LECTURE NOTES ON

INTRODUCTION TO BIG DATA
(15A05506)

III B.TECH I SEMESTER

(JNTUA-R15)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VEMU INSTITUTE OF TECHNOLOGY:: P.KOTHAKOTA
Chittoor-Tirupati National Highway, P.Kothakota, Near Pakala, Chittoor (Dt.), AP - 517112
(Approved by AICTE, New Delhi Affiliated to JNTUA Ananthapuramu. ISO 9001:2015 Certified Institute)

Introduction to Big Data
(15A05506)

SYLLABUS

Unit-1: Distributed programming using JAVA: Quick Recap and advanced Java Programming: Generics, Threads, Sockets, Simple client server Programming using JAVA, Difficulties in developing distributed programs for large scale clusters and introduction to cloud computing.

Unit-2: Distributed File systems leading to Hadoop file system, introduction, Using HDFS, Hadoop Architecture, Internals of Hadoop File Systems.

Unit-3: Map-Reduce Programming: Developing Distributed Programs and issues, why map-reduce and conceptual understanding of Map-Reduce programming, Developing Map-Reduce programs in Java, setting up the cluster with HDFS and understanding how Map- Reduce works on HDFS, Running simple word count Map-Reduce program on the cluster, Additional examples of M-R Programming.

Unit-4: Anatomy of Map-Reduce Jobs: Understanding how Map- Reduce program works, tuning Map-Reduce jobs, Understanding different logs produced by Map-Reduce jobs and debugging the Map- Reduce jobs.

Unit-5: Case studies of Big Data analytics using Map-Reduce programming: K-Means clustering, using Big Data analytics libraries using Mahout.

Text Books:

1. JAVA in a Nutshell 4th Edition.
2. Hadoop: The definitive Guide by Tom White, 3rd Edition, O'reily.

References:

1. Hadoop in Action by Chuck Lam, Manning Publications.

Unit-1

Java is a high-level programming language originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. The following are some of the salient features of Java Programming language.

- **Object Oriented** – In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent** – Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
- **Simple** – Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **Secure** – With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architecture-neutral** – Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable** – Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **Robust** – Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded** – With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** – Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.

- **High Performance** – With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed** – Java is designed for the distributed environment of the internet.
- **Dynamic** – Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Multithreading in java is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

3) Threads are **independent** so it doesn't affect other threads if exception occurs in a single thread.

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.

- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

Java Thread class

Thread class is the main class on which java's multithreading system is based. Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Java Thread Methods

S.N.	Modifier and Type	Method	Description
1	void	run()	It is used to perform action for a thread.
2	void	start()	It starts the execution of the thread.JVM calls the run() method on the thread.
3	static void	sleep(long miliseconds)	It sleeps a thread for the specified amount of time.

4	void	join(long miliseconds)	It waits for a thread to die.
5	int	getPriority()	It returns the priority of the thread.
6	void	setPriority(int priority)	It changes the priority of the thread.
7	String	getName()	It returns the name of the thread.
8	void	setName(String name)	It changes the name of the thread.
9	static Thread	currentThread()	It returns the reference of currently executing thread.
10	long	getId()	It returns the id of the thread.
11	boolean	isAlive()	It tests if the thread is alive.
12	static void	yield()	It causes the currently executing thread object to temporarily pause and allow other threads to execute.
13	void	suspend()	It is used to suspend the thread.
14	void	resume()	It is used to resume the suspended thread.
15	void	stop()	It is used to stop the thread.

16	boolean	isDaemon()	It tests if the thread is a daemon thread.
17	void	setDaemon(Boolean on)	It marks the thread as daemon or user thread.
18	void	interrupt()	It interrupts the thread.
19	static boolean	interrupted()	It tests if the current thread has been interrupted.
20	boolean	isInterrupted()	It tests if the thread has been interrupted.
21	static int	activeCount()	It returns the number of active threads in the current thread's thread group.
22	void	checkAccess()	It determines if the currently running thread has permission to modify this thread.
23	protected Object	clone()	It returns a clone if the class of this object is Cloneable.
24	static void	dumpStack()	It is used to print a stack trace of the current thread to the standard error stream.
25	Thread.State	getState()	It is used to return the state of the thread.

26	ThreadGroup	getThreadGroup()	It is used to return the thread group to which this thread belongs
27	String	toString()	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.

Java Networking is a concept of connecting two or more computing devices together so that we can share resources. Java socket programming provides facility to share data between different computing devices.

