure to format HDFS to prepare it for storage:

[hadoop-user@master]$ bin/hadoop namenode-format

Now you can start the Hadoop daemons: [hadoop-user@master]$ bin/start-all.sh

and verify the nodes are running their assigned jobs.

[hadoop-user@master]$ jps 30879 JobTracker

30717 NameNode

30965 Jps

[hadoop-user@backup]$ jps 2099 Jps

1679 SecondaryNameNode [hadoop-user@hadoop1]$ jps 7101 TaskTracker

7617 Jps

6988 DataNode

You have a functioning cluster!

5. RUNNING SIMPLE WORD COUNT MAP-REDUCE PROGRAM ON THE CLUSTER

WordCount is a simple application that counts the number of occurrences of each word in a given input set. This works with a local-standalone, pseudo-distributed or fully-distributed Hadoop installation

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
 import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
importorg.apache.hadoop.mapreduce.lib.input.FileInputFormat;
 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class WordCount
```

```
{
public static class TokenizerMapper 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);
}
}
}
public static class IntSumReducer 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);
}
}
public static void main(String[] args) throws Exception
{
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
```

}

}

Usage

job.setReducerClass(IntSumReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

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

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

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

Assuming environment variables are set as follows:

```
export JAVA_HOME=/usr/java/default
export PATH=${JAVA_HOME}/bin:${PATH}
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

Compile WordCount.java and create a jar:

```
$ bin/hadoop com.sun.tools.javac.Main WordCount.java
$ jar cf wc.jar WordCount*.class
```

Assuming that:

- /user/joe/wordcount/input - input directory in HDFS
- /user/joe/wordcount/output - output directory in HDFS Sample text-files as input:

```
$ bin/hadoop fs -ls /user/joe/wordcount/input/ /user/joe/wordcount/input/file01
/user/joe/wordcount/input/file02

$ bin/hadoop fs -cat
/user/joe/wordcount/input/file01 Hello World
Bye World

$ bin/hadoop fs -cat
```

Run the application:

```
$ bin/hadoop jar wc.jar WordCount /user/joe/wordcount/input
/user/joe/wordcount/output
```

Output:

```
$ bin/hadoop fs -cat
/user/joe/wordcount/output/part-r-00000` Bye 1
Goodbye 1
Hadoop 2
Hello 2
World
2`
```

Walk-through

The WordCount application is quite straight-forward.

```
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);
```

The Mapper implementation, via the map method, processes one line at a time, as provided by the specified TextInputFormat. It then splits the line into tokens separated by whitespaces, via the StringTokenizer, and emits a key-value pair of < <word>, 1>.

For the given sample input the first map emits:

```
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
```

The second map emits:

```
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>
```

We'll learn more about the number of maps spawned for a given job, and how to control them in a fine-grained manner, a bit later in the tutorial.

job.setCombinerClass(IntSumReducer.class);

WordCount also specifies a combiner. Hence, the output of each map is passed through the local combiner (which is same as the Reducer as per the job configuration) for local aggregation, after being sorted on the *key*s.

The output of the first map:

```
< Bye, 1>
< Hello, 1>
< World, 2>`
```

The output of the second map:

```
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>`
```

```
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);
```

The Reducer implementation, via the reduce method just sums up the values, which are the occurrence counts for each key (i.e. words in this example).

Thus the output of the job is:

```
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>`
```

The main method specifies various facets of the job, such as the input/output paths (passed via the command line), key/value types, input/output formats etc., in the Job. It then calls the job.waitForCompletion to submit the job and monitor its progress.

6. ADDITIONAL EXAMPLES OF M-R PROGRAMMING.

Problem statement: I run a highly busy website and need to pull down my site for an hour in order to apply some patches and maintenance of backend severs, which means the website will be completely unavailable for an hour. To perform this activity the primary lookout will be that shutdown outage should be affected to least number of users. The games starts here: We need to identify at what hour of the day the web traffic is least for the website so that maintenance activity can be scheduled for that time.

There is an Apache web server log for each day which records the activities happening on website. But those are huge files up to 5 GB each.

Excerpt from Log file:

64.242.88.10 – – [07/Mar/2014:22:12:28 -0800] "GET /twiki/bin/attach/TWiki/WebSearch

HTTP/1.1" 401 12846

64.242.88.10 – – [07/Mar/2014:22:15:57 -0800] "GET /mailman/listinfo/hs_rcafaculty HTTP/1.1" 200 6345

We are interested only in the date field i.e. [07/Mar/2014:22:12:28 -0800]

Solution: I need to consume log files of one month and run my MapReduce code which calculates the total number of hits for each hour of the day. Hour which has the least number of hits is perfect for the downtime. It is as simple as that!

A MapReduce program usually consists of the following 3 parts:

1. Mapper
2. Reducer
3. Driver

As the name itself states Map and Reduce, the code is divided basically into two phases one is Map and second is Reduce. Both phase has an input and output as key-value pairs. Programmer has been given the liberty to choose the data model for the input and output for Map and Reduce both. Depending upon the business problem we need to use the appropriate data model.

What Mappers does?

- The Map function reads the input files as key/value pairs, processes each, and generates zero or more output key/value pairs.
- The Map class extends Mapper class which is a subclass of org.apache.hadoop.mapreduce.
- java.lang.Object : org.apache.hadoop.mapreduce.Mapper
- The input and output types of the map can be (and often are) different from each other.
- If the application is doing a word count, the map function would break the line into words and output a key/value pair for each word. Each output pair would contain the word as the key and the number of instances of that word in the line as the value.
- The Map function is also a good place to filter any unwanted fields/ data from input file, we take the data only we are interested to remove unnecessary workload.

I have used Hadoop 1.2.1 API, Java 1.7 to write this program.

```
package com.balajitk.loganalyzer;
 import java.io.IOException;
 import java.text.ParseException;
 import java.util.regex.Matcher;
 import java.util.regex.Pattern;
 import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
```

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.slf4j.Logger; 12import org.slf4j.LoggerFactory;
import com.balajitk.loganalyzer.ParseLog; 14
public class LogMapper extends
}
Mapper&lt;LongWritable, Text, IntWritable, IntWritable&gt; {

private static Logger logger = LoggerFactory.getLogger(LogMapper.class);
 private IntWritable hour = new IntWritable();
private final static IntWritable one = new IntWritable(1);
private static Pattern logPattern = Pattern.compile("([^ ]*) ([^ ]*) ([^ ]*) \\[([^]]*)\\]"+ "
                                    \"([^\"]*)\""+ " ([^ ]*) ([^ ]*).*");
    public void map(LongWritable key, Text value, Context context) throws
    InterruptedException, IOException {
logger.info("Mapper started");
String line = ((Text) value).toString();
Matcher matcher = logPattern.matcher(line);
 if (matcher.matches()) {
String timestamp = matcher.group(4);
try {
hour.set(ParseLog.getHour(timestamp));
    }
    catch (ParseException e)
    {
     logger.warn("Exception", e);
}
context.write(hour, one);
}
logger.info("Mapper Completed");
}
```

The Mapper code which is written above is written for processing single record from programmer's point of view. We will never write logic in MapReduce to deal with entire data set. The framework is responsible to convert the code to process entire data set by converting into desired key value pair.

The Mapper class has four parameters that specifies the input key, input value, output key, and output values of the Map function.

1Mapper<LongWritable, Text, IntWritable, IntWritable> 1Mapper<Input key, Input value, Output key, and Output values>

1Mapper<Offset of the input file, Single Line of the file, Hour of the day, Integer One>

Hadoop provides its own set of basic types that are optimized for network serialization which can be found in the org.apache.hadoop.io package.

In my program I have used LongWritable, which corresponds to a Java Long, Text (like Java String), and IntWritable (like Java Integer). Mapper write their output using instance of Context class which is used to communicate in Hadoop.

What Reducer does?

1. The Reducer code reads the outputs generated by the different mappers as pairs and emits key value pairs.
2. Reducer reduces a set of intermediate values which share a key to a smaller set of values.
3. java.lang.Object : org.apache.hadoop.mapreduce.Reducer
4. Reducer has 3 primary phases: shuffle, sort and reduce.
5. Each reduce function processes the intermediate values for a particular key generated by the map function. There exists a one-one mapping between keys and reducers.
6. Multiple reducers run in parallel, as they are independent of one another. The number of reducers for a job is decided by the programmer. By default, the number of reducers is 1.
7. The output of the reduce task is typically written to the FileSystem via OutputCollector.collect(WritableComparable, Writable)

```
package com.balajitk.loganalyzer;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Reducer;
import org.slf4j.Logger;
```

```
        import org.slf4j.LoggerFactory;
 public class LogReducer extends
}
    Reducer&lt;IntWritable, IntWritable, IntWritable, IntWritable&gt; { private static Logger
    logger = LoggerFactory.getLogger(LogReducer.class);
    public void reduce(IntWritable key, Iterable&lt;IntWritable&gt; values, Context context)
    throws IOException, InterruptedException {
logger.info("Reducer started");
 int sum = 0;
    for (IntWritable value : values) {
    sum = sum + value.get();
}
context.write(key, new IntWritable(sum));
logger.info("Reducer completed");

}
```

Four parameters are used in Reducers to specify input and output, which define the types of the input and output key/value pairs. Output of the map task will be input to reduce task. First two parameter are the input key value pair from map task. In our example IntWritable, IntWritable

1Reducer<IntWritable, IntWritable, IntWritable, IntWritable> 1Reducer<Input key, Input value, Output key, and Output values> 1Reducer<Hour of the day, List of counts, Hour, Total Count for the Hour>;

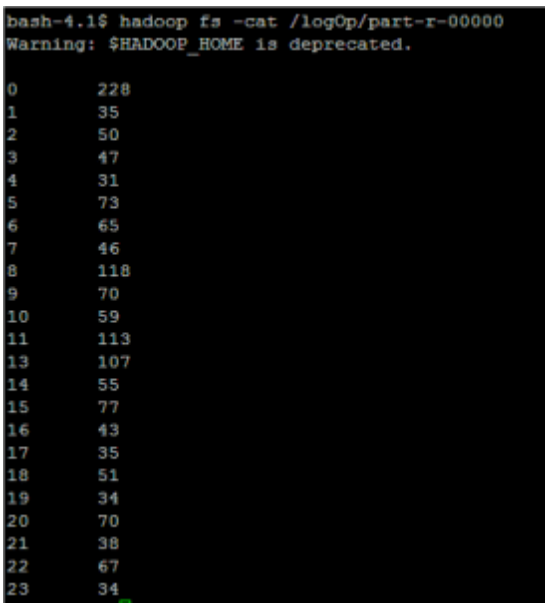
What Driver does?

Driver class is responsible to execute the MapReduce framework. Job object allows you to configure the Mapper, Reducer, InputFormat, OutputFormat etc.