Advantage of Java Networking

1. sharing resources
2. centralize software management

The widely used java networking terminologies are given below:

1. IP Address
2. Protocol
3. Port Number
4. MAC Address
5. Connection-oriented and connection-less protocol
6. Socket

1) IP Address

IP address is a unique number assigned to a node of a network e.g. 192.168.0.1 . It is composed of octets that range from 0 to 255.

It is a logical address that can be changed.

2) Protocol

A protocol is a set of rules basically that is followed for communication. For example:

- TCP
- FTP
- Telnet
- SMTP
- POP etc.

3) Port Number

The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications.

The port number is associated with the IP address for communication between two applications.

4) MAC Address

MAC (Media Access Control) Address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NIC but each with unique MAC.

5) Connection-oriented and connection-less protocol

In connection-oriented protocol, acknowledgement is sent by the receiver. So it is reliable but slow. The example of connection-oriented protocol is TCP.

But, in connection-less protocol, acknowledgement is not sent by the receiver. So it is not reliable but fast. The example of connection-less protocol is UDP.

6) Socket

A socket is an endpoint between two way communication.

Java Socket Programming

Java Socket programming is used for communication between the applications running on different JRE.

Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Socket class

A socket is simply an endpoint for communications between the machines. 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

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

File: MyServer.java

```
1.  import java.io.*;
2.  import java.net.*;
3.  public class MyServer {
4.  public static void main(String[] args){
5.  try{
6.  ServerSocket ss=new ServerSocket(6666);
7.  Socket s=ss.accept();//establishes connection
8.  DataInputStream dis=new DataInputStream(s.getInputStream());
9.  String  str=(String)dis.readUTF();
10. System.out.println("message= "+str);
11. ss.close();
12. }catch(Exception e){System.out.println(e);}
13. }
14. }
```

File: MyClient.java

```
1.  import java.io.*;
2.  import java.net.*;
```

```
3.  public class MyClient {
4.  public static void main(String[] args) {
5.  try{
6.  Socket s=new Socket("localhost",6666);
7.  DataOutputStream dout=new DataOutputStream(s.getOutputStream());
8.  dout.writeUTF("Hello Server");
9.  dout.flush();
10. dout.close();
11. s.close();
12. }catch(Exception e){System.out.println(e);}
13. }
14. }
```

download this example

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 of Java Socket Programming (Read-Write both side)

In this example, client will write first to the server then server will receive and print the text. Then server will write to the client and client will receive and print the text. The step goes on.

File: MyServer.java

```
1.  import java.net.*;
2.  import java.io.*;
3.  class MyServer{
4.  public static void main(String args[])throws Exception{
5.  ServerSocket ss=new ServerSocket(3333);
6.  Socket s=ss.accept();
7.  DataInputStream din=new DataInputStream(s.getInputStream());
8.  DataOutputStream dout=new DataOutputStream(s.getOutputStream());
```

```
9.      BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
10.
11.     String str="",str2="";
12.     while(!str.equals("stop")){
13.     str=din.readUTF();
14.     System.out.println("client says: "+str);
15.     str2=br.readLine();
16.     dout.writeUTF(str2);
17.     dout.flush();
18.     }
19.     din.close();
20.     s.close();
21.     ss.close();
22.     }}
```

File: MyClient.java

```
1.      import java.net.*;
2.      import java.io.*;
3.      class MyClient{
4.      public static void main(String args[])throws Exception{
5.      Socket s=new Socket("localhost",3333);
6.      DataInputStream din=new DataInputStream(s.getInputStream());
7.      DataOutputStream dout=new DataOutputStream(s.getOutputStream());
8.      BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
9.
10.     String str="",str2="";
11.     while(!str.equals("stop")){
12.     str=br.readLine();
13.     dout.writeUTF(str);
14.     dout.flush();
15.     str2=din.readUTF();
16.     System.out.println("Server says: "+str2);
17.     }
18.
19.     dout.close();
20.     s.close();
```

```
21.       }}
```

Generics

generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

- Stronger type checks at compile time.
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- Elimination of casts.
  The following code snippet without generics requires casting:
- List list = new ArrayList();
- list.add("hello");
- String s = **(String)** list.get(0);

  When re-written to use generics, the code does not require casting:

  List<String> list = new ArrayList<String>();
  list.add("hello");
  String s = list.get(0); // no cast

- Enabling programmers to implement generic algorithms.
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Generic Types

A *generic type* is a generic class or interface that is parameterized over types. The following `Box` class will be modified to demonstrate the concept.

A Simple Box Class

Begin by examining a non-generic `Box` class that operates on objects of any type. It needs only to provide two methods: `set`, which adds an object to the box, and `get`, which retrieves it:

```
public class Box {
```

```
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Since its methods accept or return an `Object`, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an `Integer` in the box and expect to get `Integer`s out of it, while another part of the code may mistakenly pass in a `String`, resulting in a runtime error.

A Generic Version of the Box Class

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) `T1`, `T2`, ..., and `Tn`.

To update the `Box` class to use generics, you create a *generic type declaration* by changing the code "`public class Box`" to "`public class Box<T>`". This introduces the type variable, `T`, that can be used anywhere inside the class.

With this change, the `Box` class becomes:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of `Object` are replaced by `T`. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

You'll see these names used throughout the Java SE API and the rest of this lesson.

Invoking and Instantiating a Generic Type

To reference the generic `Box` class from within your code, you must perform a *generic type invocation*, which replaces `T` with some concrete value, such as `Integer`:

```
Box<Integer> integerBox;
```

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a *type argument* — `Integer` in this case — to the `Box` class itself.

Type Parameter and Type Argument Terminology: Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides type arguments in order to create a parameterized type. Therefore, the `T` in `Foo<T>` is a type parameter and the `String` in `Foo<String> f` is a type argument. This lesson observes this definition when using these terms.

Like any other variable declaration, this code does not actually create a new `Box` object. It simply declares that `integerBox` will hold a reference to a "`Box` of `Integer`", which is how `Box<Integer>` is read.

An invocation of a generic type is generally known as a *parameterized type*.

To instantiate this class, use the `new` keyword, as usual, but place `<Integer>` between the class name and the parenthesis:

```
Box<Integer> integerBox = new Box<Integer>();
```

The Diamond

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called *the diamond*. For example, you can create an instance of `Box<Integer>` with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic `OrderedPair` class, which implements the generic `Pair` interface:

```
public interface Pair<K, V> {
    public K  getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()  { return key; }
    public V getValue() { return value; }
}
```

The following statements create two instantiations of the `OrderedPair` class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

The code, `new OrderedPair<String, Integer>`, instantiates `K` as a `String` and `V` as an `Integer`. Therefore, the parameter types of `OrderedPair`'s constructor are `String` and `Integer`, respectively. Due to autoboxing, it is valid to pass a `String` and an `int` to the class.

As mentioned in The Diamond, because a Java compiler can infer the `K` and `V` types from the declaration `OrderedPair<String, Integer>`, these statements can be shortened using diamond notation:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

To create a generic interface, follow the same conventions as for creating a generic class.

Parameterized Types

You can also substitute a type parameter (i.e., `K` or `V`) with a parameterized type (i.e., `List<String>`). For example, using the `OrderedPair<K, V>` example:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new
Box<Integer>(...));
```

Raw Types

A *raw type* is the name of a generic class or interface without any type arguments. For example, given the generic `Box` class:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

To create a parameterized type of `Box<T>`, you supply an actual type argument for the formal type parameter `T`:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of `Box<T>`:

```
Box rawBox = new Box();
```

Therefore, `Box` is the raw type of the generic type `Box<T>`. However, a non-generic class or interface type is *not* a raw type.

Raw types show up in legacy code because lots of API classes (such as the `Collections` classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a `Box` gives you `Object`s. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;               // OK
```

But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

```
Note: Example.java uses unchecked or unsafe operations.
```

This can happen when using an older API that operates on raw types, as shown in the following example:

```
public class WarningDemo {
    public static void main(String[] args){
        Box<Integer> bi;
        bi = createBox();
    }

    static Box createBox(){
        return new Box();
    }
}
```

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with `-Xlint:unchecked`.

Recompiling the previous example with `-Xlint:unchecked` reveals the following additional information:

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found   : Box
required: Box<java.lang.Integer>
        bi = createBox();
                     ^
1 warning
```

CHALLENGES AT LARGE SCALE

Performing large-scale computation is difficult. To work with this volume of data requires distributing parts of the problem to multiple machines to handle in parallel. Whenever multiple machines are used in cooperation with one another, the probability of failures rises. In a single-machine environment, failure is not something that program designers explicitly worry about very often: if the machine has crashed, then there is no way for the program to recover anyway.