```
package com.balajitk.loganalyzer;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
public class LogDriver {
    private static Logger logger = LoggerFactory.getLogger(LogDriver.class);
        public static void main(String[] args) throws Exception {
        logger.info("Code started");
         Job job = new Job();
         job.setJarByClass(LogDriver.class);
         job.setJobName("Log Analyzer");
         job.setMapperClass(LogMapper.class);
         job.setReducerClass(LogReducer.class);
         job.setOutputKeyClass(IntWritable.class);
         job.setOutputValueClass(IntWritable.class);
         FileInputFormat.addInputPath(job, new Path(args[0]));
         FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.waitForCompletion(true); logger.info("Code ended");
}
```

Job control is performed through the Job class in the new API, rather than the old JobClient, which no longer exists in the new API.

Output:



Unit-4: Anatomy of Map-Reduce Jobs

- In releases of Hadoop up to and including the 0.20 release series, mapred.job.tracker determines the means of execution.
- In Hadoop 0.23.0 a new MapReduce implementation was introduced. The new implementation (called MapReduce 2) is built on a system called YARN, described in "YARN (MapReduce 2)".
- For now, the framework that is used for execution is set by the mapreduce.framework.name property, which takes the values local (for the local job runner), classic (for the "classic" MapReduce framework, also called MapReduce 1, which uses a jobtracker and tasktrackers), and yarn (for the new framework).

1. Classic MapReduce (MapReduce 1)

A job run in classic MapReduce is illustrated in Figure 5-1. At the highest level, there are **four** independent entities:

- The **client**, which submits the MapReduce job.
- The **jobtracker**, which coordinates the job run. The jobtracker is a Java application whose main class is JobTracker.
- The **tasktrackers**, which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is TaskTracker.
- The **distributed filesystem** (normally HDFS), which is used for sharing job files between the other entities.

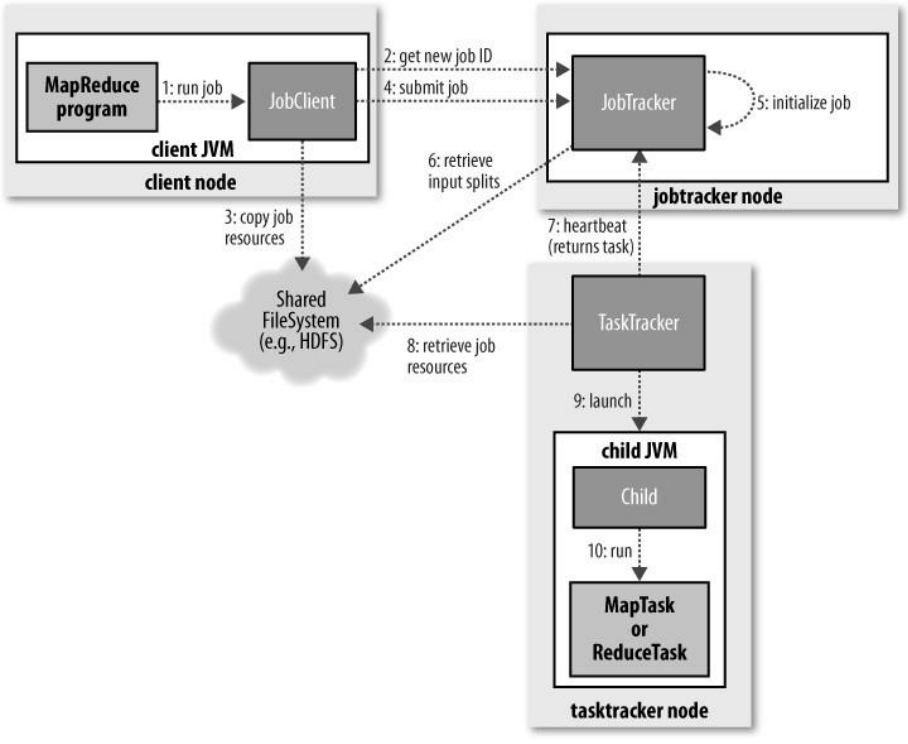


Figure 5-1. How Hadoop runs a MapReduce job using the classic framework

Job Submission

The submit() method on Job creates an internal JobSummitter instance and calls submitJobInternal() on it (step1 in Figure 5-1). Having submitted the job, waitForCompletion() polls the job's progress once a second and reports the progress to the console if it has changed since the last report. When the job is complete, if it was successful, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented by **JobSummitter** does the following:

- Asks the **jobtracker** for a new job ID (by calling **getNewJobId**() on **JobTracker**) (**step 2**).
- Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.
- Computes the input splits for the job. If the splits cannot be computed, because the input paths don't exist, for example, then the job is not submitted and an error is thrown to the MapReduce program.
- Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the jobtracker's filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the mapred.submit.replication property, which defaults to 10) so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job (**step 3**).
- Tells the jobtracker that the job is ready for execution (by calling **submitJob()** on **JobTracker**) (**step4**).

Job Initialization

- When the JobTracker receives a call to its **submitJob**() method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it. Initialization involves creating an object to represent the job being run, which encapsulates its tasks, and bookkeeping information to keep track of the tasks' status and progress (**step 5**).
- To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the client from the shared filesystem (**step 6**).
- It then creates one map task for each split. The number of reduce tasks to create is determined by the mapred.reduce.tasks property in the Job, which is set by the **setNumReduceTasks**() method, and the scheduler simply creates this number of reduce tasks to be run. Tasks are given IDs at this point.

- In addition to the map and reduce tasks, two further tasks are created: a **job setup** task and a **job cleanup** task. These are run by tasktrackers and are used to run code to setup the job before any map tasks run, and to cleanup after all the reduce tasks are complete.
- The OutputCommitter that is configured for the job determines the code to be run, and by default this is a FileOutputCommitter. For the job setup task it will create the final output directory for the job and the temporary working space for the task output, and for the job cleanup task it will delete the temporary working space for the task output.

Task Assignment

- Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker. Heartbeats tell the jobtracker that a tasktracker is alive, but they also double as a channel for messages.
- As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the jobtracker will allocate it a task, which it communicates to the tasktracker using the heartbeat return value (**step 7**).
- Before it can choose a task for the tasktracker, the jobtracker must choose a job to select the task from. Job Scheduler simply maintains a priority list of jobs. Having chosen a job, the jobtracker now chooses a task for the job.
- Tasktrackers have a fixed number of slots for map tasks and for reduce tasks: for example, a tasktracker may be able to run two map tasks and two reduce tasks simultaneously.
- (The precise number depends on **the number of cores and the amount of memory** on the tasktracker; The default scheduler fills empty map task slots before reduce task slots, so if the tasktracker has at least one empty map task slot, the jobtracker will select a map task; otherwise, it will select a reduce task.
- To choose a reduce task, the jobtracker simply takes the next in its list of yet-to-be-run reduce tasks, since there are no data locality considerations.
- For a map task, however, it takes account of the tasktracker's network location and picks a task whose input split is as close as possible to the tasktracker.
- In the optimal case, the task is *data-local*, that is, running on the same node that the split resides on. Alternatively, the task may be *rack-local*: on the same rack, but not the same node, as the split. Some tasks are neither data-local nor rack-local and retrieve their data from a different rack from the one they are running on. You can tell the proportion of each type of task by looking at a job's counters

Task Execution

- Now that the tasktracker has been assigned a task, the next step is for it to run the task.
- First, it localizes the job JAR by copying it from the shared filesystem to the

tasktracker's filesystem. It also copies any files needed from the distributed cache by the application to the local disk; (**step 8**).

- Second, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory.
- Third, it creates an instance of TaskRunner to run the task.
- TaskRunner launches a new Java Virtual Machine (**step 9)** to run each task in (**step 10**), so that any bugs in the user-defined map and reduce functions don't affect the tasktracker (by causing it to crash or hang, for example).
- The child process communicates with its parent through the *umbilical* interface. This way it informs the parent of the task's progress every few seconds until the task is complete.
- Each task can perform setup and cleanup actions, which are run in the same JVM as the task itself, and are determined by the OutputCommitter for the job.
- The cleanup action is used to commit the task, which in the case of file-based jobs means that its output is written to the final location for that task.
- The commit protocol ensures that when speculative execution is enabled, only one of the duplicate tasks is committed and the other is aborted.

Streaming and Pipes

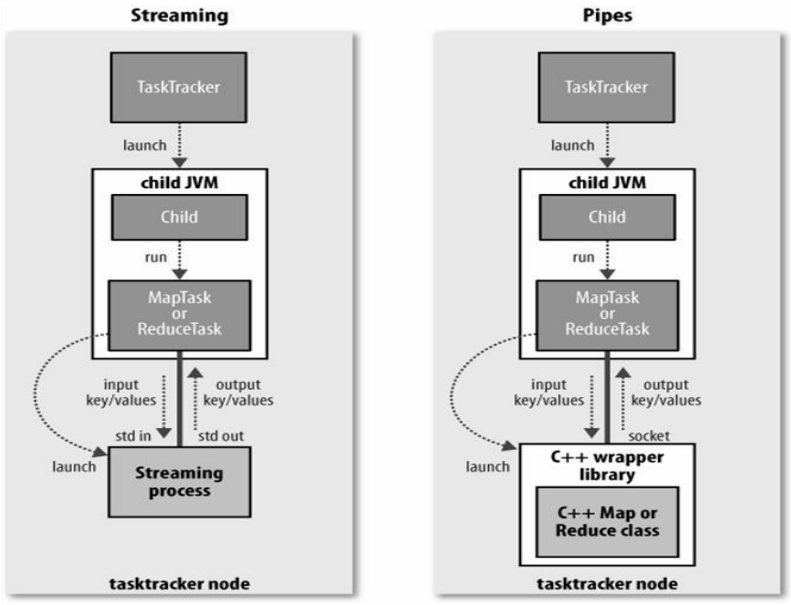


Figure 5-2. The relationship of the Streaming and Pipes executable to the tasktracker and its child

Both Streaming and Pipes run special map and reduce tasks for the purpose of launching the user-supplied **executable** and **communicating** with it (Figure 5-2).

- In the case of Streaming, the Streaming task communicates with the process (which may

be written in any language) using standard input and output streams.

- The Pipes task, on the other hand, listens on a socket and passes the C++ process a port number in its environment, so that on startup, the C++ process can establish a persistent **socket** connection back to the parent Java Pipes task.

Progress and Status Updates

- MapReduce jobs are long-running batch jobs, taking anything from minutes to hours to run.
- A job and each of its tasks have a *status*, which includes such things as the state of the job or task (e.g., running, successfully completed, failed), the progress of maps and reduces, the values of the job's counters, and a status message or description (which may be set by user code).
- These statuses change over the course of the job, so how do they get communicated back to the client?
- When a task is running, it keeps track of its *progress*, that is, the proportion of the task completed.
- If a task reports progress, it sets a flag to indicate that the status change should be sent to the tasktracker.
- The flag is checked in a separate thread every three seconds, and if set it notifies the tasktracker of the current task status. Meanwhile, the tasktracker is sending heartbeats to the jobtracker longer), and the status of all the tasks being run by the tasktracker is sent in the call.
- The jobtracker combines these updates to produce a global view of the status of all the jobs being run and their constituent tasks.
- Finally, as mentioned earlier, the Job receives the latest status by polling the jobtracker every second. Clients can also use Job's **getStatus**() method to obtain a **JobStatus** instance, which contains all of the status information for the job.
- The method calls are illustrated in **Figure 5-**3

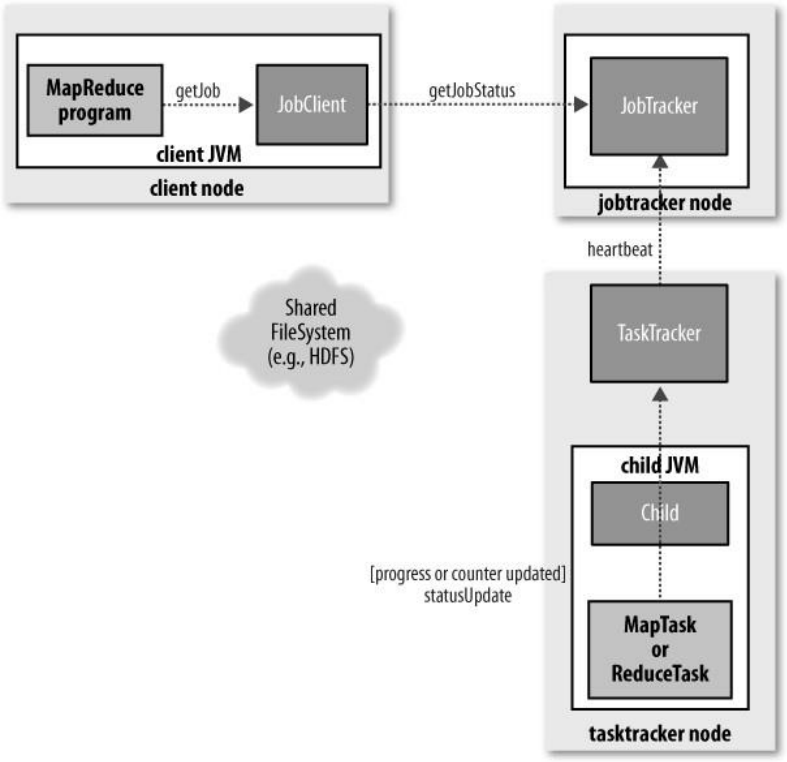


Figure 5-3. How status updates are propagated through the MapReduce 1 system

Job Completion

- When the jobtracker receives a notification that the last task for a job is complete, it changes the status for the job to "successful."
- Then, when the Job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the waitForCompletion() method.
- The jobtracker also sends an **HTTP** job notification if it is configured to do so. This can be configured by clients wishing to receive callbacks, via the job.end.notification.url property.
- Last, the jobtracker cleans up its working state for the job and instructs tasktrackers to do the same

2. YARN (MapReduce 2)

For very large clusters in the region of 4000 nodes and higher, the MapReduce system described in the previous section begins to hit scalability bottlenecks, so in 2010 a group at Yahoo! began to design the next generation of MapReduce. The result was *YARN*, short for Yet Another Resource Negotiator (or if you prefer recursive ancronyms, YARN Application Resource Negotiator).

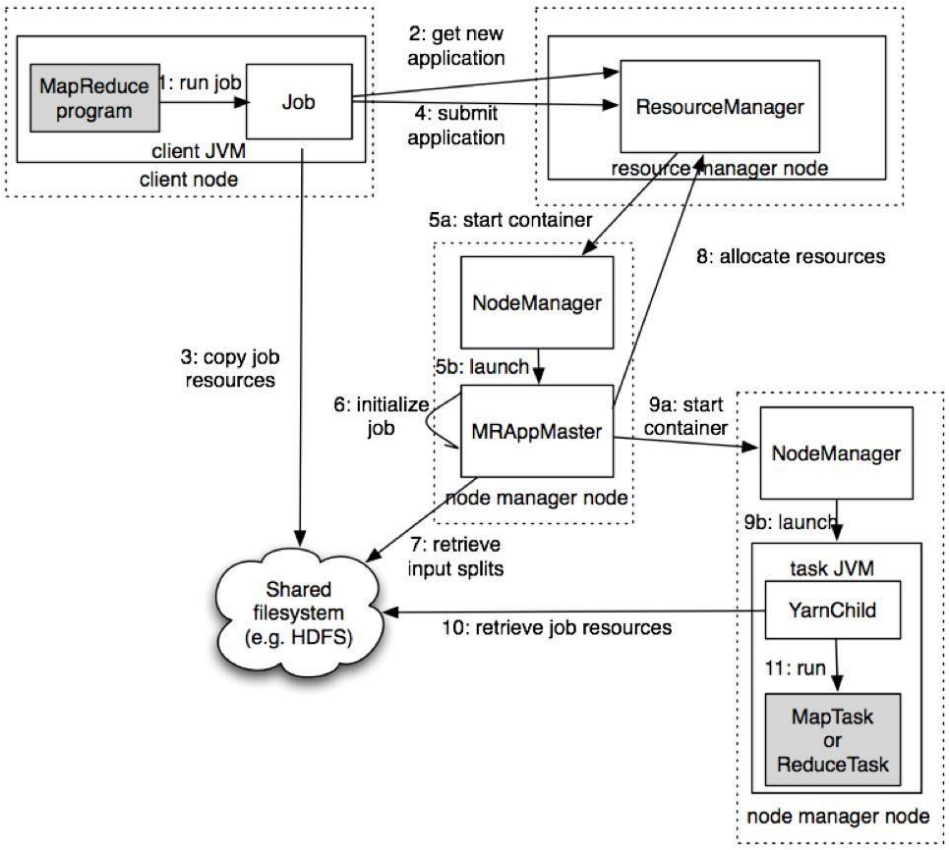


Figure 5-4. How Hadoop runs a MapReduce job using YARN

You can run a MapReduce job with a single method call: submit() on a Job object (you can also call waitForCompletion(), which submits the job if it hasn't been submitted already, then waits for it to finish).

This method call conceals a great deal of processing behind the scenes. The whole process is illustrated in Figure 5-4.

At the highest level, there are five independent entities

1. The **client**, which submits the MapReduce job.
2. The YARN **resource manager**, which coordinates the allocation of compute resources on the cluster.
3. The YARN **node managers**, which launch and monitor the compute containers on machines in the cluster.
4. The MapReduce **application master**, which coordinates the tasks running the MapReduce job. The application master and the MapReduce tasks run in containers that are scheduled by the resource manager and managed by the node managers.