In a distributed environment, however, partial failures are an expected and common occurrence. Networks can experience partial or total failure if switches and routers break down. Data may not arrive at a particular point in time due to unexpected network congestion. Individual compute nodes may overheat, crash, experience hard drive failures, or run out of memory or disk space. Data may be corrupted, or maliciously or improperly transmitted. Multiple implementations or versions of client software may speak slightly different protocols from one another. Clocks may become desynchronized, lock files may not be released, parties involved in distributed atomic transactions may lose their network connections part-way

through, etc. In each of these cases, the rest of the distributed system should be able to recover from the component failure or transient error condition and continue to make progress. Of course, actually providing such resilience is a major software engineering challenge.

Different distributed systems specifically address certain modes of failure, while worrying less about others. Hadoop provides no security model, nor safeguards against maliciously inserted data. For example, it cannot detect a man-in-the-middle attack between nodes. On the other hand, it is designed to handle hardware failure and data congestion issues very robustly. Other distributed systems make different trade-offs, as they intend to be used for problems with other requirements (e.g., high security).

In addition to worrying about these sorts of bugs and challenges, there is also the fact that the compute hardware has finite resources available to it.

The major resources include:

- Processor time
- Memory
- Hard drive space
- Network bandwidth

Individual machines typically only have a few gigabytes of memory. If the input data set is several terabytes, then this would require a thousand or more machines to hold it in RAM -- and even then, no single machine would be able to process or address all of the data.

Hard drives are much larger; a single machine can now hold multiple terabytes of information on its hard drives. But intermediate data sets generated while performing a large-scale computation can easily fill up several times more space than what the original input data set had occupied. During this process, some of the hard drives employed by the system may become full, and the distributed system may need to route this data to other nodes which can store the overflow.

Finally, bandwidth is a scarce resource even on an internal network. While a set of nodes directly connected by a gigabit Ethernet may generally experience high throughput between them, if all of the machines were transmitting multi-gigabyte data sets, they can easily saturate the switch's bandwidth capacity. Additionally if the machines are spread across multiple racks, the bandwidth available for the data transfer would be much less.

Furthermore RPC requests and other data transfer requests using this channel may be delayed or dropped.

To be successful, a large-scale distributed system must be able to manage the above mentioned resources efficiently. Furthermore, it must allocate some of these resources toward maintaining the system as a whole, while devoting as much time as possible to the actual core computation.

Synchronization between multiple machines remains the biggest challenge in distributed system design. If nodes in a distributed system can explicitly communicate with one another, then application designers must be cognizant of risks associated with such communication patterns. It becomes very easy to generate more remote procedure calls (RPCs) than the system can satisfy! Performing multi-party data exchanges is also prone to deadlock or race conditions. Finally, the ability to continue computation in the face of failures becomes more challenging. For example, if 100 nodes are present in a system and one of them crashes, the other 99 nodes should be able to continue the computation, ideally with only a small penalty proportionate to the loss of 1% of the computing power. Of course, this will require re-computing any work lost on the unavailable node. Furthermore, if a complex communication network is overlaid on the distributed infrastructure, then determining how best to restart the lost computation and propagating this information about the change in network topology may be non trivial to implement.

Cloud computing is an information technology (IT) paradigm that enables ubiquitous access to shared pools of configurable system resources and higher-level services that can be rapidly provisioned with minimal management effort, often over the Internet. Cloud computing relies on sharing of resources to achieve coherence and economies of scale, similar to a public utility. Third-party clouds enable organizations to focus on their core businesses instead of expending resources on computer infrastructure and maintenance.[1] Advocates note that cloud computing allows companies to avoid or minimize up-front IT infrastructure costs. Proponents also claim that cloud computing allows enterprises to get their applications up and running faster, with improved manageability and less maintenance, and that it enables IT teams to more rapidly adjust resources to meet fluctuating and unpredictable demand.Cloud providers typically use a "pay-as-you-go" model, which can lead to unexpected operating expenses if administrators are not familiarized with cloud-pricing models.

Unit-2:

Distributed File System Basics

A distributed file system is designed to hold a large amount of data and provide access to this data to many clients distributed across a network. There are a number of distributed file systems that solve this problem in different ways.