5. The distributed filesystem (normally HDFS), which is used for sharing job files between the other entities.

Job Submission

The submit() method on Job creates an internal JobSubmitter instance and calls submitJobInternal() on it (step 1 in Figure 5-4). Having submitted the job, waitForCompletion() polls the job's progress once per second and reports the progress to the console if it has changed since the last report.

When the job completes successfully, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented by JobSubmitter does the following:

- Asks the resource manager for a new application ID, used for the MapReduce job ID (**step 2**).
- Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.
- Computes the input splits for the job. If the splits cannot be computed (because the input paths don't exist, for example), the job is not submitted and an error is thrown to the MapReduce program.
- Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the shared filesystem in a directory named after the job ID (**step 3**).
- The job JAR is copied with a high replication factor (controlled by the mapreduce.client.submit.file.replication property, which defaults to 10) so that there are lots of copies across the cluster for the node managers to access when they run tasks for the job.
- Submits the job by calling **submitApplication**() on the resource manager (**step 4**).

Job Initialization

- When the resource manager receives a call to its submitApplication() method, it hands off the request to the YARN scheduler. The scheduler allocates a container, and the resource manager then launches the application master's process there, under the node manager's management (**steps 5a and 5b**).
- The application master for MapReduce jobs is a Java application whose main class is MRAppMaster. It initializes the job by creating a number of bookkeeping objects to keep track of the job's progress, as it will receive progress and completion reports from the tasks (**step 6**).
- Next, it retrieves the input splits computed in the client from the shared filesystem (**step 7**). It then creates a map task object for each split, as well as a number of reduce task

objects determined by the mapreduce.job.reduces property (set by the setNumReduceTasks() method on Job). Tasks are given IDs at this point.

- The application master decides if the job is small, the application master may choose to run the tasks in the same JVM as itself. This happens when it judges that the overhead of allocating and running tasks in new containers outweighs the gain to be had in running them in parallel, compared to running them sequentially on one node. Such a job is said to be ***uberized***, or run as an ***uber task***.
- What qualifies as a small job? By default, a small job is one that has less than 10 mappers, only one reducer, and an input size that is less than the size of one HDFS block.

- Finally, before any tasks can be run, the application master calls the setupJob() method on the OutputCommitter. For FileOutputCommitter, which is the default, it will create the final output directory for the job and the temporary working space for the task output.

Task Assignment

- If the job does not qualify for running as an uber task, then the application master requests containers for all the map and reduce tasks in the job from the resource manager (**step 8**).

- Requests for map tasks are made first and with a higher priority than those for reduce tasks, since all the map tasks must complete before the sort phase of the reduce can start.

- Requests for reduce tasks are not made until 5% of map tasks have completed.

- Reduce tasks can run anywhere in the cluster, but requests for map tasks have data locality constraints that the scheduler tries to honor. In the optimal case, the task is *data local*—that is, running on the same node that the split resides on. Alternatively, the task may be *rack local*: on the same rack, but not

the same node, as the split. Some tasks are neither data local nor rack local and retrieve their data from a different rack than the one they are running on. For a particular job run, you can determine the number of tasks that ran at each locality level by looking at the job's counters.

- Requests also specify memory requirements and CPUs for tasks. By default, each map and reduce task is allocated 1,024 MB of memory and one virtual core. The values are configurable on a per-job basis via the following properties: mapreduce.map.memory.mb, mapreduce.reduce.memory.mb, mapreduce.map.cpu.vcores and mapreduce.reduce.cpu.vcores.

Task Execution

- Once a task has been assigned resources for a container on a particular node by the resource manager's scheduler, the application master starts the container by contacting the node manager (steps 9a and 9b).

- The task is executed by a Java application whose main class is YarnChild. Before it can run the task, it localizes the resources that the task needs, including the job configuration and JAR file, and any files from the distributed cache (**step 10**).

- Finally, it runs the map or reduce task (**step 11**).

- The YarnChild runs in a dedicated JVM, so that any bugs in the user-defined map and reduce functions (or even in YarnChild) don't affect the node manager—by causing it to crash or hang, for example.

- Each task can perform setup and commit actions, which are run in the same JVM as the task itself and are determined by the OutputCommitter for the job.

- For file-based jobs, the commit action moves the task output from a temporary location to its final location. The commit protocol ensures that when speculative execution is enabled, only one of the duplicate tasks is committed and the other is aborted.

Streaming

Streaming runs special map and reduce tasks for the purpose of launching the usersupplied executable and communicating with it (Figure 5-5).

The Streaming task communicates with the process (which may be written in any language) using standard input and output streams. During execution of the task, the Java process passes input key-value pairs to the external process, which runs it through the user-defined map or reduce function and passes the output key-value pairs back to the Java process. From the node manager's point of view, it is as if the child process ran the map or reduce code itself.

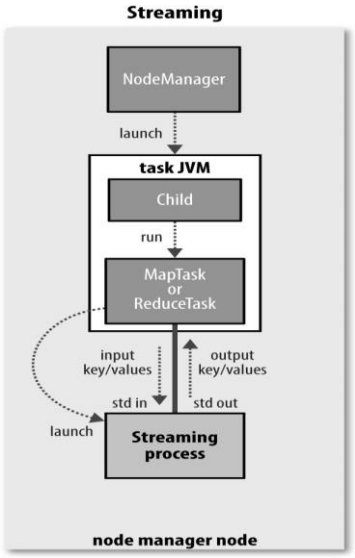


Figure 5-5. The relationship of the Streaming executable to the node manager and the task container

Progress and Status Updates

A job and each of its tasks have a *status*, which includes such things as the state of the job or task (e.g., running, successfully completed, failed), the progress of maps and reduces, the values of the job's counters, and a status message or description (which may be set by user code).

These statuses change over the course of the job, so how do they get communicated back to the client? When a task is running, it keeps track of its *progress* (i.e., the proportion of the task completed). For map tasks, this is the proportion of the input that has been processed. For reduce tasks the proportion of the reduce input processed.

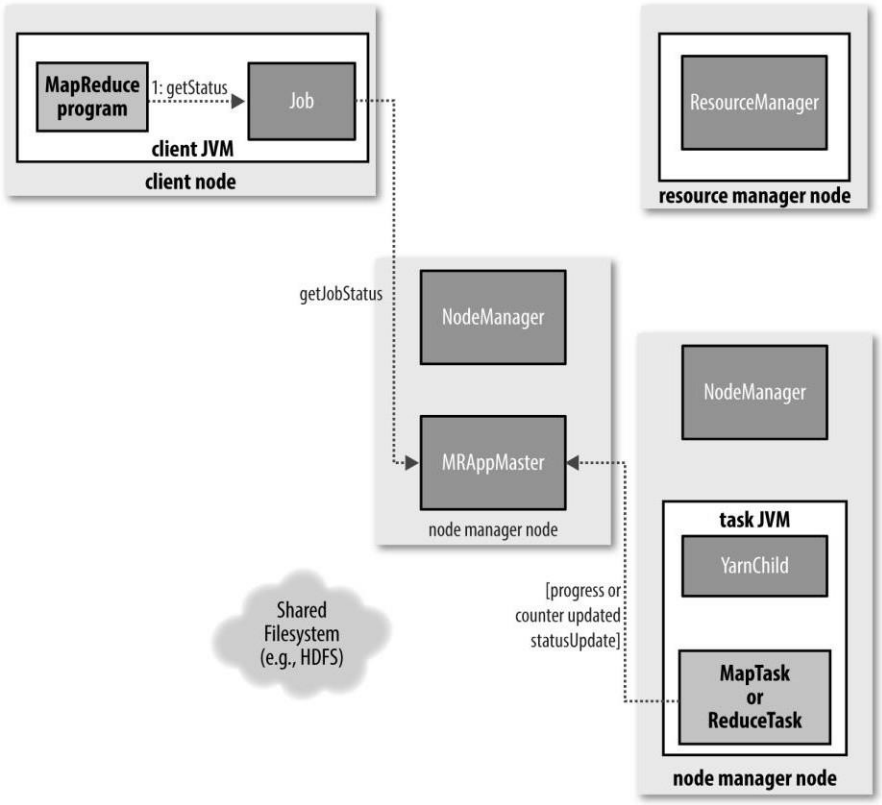


Figure 5-6. How status updates are propagated through the MapReduce system

Tasks also have a set of counters that count various events as the task runs, which are either built into the framework, such as the number of map output records written, or defined by users. As the map or reduce task runs, the child process communicates with its parent application master through the *umbilical* interface. The task reports its progress and status (including counters) back to its application master, which has an aggregate view of the job, every three seconds over the umbilical interface.

The resource manager web UI displays all the running applications with links to the web UIs of their respective application masters, each of which displays further details on the MapReduce job, including its progress.

During the course of the job, the client receives the latest status by polling the application master every second (the interval is set via mapreduce.client.progressmonitor.pol linterval). Clients can also use Job's getStatus() method to obtain a JobStatus instance, which contains all of the status information for the job.

The process is illustrated in **Figure 5-6.**

Job Completion

When the application master receives a notification that the last task for a job is complete, it changes

the status for the job to "successful." Then, when the Job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the waitForCompletion() method. Job statistics and counters are printed to the console at this point.The application master also sends an HTTP job notification if it is configured to do so. This can be configured by clients wishing to receive callbacks, via the mapreduce.job.end-notification.url property.

Finally, on job completion, the application master and the task containers clean up their working state (so intermediate output is deleted), and the OutputCommitter's commitJob() method is called. Job information is archived by the job history server to enable later interrogation by users if desired.

Tuning Map-Reduce jobs

After a job is working, the question many developers ask is, "Can I make it run faster?" There are a few Hadoop-specific "usual suspects" that are worth checking to see whether they are responsible for a performance problem. You should run through the checklist in Table-1 before you start trying to profile or optimize at the task level.

Area	Best practice
Number of mappers	How long are your mappers running for? If they are only running for a few seconds on average, you should see whether there's a way to have fewer mappers and make them all run longer—a minute or so, as a rule of thumb. The extent to which this is possible depends on the input format you are using.
Number of reducers	Check that you are using more than a single reducer. Reduce tasks should run for five minutes or so and produce at least a block's worth of data, as a rule of thumb.
Combiners	Check whether your job can take advantage of a combiner to reduce the amount of data passing through the shuffle.
Intermediate compression	Job execution time can almost always benefit from enabling map output compression.
Custom serialization	If you are using your own custom `Writable` objects or custom comparators, make sure you have implemented `RawComparator`.
Shuffle tweaks	The MapReduce shuffle exposes around a dozen tuning parameters for memory management, which may help you wring out the last bit of performance.

Table-1 Tuning checklist

Profiling Tasks

Hadoop allows you to profile a fraction of the tasks in a job and, as each task completes, pulls down the profile information to your machine for later analysis with standard profiling tools.

Of course, it's possible, and somewhat easier, to profile a job running in the local job runner. And provided you can run with enough input data to exercise the map and reduce tasks, this can be a valuable way of improving the performance of your mappers and reducers. There are a couple of caveats, however. The local job runner is a very different environment from a cluster, and the data

flow patterns are very different. Optimizing the CPU performance of your code may be pointless if your MapReduce job is I/O-bound (as many jobs are). To be sure that any tuning is effective, you should compare the new execution time with the old one running on a real cluster. Even this is easier said than done, since job execution times can vary due to resource contention with other jobs and the decisions the scheduler makes regarding task placement. To get a good idea of job execution time under these circumstances, perform a series of runs (with and without the change) and check whether any improvement is statistically significant.

The HPROF profiler

There are a number of configuration properties to control profiling, which are also exposed via convenience methods on JobConf. Enabling profiling is as simple as setting the property mapreduce.task.profile to true:

```
>% hadoop jar hadoop-examples.jar v4.MaxTemperatureDriver \ -conf conf/hadoop-cluster.xml \ -
D mapreduce.task.profile=true \ input/ncdc/all max-temp
```

This runs the job as normal, but adds an -agentlib parameter to the Java command used to launch the task containers on the node managers. You can control the precise parameter that is added by setting the mapreduce.task.profile.params property. The default uses HPROF, a profiling tool that comes with the JDK that,

The profile output for each task is saved with the task logs in the *userlogs* subdirectory of the node manager's local log directory (alongside the *syslog*, *stdout*, and *stderr* files), and can be retrieved in the way described in "Hadoop Logs", according to whether log aggregation is enabled or not.

Hadoop produces logs in various places, and for various audiences. These are summarized in Table-2

Understanding different logs produced by Map-Reduce

.

Logs	Primary audience	Description
System daemon logs	Administrators	Each Hadoop daemon produces a logfile (using log4j) and another file that combines standard out and error. Written in the directory defined by the HADOOP_LOG_DIR environment variable.
HDFS audit logs	Administrators	A log of all HDFS requests, turned off by default. Written to the namenode's log, although this is configurable.
MapReduce job history logs	Users	A log of the events (such as task completion) that occur in the course of running a job. Saved centrally in HDFS.
MapReduce task logs	Users	Each task child process produces a logfile using log4j (called *syslog*), a file for data sent to standard out (*stdout*), and a file for standard error (*stderr*). Written in the *userlogs* subdirectory of the directory defined by the YARN_LOG_DIR environment variable.

Table-2. Types of Hadoop logs

MapReduce log levels

MapReduce logs support various levels. You can configure the log levels for the MapReduce service and tasks.

You can set log levels to any of the following values:

Level	Description
DEBUG	Logs all debug-level and informational messages.
INFO	Logs all informational messages and more serious messages. This is the default log level.
WARN	Logs only those messages that are warnings or more serious messages. This is the default level of debug information.
ERROR	Logs only those messages that indicate error conditions or more serious messages.
FATAL	Logs only those messages in which the system is unusable.

To modify the level of the log printed to the console, change the value of the log4j.rootLogger property in the log configuration file

System logfiles

System logfiles produced by Hadoop are stored in $HADOOP_INSTALL/logs by default. This can be changed in *hadoop-env.sh***.**

Each Hadoop daemon running on a machine produces two logfiles.

The **first** is the log output written via log4j. This file, which ends in ***.log***. Old logfiles are never deleted, so you should arrange for them to be periodically deleted or archived, so as to not run out of disk space on the local node.

The **second** logfile is the combined standard output and standard error log. This logfile, which ends in ***.out,*** usually contains little or no output, since Hadoop uses log4j for logging. It is only rotated when the daemon is restarted, and only the last five logs are retained. Old logfiles are suffixed with a number between 1 and 5, with 5 being the oldest file.

Audit Logging

HDFS has the ability to log all filesystem access requests, a feature that some organizations require for auditing purposes. Audit logging is implemented using log4j logging at the INFO level, and in the default configuration it is disabled.You can enable audit logging by replacing WARN with INFO, and the result will be a log line written to the namenode's log for every HDFS event.

It is a good idea to configure log4j so that the audit log is written to a separate file and isn't mixed up with the namenode's other log entries.

Job History Logging

Job history refers to the events and configuration for a completed job. It is retained whether the job was successful or not, in an attempt to provide interesting information for the user running a job.

Job history files are stored on the local filesystem of the jobtracker in a *history* subdirectory of the logs directory.

The jobtracker's history files are kept for 30 days before being deleted by the system.

The history log includes job, task, and attempt events, all of which are stored in a plaintext file. The history for a particular job may be viewed through the web UI, or via the command line, using hadoop job -history (which you point at the job's output directory).

MapReduce task logs

These are accessible throughthe web UI, which is the most convenient way to view them. You can also find the logfiles on the local filesystem of the tasktracker that ran the task attempt, in a directory named by the task attempt. If task JVM reuse is enabled, then each task attempts will be found in each logfile. It is straightforward to write to these logfiles. Anything written to standard output, or standard error, is directed to the relevant logfile.

The default log level is INFO, so DEBUG level messages do not appear in the *syslog* task log file. However, sometimes you want to see these messages—to do this set mapred.map.child.log.level or mapred.reduce.child.log.level, as appropriate (from 0.22). For example, in this case we could set it for the mapper to see the map values in the log as follows:

```
>% hadoop jar hadoop-examples.jar LoggingDriver -conf conf/hadoop-
cluster.xml \-D mapred.map.child.log.level=DEBUG input/ncdc/sample.txt
logging-out
```

There are some controls for managing retention and size of task logs. By default, logs are deleted after a minimum of 24 hours (set using the mapred.userlog.retain.hoursproperty). You can also set a cap on the maximum size of each logfile using the mapred.userlog.limit.kb property, which is 0 by default, meaning there is no cap

4. Debugging the Map- Reduce jobs.

The time-honored way of debugging programs is via print statements, and this is certainly possible in Hadoop. However, there are complications to consider: with programs running on tens, hundreds, or thousands of nodes, how do we find and examine the output of the debug statements, which may be scattered across these nodes?

For this particular case, where we are looking for (what we think is) **an unusual case**, we can use a debug statement to log to standard error, in conjunction with a message to update the task's status message to prompt us to look in the error log. The web UI makes this easy, as we will see.

We also create a custom counter to count the total number of records with implausible temperatures in the whole dataset. This gives us valuable information about how to deal with the condition—if it turns out to be a common occurrence, then we might need to learn more about the condition and how to extract the temperature in these cases, rather than simply dropping the record. In fact, when trying to debug a job, you should always ask yourself if you can use a counter to get the information you need to find out what's happening. Even if you need to use logging or a status message, it may be useful to use a counter to gauge the extent of the problem.

If the amount of log data you produce in the course of debugging is large, then you've got a couple of options. The first is to write the information to the map's output, rather than to standard error, for analysis and aggregation by the reduce. This approach usually necessitates structural changes to your program, so start with the other techniques

You can write a program (in MapReduce of course) to analyze the logs produced by your job. We add our debugging to the mapper, as opposed to the reducer, as we want to find out what the source data causing the anomalous output looks like:

```
public class MaxTemperatureMapper
  extends Mapper<LongWritable, Text, Text, IntWritable> {

  enum Temperature {
    OVER_100
  }

  private NcdcRecordParser parser = new NcdcRecordParser();

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

    parser.parse(value);
    if (parser.isValidTemperature()) {
      int airTemperature = parser.getAirTemperature();
      if (airTemperature > 1000) {
        System.err.println("Temperature over 100 degrees for input: " + value);
        context.setStatus("Detected possibly corrupt record: see logs.");
        context.getCounter(Temperature.OVER_100).increment(1);
      }
      context.write(new Text(parser.getYear()), new IntWritable(airTemperature));
    }
  }
}
```

- Log
- Status
- Counter

If the temperature is over 100°C (represented by 1000, since temperatures are in tenths of a degree),

- we **print** a line to standard error with the suspect line, as well as updating the map's **status** message using the setStatus() method on Context directing us to look in the **log**. We also increment a **counter**, which in Java is represented by a field of an enum type. In this program, we have defined a single field OVER_100 as a way to count the number of records with a temperature of over 100°C.

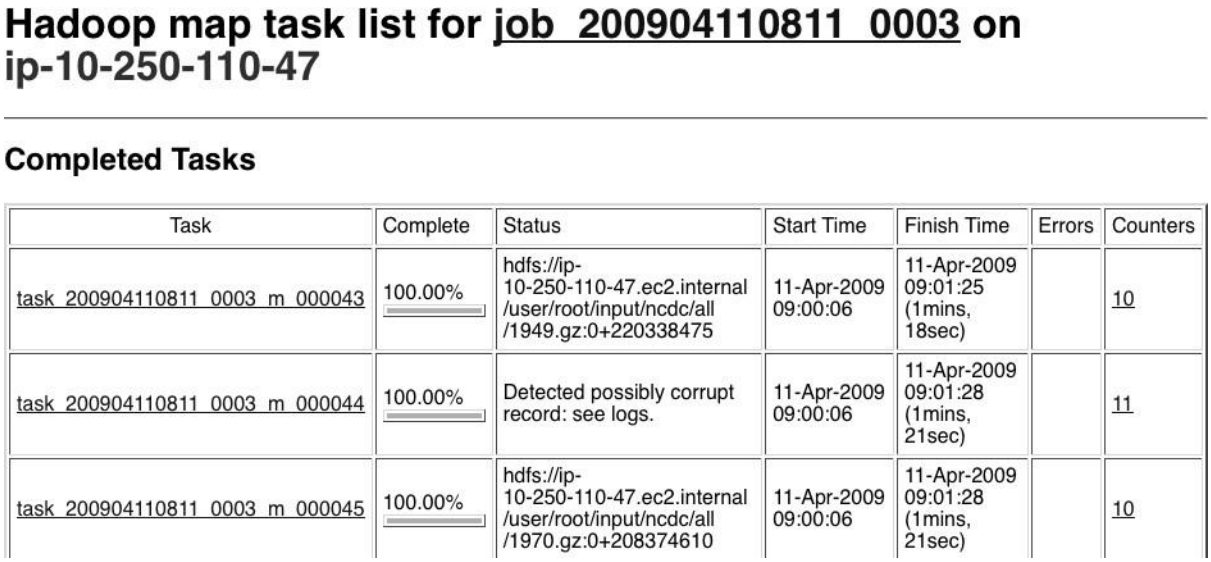
With this modification, we recompile the code, re-create the JAR file, then rerun the job, and while it's running go to the tasks page. The tasks page The job page has a number of links for look at the tasks in a job in more detail. For example, by clicking on the "map" link, you are brought to a page that lists information for all of the map tasks on one page. You can also see just the completed tasks. The screenshot in **Figure 5-7** shows a portion of this page for the job run with our debugging statements. Each row in the table is a task, and it provides such information as the start and end times for each task, any errors reported back from the tasktracker, and a link to view the counters for an individual task.

The "Status" column can be helpful for debugging, since it shows a task's latest status message.

Before a task starts, it shows its status as "initializing," then once it starts reading records it shows the split information for the split it is reading as a filename with a byte offset and length. You can see the status we set for debugging for task task_200904110811_0003_m_000044, so let's click through to the logs page to find the associated debug message. (Notice, too, that there is an extra counter for this task, since our user counter has a nonzero count for this task.)

The task details page From the tasks page, you can click on any task to get more information about it. The task details page, shown in **Figure 5-8,** shows each task attempt. In this case, there was one task attempt, which completed successfully. The table provides further useful data, such as the node the task attempt ran on, and links to task logfiles and counters.

The "Actions" column contains links for killing a task attempt. By default, this is disabled, making the web UI a read-only interface. Set webinterface.private.actions to true to enable the actions links.

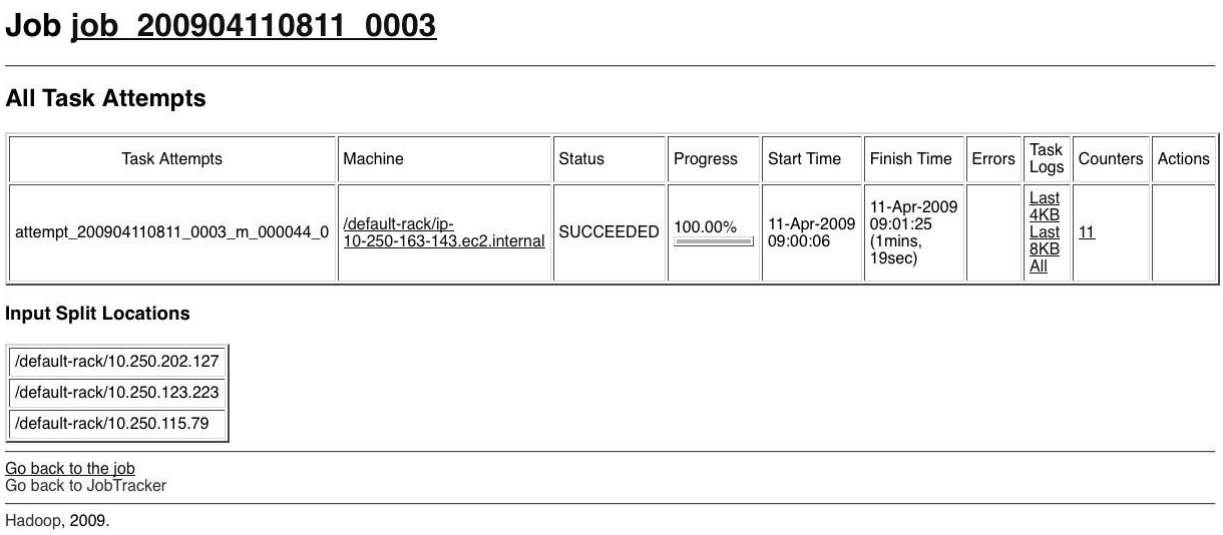


Hadoop map task list for job_200904110811_0003 on ip-10-250-110-47

Completed Tasks

Task	Complete	Status	Start Time	Finish Time	Errors	Counters
task_200904110811_0003_m_000043	100.00%	hdfs://ip-10-250-110-47.ec2.internal/user/root/input/ncdc/all/1949.gz:0+220338475	11-Apr-2009 09:00:06	11-Apr-2009 09:01:25 (1mins, 18sec)		10
task_200904110811_0003_m_000044	100.00%	Detected possibly corrupt record: see logs.	11-Apr-2009 09:00:06	11-Apr-2009 09:01:28 (1mins, 21sec)		11
task_200904110811_0003_m_000045	100.00%	hdfs://ip-10-250-110-47.ec2.internal/user/root/input/ncdc/all/1970.gz:0+208374610	11-Apr-2009 09:00:06	11-Apr-2009 09:01:28 (1mins, 21sec)		10

Figure 5-7. Screenshot of the tasks page



Job job_200904110811_0003

All Task Attempts

Task Attempts	Machine	Status	Progress	Start Time	Finish Time	Errors	Task Logs	Counters	Actions
attempt_200904110811_0003_m_000044_0	/default-rack/ip-10-250-163-143.ec2.internal	SUCCEEDED	100.00%	11-Apr-2009 09:00:06	11-Apr-2009 09:01:25 (1mins, 19sec)		Last 4KB Last 8KB All	11	

Input Split Locations

/default-rack/10.250.202.127
/default-rack/10.250.123.223
/default-rack/10.250.115.79

Go back to the job
Go back to JobTracker

Hadoop, 2009.

Figure 5-8. Screenshot of the task details page

For map tasks, there is also a section showing which nodes the input split was located on. By following one of the links to the logfiles for the successful task attempt (you can see the last 4 KB or 8 KB of each logfile, or the entire file), we can find the suspect input record that we logged (the line is wrapped and truncated to fit on the page):

Temperature over 100 degrees for input:

0335999999433181957042302005+37950+139117SAO+0004RJSNV02011359003150070356999999433**019**

57010100005+35317+139650SAO +000899999V02002359002650076249N004000599+0067...

This record seems to be in a different format to the others. For one thing, there are spaces in the line, which are not described in the specification.

When the job has finished, we can look at the value of the counter we defined to see how many records over 100°C there are in the whole dataset. Counters are accessible via the web UI or the command line:

```
>% hadoop job -counter job_200904110811_0003
    'v4.MaxTemperatureMapper$Temperature'
\OVER_100

3
```

The -counter option takes the job ID, counter group name (which is the fully qualified classname here), and the counter name (the enum name). There are only three malformed records in the entire dataset of over a billion records.

Throwing out bad records is standard for many big data problems, although we need to be careful in this case, since we are looking for an extreme value—the maximum temperature rather than an aggregate measure. Still, throwing away three records is probably not going to change the result.

Unit-5

Case studies of Big Data analytics using Map-Reduce programming

What is Big Data Analytics?

Big data analytics is the use of advanced analytic techniques against very large, diverse data sets that include different types such as structured/unstructured and streaming/batch and different sizes from terabytes to zettabytes. Big data is a term applied to data sets whose size or type is beyond the ability of traditional relational databases to capture, manage, and process the data with low-latency. And it has one or more of the following characteristics – high volume, high velocity, or high variety. Big data comes from sensors, devices, video/audio, networks, log files, transactional applications, web, and social media - much of it generated in real time and in a very large scale.

Analyzing big data allows analysts, researchers, and business users to make better and faster decisions using data that was previously inaccessible or unusable. Using advanced analytics techniques such as text analytics, machine learning, predictive analytics, data mining, statistics, and natural language processing, businesses can analyze previously untapped data sources independent or together with their existing enterprise data to gain new insights resulting in significantly better and faster decisions.

What is Machine Learning?

Machine learning is a branch of science that deals with programming the systems in such a way that they automatically learn and improve with experience. Here, learning means recognizing and understanding the input data and making wise decisions based on the supplied data.

It is very difficult to cater to all the decisions based on all possible inputs. To tackle this problem, algorithms are developed. These algorithms build knowledge from specific data and past experience with the principles of statistics, probability theory, logic, combinatorial optimization, search, reinforcement learning, and control theory.

The developed algorithms form the basis of various applications such as:

- Vision processing
- Language processing
- Forecasting (e.g., stock market trends)
- Pattern recognition
- Games
- Data mining
- Expert systems
- Robotics

Machine learning is a vast area and it is quite beyond the scope of this tutorial to cover all its features. There are several ways to implement machine learning techniques, however the most commonly used ones are **supervised** and **unsupervised learning**.

Supervised Learning

Supervised learning deals with learning a function from available training data. A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. Common examples of supervised learning include:

- classifying e-mails as spam,
- labeling webpages based on their content, and
- voice recognition.

There are many supervised learning algorithms such as neural networks, Support Vector Machines (SVMs), and Naive Bayes classifiers. Mahout implements Naive Bayes classifier.

Unsupervised Learning

Unsupervised learning makes sense of unlabeled data without having any predefined dataset for its training. Unsupervised learning is an extremely powerful tool for analyzing available data and look for patterns and trends. It is most commonly used for clustering similar input into logical groups. Common approaches to unsupervised learning include:

- k-means
- self-organizing maps, and
- hierarchical clustering

K-Means Clustering

Clustering is used to form groups or clusters of similar data based on common

characteristics. Clustering is a form of unsupervised learning.

- Search engines such as Google and Yahoo! use clustering techniques to group data with similar characteristics.
- Newsgroups use clustering techniques to group various articles based on related topics.

The clustering engine goes through the input data completely and based on the characteristics of the data, it will decide under which cluster it should be grouped.

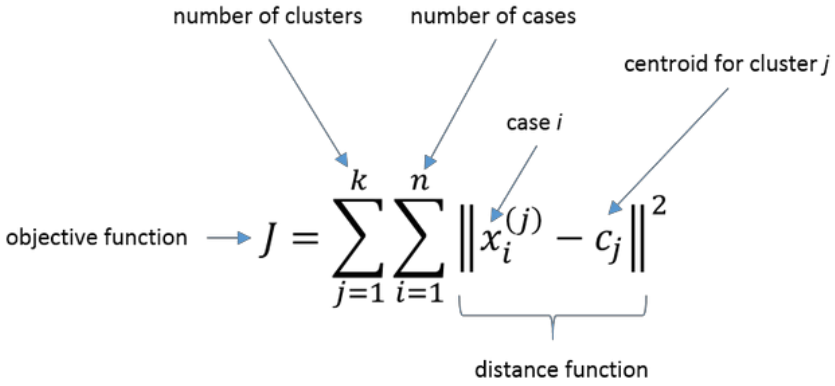
K-means clustering is a type of unsupervised learning, which is used when you have unlabeled data (i.e., data without defined categories or groups). The goal of this algorithm is to find groups in the data, with the number of groups represented by the variable *K*. The algorithm works iteratively to assign each data point to one of *K* groups based on the features that are provided. Data points are clustered based on feature similarity. The results of the *K*-means clustering algorithm are:

1. The centroids of the *K* clusters, which can be used to label new data
2. Labels for the training data (each data point is assigned to a single cluster)

Rather than defining groups before looking at the data, clustering allows you to find and analyze the groups that have formed organically. The "Choosing K" section below describes how the number of groups can be determined.

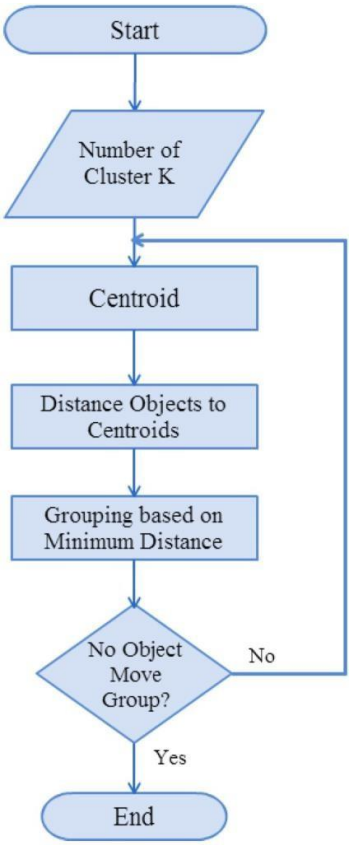
Each centroid of a cluster is a collection of feature values which define the resulting groups. Examining the centroid feature weights can be used to qualitatively interpret what kind of group each cluster represents.

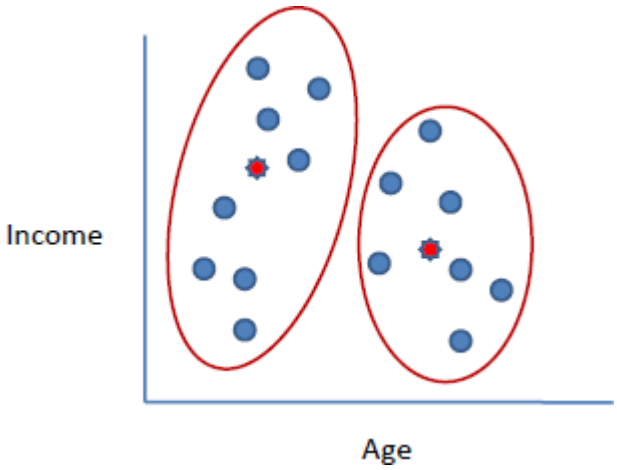
In general, we have n data points ***xi, i=1...n*** that have to be partitioned in k clusters. The goal is to assign a cluster to each data point. K-means is a clustering method that aims to find the positions c_i, $i=1...k$ of the clusters that minimize the *distance* from the data points to the cluster. K-means clustering solves



K-means algorithm

1. Clusters the data into *k* groups where *k* is predefined.
2. Select *k* points at random as cluster centers.
3. Assign objects to their closest cluster center according to the *Euclidean distance* function.
4. Calculate the centroid or mean of all objects in each cluster.
5. Repeat steps 2, 3 and 4 until the same points are assigned to each cluster in consecutive rounds.





K-Means is relatively an efficient method. However, we need to specify the number of clusters, in advance and the final results are sensitive to initialization and often terminates at a local optimum. Unfortunately there is no global theoretical method to find the optimal number of clusters. A practical approach is to compare the outcomes of multiple runs with different *k* and choose the best one based on a predefined criterion. In general, a large *k* probably decreases the error but increases the risk of overfitting.

Example:

Suppose we want to group the visitors to a website using just their age (a one-dimensional space) as follows:

15,15,16,19,19,20,20,21,22,28,35,40,41,42,43,44,60,61,65

Initial clusters:

Centroid (C1) = 16 [16]

Centroid (C2) = 22 [22]

15+15+16=46
46/3=15.33

Iteration **1**:

C1 = 15.33 [15,15,16]

C2 = 36.25 [19,19,20,20,21,22,28,35,40,41,42,43,44,60,61,65]

Iteration **2**:

C1 = 18.56 [15,15,16,19,19,20,20,21,22]

C2 = 45.90 [28,35,40,41,42,43,44,60,61,65]

Iteration **3**:

C1 = 19.50 [15,15,16,19,19,20,20,21,22,28]

C2 = 47.89 [35,40,41,42,43,44,60,61,65]

Iteration **4**:

C1 = 19.50 [15,15,16,19,19,20,20,21,22,28]

C2 = 47.89 [35,40,41,42,43,44,60,61,65]

No change between iterations 3 and 4 has been noted. By using clustering, 2 groups have been identified 15-28 and 35-65. The initial choice of centroids can affect the output clusters, so the algorithm is often run multiple times with different starting conditions in order to get a fair view of what the clusters should be.

MapReduce Approach

MapReduce works on keys and values, and is based on data partitioning. Thus, the assumption of having all data points in memory fails in this paradigm. We have to design the algorithm in such a manner that the task can be parallelized and doesn't depend on other splits for any computation (Figure below).

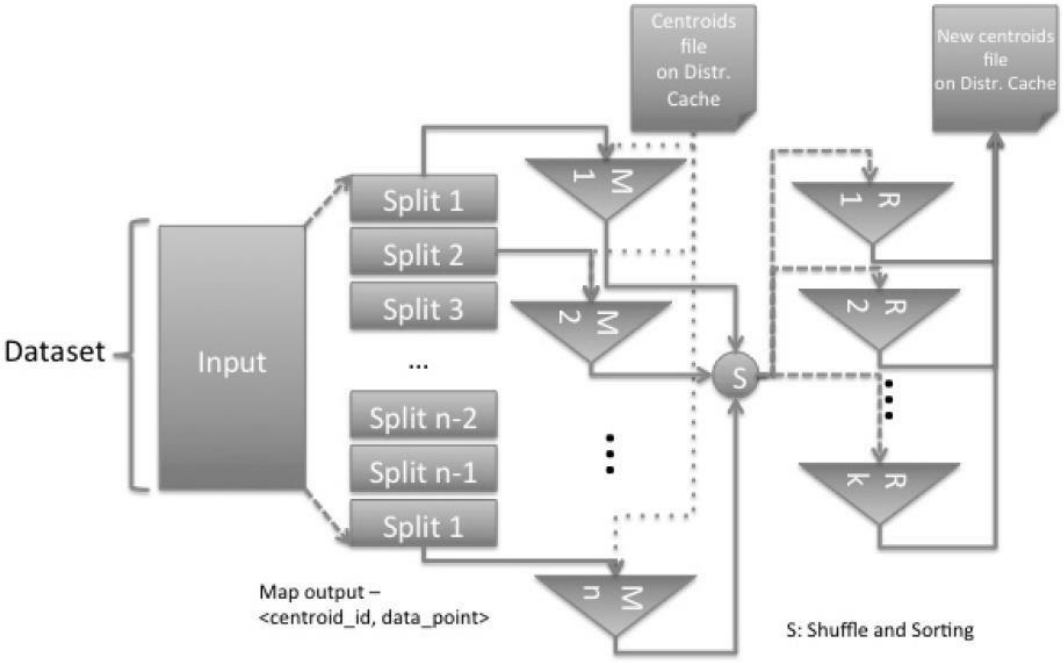


Figure . Single pass of K-Means on MapReduce

The Mappers do the distance computation and spill out a key-value pair – **<centroid_id, datapoint>.** This step finds the associativity of a data point with the cluster.

The Reducers work with specific cluster_id and a list of the data points associated with it. A reducer computes new means and writes to the new centroid file.

Now, based on the user's choice, algorithm termination method works – specific number of iterations, or comparison with centroid in the previous iteration.

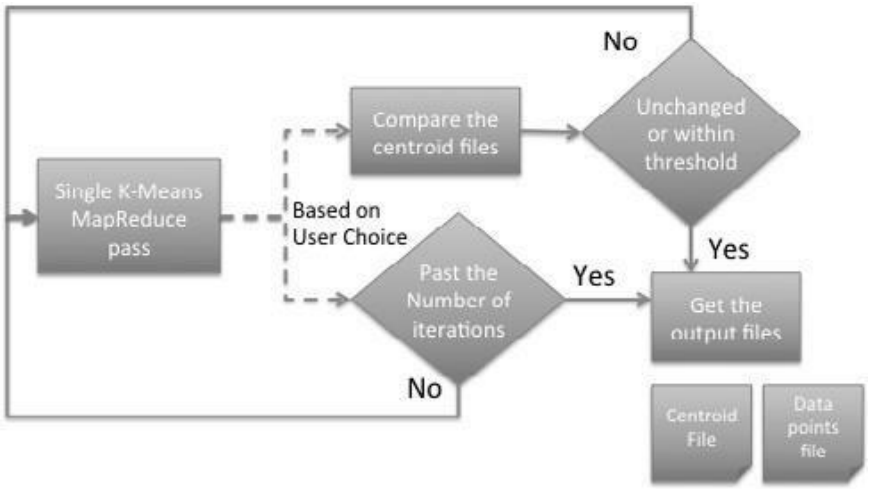


Figure . K-Means Algorithm. Algorithm termination method is user-driven

APACHE MAHOUT:

Mahout - Introduction

We are living in a day and age where information is available in abundance. The information overload has scaled to such heights that sometimes it becomes difficult to manage our little mailboxes! Imagine the volume of data and records some of the popular websites (the likes of Facebook, Twitter, and Youtube) have to collect and manage on a daily basis. It is not uncommon even for lesser known websites to receive huge amounts of information in bulk.

Normally we fall back on data mining algorithms to analyze bulk data to identify trends and draw conclusions. However, no data mining algorithm can be efficient enough to process very large datasets and provide outcomes in quick time, unless the computational tasks are run on multiple machines distributed over the cloud.