NFS, the Network File System, is the most ubiquitous distributed file system. It is one of the oldest still in use. While its design is straightforward, it is also very constrained. NFS provides remote access to a single logical volume stored on a single machine. An NFS server makes a portion of its local file system visible to external clients. The clients can then mount this remote file system directly into their own Linux file system, and interact with it as though it were part of the local drive.

One of the primary advantages of this model is its transparency. Clients do not need to be particularly aware that they are working on files stored remotely. The existing standard library methods like open(), close(), fread(), etc. will work on files hosted over NFS.

But as a distributed file system, it is limited in its power. The files in an NFS volume all reside on a single machine. This means that it will only store as much information as can be stored in one machine, and does not provide any reliability guarantees if that machine goes down (e.g., by replicating the files to other servers). 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.

HDFS is designed to be robust to a number of the problems that other DFS's such as NFS are vulnerable to. In particular:

- 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 NFS.
- HDFS should store data reliably. If individual machines in the cluster malfunction, data should still be available.
- 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.
- HDFS should integrate well with Hadoop MapReduce, allowing data to be read and computed upon locally when possible.

But while HDFS is very scalable, its high performance design also restricts it to a particular class of applications; it is not as general-purpose as NFS. There are a large number of additional decisions and trade-offs that were made with HDFS. In particular:

- Applications that use HDFS are assumed to perform long sequential streaming reads from files. HDFS is optimized to provide streaming read performance; this comes at the expense of random seek times to arbitrary positions in files.
- 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. (An extension to Hadoop will provide support for appending new data to the ends of files; it is scheduled to be included in Hadoop 0.19 but is not available yet.)
- 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. The overhead of caching is great enough that data should simply be re-read from HDFS source.

- 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 (e.g., if a rack fails all together). While performance may degrade proportional to the number of machines lost, the system as a whole should not become overly slow, nor should information be lost. Data replication strategies combat this problem.

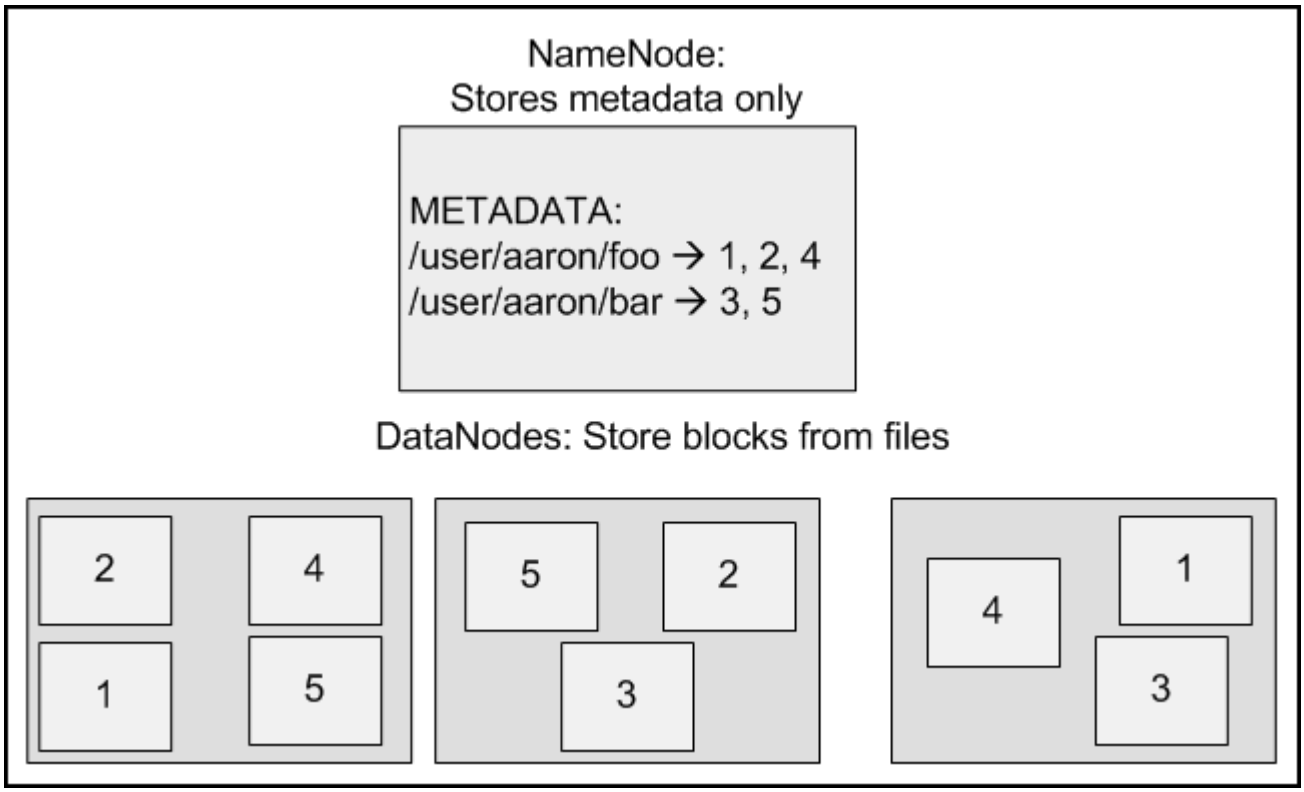


Fig: 1 DataNodes holding blocks of multiple files with a replication factor of 2. The NameNode maps the filenames onto the block ids.

HDFS Architecture

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 relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project.

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.

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.

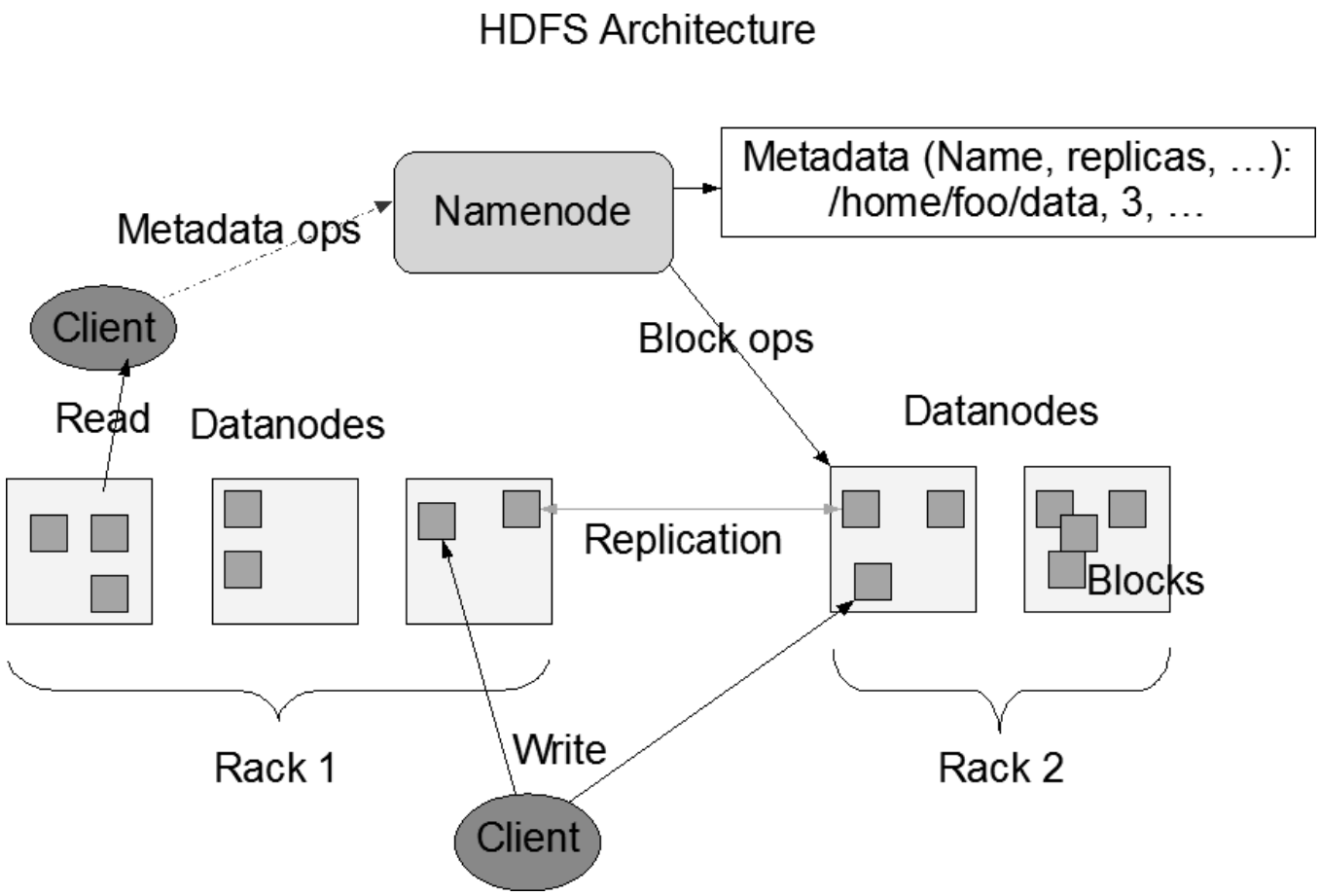
Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in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

HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed except for appends and truncates. Appending the content to the end of the files is supported but cannot be updated at arbitrary point. 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.