We now have new frameworks that allow us to break down a computation task into multiple segments and run each segment on a different machine. **Mahout** is such a data mining framework that normally runs coupled with the Hadoop infrastructure at its background to manage huge volumes of data.

What is Apache Mahout?

A *mahout* is one who drives an elephant as its master. The name comes from its close association with Apache Hadoop which uses an elephant as its logo.

Hadoop is an open-source framework from Apache that allows to store and process big data in a distributed environment across clusters of computers using simple programming models.

Apache **Mahout** is an open source project that is primarily used for creating scalable machine learning algorithms. It implements popular machine learning techniques such as:

a. Recommendation
b. Classification
c. Clustering

Apache Mahout started as a sub-project of Apache's Lucene in 2008. In 2010, Mahout became a top level project of Apache.

Features of Mahout

The primitive features of Apache Mahout are listed below.

- The algorithms of Mahout are written on top of Hadoop, so it works well in distributed environment. Mahout uses the Apache Hadoop library to scale effectively in the cloud.
- Mahout offers the coder a ready-to-use framework for doing data mining tasks on large volumes of data.
- Mahout lets applications to analyze large sets of data effectively and in quick time.
- Includes several MapReduce enabled clustering implementations such as k-means, fuzzy k- means, Canopy, Dirichlet, and Mean-Shift.
- Supports Distributed Naive Bayes and Complementary Naïve Bayes classification implementations.
- Comes with distributed fitness function capabilities for evolutionary programming.
- Includes matrix and vector libraries.

Applications of Mahout

- Companies such as Adobe, Facebook, LinkedIn, Foursquare, Twitter, and Yahoo use Mahout internally.
- Foursquare helps you in finding out places, food, and entertainment available in a particular area. It uses the recommender engine of Mahout.
- Twitter uses Mahout for user interest modelling.
- Yahoo! uses Mahout for pattern mining.

Getting started with Mahout

Getting up and running with Mahout is relatively straightforward. To start, you need to install the following prerequisites:

JDK 1.6 or higher Ant 1.7 or higher

If you want to build the Mahout source, Maven 2.0.9 or 2.0.10

You also need this article's sample code (see Download), which includes a copy of Mahout and its dependencies. Follow these steps to install the sample code:

1. **unzip sample.zip**
2. **cd apache-mahout-examples**
3. **ant install**

Step 3 downloads the necessary Wikipedia files and compiles the code. The Wikipedia file used is approximately 2.5 gigabytes, so download times will depend on your bandwidth.

a. Recommendation

Recommendation is a popular technique that provides close recommendations based on user information such as previous purchases, clicks, and ratings.

- Amazon uses this technique to display a list of recommended items that you might be interested in, drawing information from your past actions. There are recommender engines that work behind Amazon to capture user behavior and recommend selected items based on your earlier actions.
- Facebook uses the recommender technique to identify and recommend the “people you may know list”.

Building a recommendation engine:

Mahout currently provides tools for building a recommendation engine through the Taste library

— a fast and flexible engine for CF. Taste supports both user-based and item-based recommendations and comes with many choices for making recommendations, as well as

interfaces for you to define your own. Taste consists of five primary components that work with Users, Items and Preferences:

Data Model: Storage for Users, Items, and Preferences

User Similarity: Interface defining the similarity between two users Item Similarity: Interface defining the similarity between two items Recommender: Interface for providing recommendations

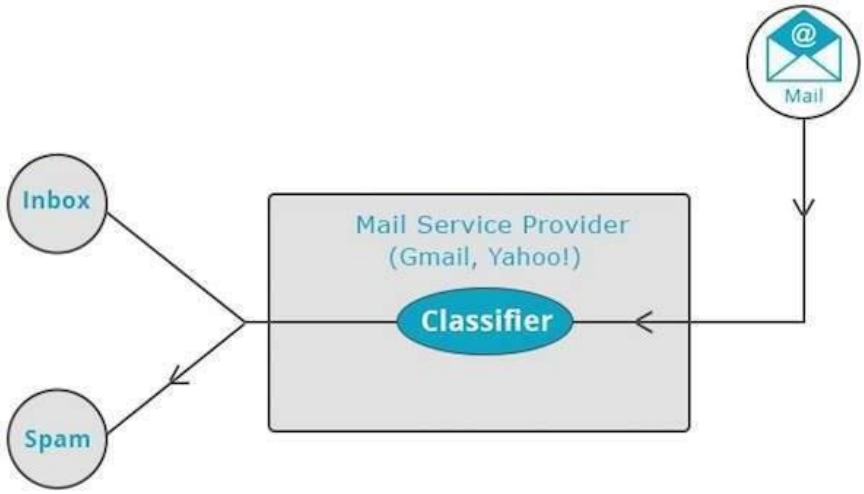
User Neighborhood: Interface for computing a neighborhood of similar users that can then be used by the Recommenders

These components and their implementations make it possible to build out complex recommendation systems for either real-time-based recommendations or offline recommendations. Real-time-based recommendations often can handle only a few thousand users, whereas offline recommendations can scale much higher. Taste even comes with tools for leveraging Hadoop to calculate recommendations offline. In many cases, this is a reasonable approach that allows you to meet the demands of a large system with a lot of users, items, and preferences.

b. Classification

Classification, also known as **categorization**, is a machine learning technique that uses known data to determine how the new data should be classified into a set of existing categories. Classification is a form of supervised learning.

- Mail service providers such as Yahoo! and Gmail use this technique to decide whether a new mail should be classified as a spam. The categorization algorithm trains itself by analyzing user habits of marking certain mails as spams. Based on that, the classifier decides whether a future mail should be deposited in your inbox or in the spams folder.

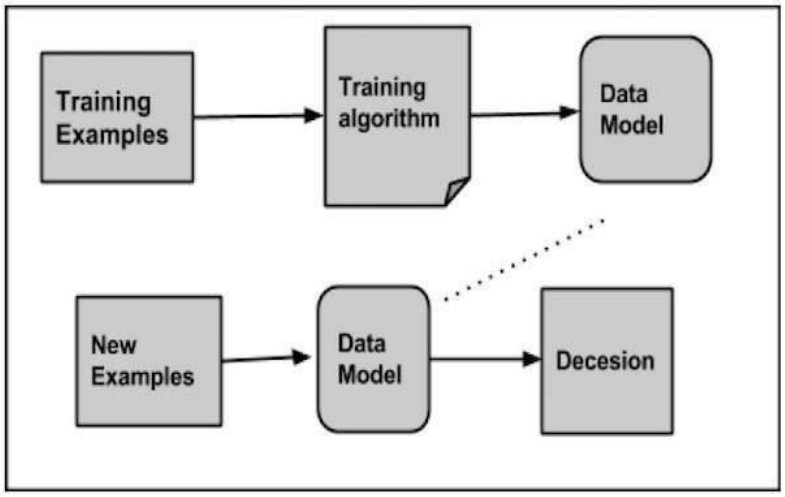


- iTunes application uses classification to prepare playlists.

How Classification Works

While classifying a given set of data, the classifier system performs the following actions:

- Initially a new data model is prepared using any of the learning algorithms.
- Then the prepared data model is tested.
- Thereafter, this data model is used to evaluate the new data and to determine its class.



Applications of Classification

- **Credit card fraud detection** - The Classification mechanism is used to predict credit card frauds. Using historical information of previous frauds, the classifier can predict which future transactions may turn into frauds.
- **Spam e-mails** - Depending on the characteristics of previous spam mails, the classifier determines whether a newly encountered e-mail should be sent to the spam folder.

Naive Bayes Classifier

Mahout uses the Naive Bayes classifier algorithm. It uses two implementations:

- Distributed Naive Bayes classification
- Complementary Naive Bayes classification

Naive Bayes is a simple technique for constructing classifiers. It is not a single algorithm for training such classifiers, but a family of algorithms. A Bayes classifier constructs models to classify problem instances. These classifications are made using the available data.

An advantage of naive Bayes is that it only requires a small amount of training data to estimate the parameters necessary for classification.

For some types of probability models, naive Bayes classifiers can be trained very efficiently in a supervised learning setting.

Despite its oversimplified assumptions, naive Bayes classifiers have worked quite well in many complex real-world situations.

Procedure of Classification

The following steps are to be followed to implement Classification:

- Generate example data
- Create sequence files from data
- Convert sequence files to vectors
- Train the vectors
- Test the vectors

c. Clustering

Clustering is used to form groups or clusters of similar data based on common characteristics. Clustering is a form of unsupervised learning.

- Search engines such as Google and Yahoo! use clustering techniques to group data with similar characteristics.
- Newsgroups use clustering techniques to group various articles based on related topics.

The clustering engine goes through the input data completely and based on the characteristics of the data, it will decide under which cluster it should be grouped.

Using Mahout, we can cluster a given set of data. The steps required are as follows:

- **Algorithm** You need to select a suitable clustering algorithm to group the elements of a cluster.
-
- **Similarity and Dissimilarity** You need to have a rule in place to verify the similarity between the newly encountered elements and the elements in the groups.
- **Stopping Condition** A stopping condition is required to define the point where no clustering is required.

Procedure of Clustering

To cluster the given data you need to -

- Start the Hadoop server. Create required directories for storing files in Hadoop File

System. (Create directories for input file, sequence file, and clustered output in case of canopy).

- Copy the input file to the Hadoop File system from Unix file system.
- Prepare the sequence file from the input data.
- Run any of the available clustering algorithms.
- Get the clustered data.

Mahout supports several clustering-algorithm implementations, all written in Map-Reduce, each with its own set of goals and criteria:

Canopy: A fast clustering algorithm often used to create initial seeds for other clustering algorithms.

k-Means (and **fuzzy k-Means**): Clusters items into k clusters based on the distance the items are from the centroid, or center, of the previous iteration.

Mean-Shift: Algorithm that does not require any *a priori* knowledge about the number of clusters and can produce arbitrarily shaped clusters.

Dirichlet: Clusters based on the mixing of many probabilistic models giving it the advantage that it doesn't need to commit to a particular view of the clusters prematurely.