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.

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.



HDFS Architecture

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 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. The report is called the *Blockreport*. 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.

When a client is writing data to an HDFS file with a replication factor of three, the NameNode retrieves a list of DataNodes using a replication target choosing algorithm. This list contains the DataNodes that will host a replica of that block. The client then writes to the first DataNode. The first DataNode starts receiving the data in 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. Configuring HDFS

The HDFS for your cluster can be configured in a very short amount of time. First we will fill out the relevant sections of the Hadoop configuration file, then format the NameNode.

CLUSTER CONFIGURATION

The HDFS configuration is located in a set of XML files in the Hadoop configuration directory; conf/ under the main Hadoop install directory (where you unzipped Hadoop to). The conf/hadoop-defaults.xml file contains default values for every parameter in Hadoop. This file is considered read-only. You override this configuration by setting new values in conf/hadoop-site.xml. This file should be replicated consistently across all machines in the cluster. (It is also possible, though not advisable, to host it on NFS.)

Configuration settings are a set of key-value pairs of the format:

```
<property>
  <name>property-name</name>
  <value>property-value</value>

</property>
```

Adding the line <final>true</final> inside the property body will prevent properties from being overridden by user applications. This is useful for most system-wide configuration options.

The following settings are necessary to configure HDFS:

key	value	example
fs.default.name	*protocol*://*servername*:*port*	hdfs://alpha.milkman.org:9000
dfs.data.dir	*pathname*	/home/*username*/hdfs/data
dfs.name.dir	*pathname*	/home/*username*/hdfs/name

These settings are described individually below:

fs.default.name - This is the URI (protocol specifier, hostname, and port) that

describes the NameNode for the cluster. Each node in the system on which Hadoop is expected to operate needs to know the address of the NameNode. The DataNode instances will register with this NameNode, and make their data available through it. Individual client programs will connect to this address to retrieve the locations of actual file blocks.

dfs.data.dir - This is the path on the local file system in which the DataNode instance should store its data. It is not necessary that all DataNode instances store their data under the same local path prefix, as they will all be on separate machines; it is acceptable that these machines are heterogeneous. However, it will simplify configuration if this directory is standardized throughout the system. By default, Hadoop will place this under /tmp. This is fine for testing purposes, but is an easy way to lose actual data in a production system, and thus must be overridden.

dfs.name.dir - This is the path on the local file system of the NameNode instance where the NameNode metadata is stored. It is only used by the NameNode instance to find its information, and does not exist on the DataNodes. The caveat above about /tmp applies to this as well; this setting must be overridden in a production system.

Another configuration parameter, not listed above, is **dfs.replication**. This is the default replication factor for each block of data in the file system. For a production cluster, this should usually be left at its default value of 3. (You are free to increase your replication factor, though this may be unnecessary and use more space than is required. Fewer than three replicas impact the high availability of information, and possibly the reliability of its storage.)

The following information can be pasted into the hadoop-site.xml file for a single-node configuration:

```
<configuration>
  <property>
    <name>fs.default.name</name>

    <value>hdfs://your.server.name.com:9000</value>
  </property>
```

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

    <value>/home/username/hdfs/data</value>
  </property>
  <property>
    <name>dfs.name.dir</name>

    <value>/home/username/hdfs/name</value>
  </property>
</configuration>
```

Of course, your.server.name.com needs to be changed, as does *username*. Using port 9000 for the NameNode is arbitrary.

After copying this information into your conf/hadoop-site.xml file, copy this to the conf/ directories on all machines in the cluster.

The master node needs to know the addresses of all the machines to use as DataNodes; the startup scripts depend on this. Also in the conf/ directory, edit the file slaves so that it contains a list of fully-qualified hostnames for the slave instances, one host per line. On a multi-node setup, the master node (e.g., localhost) is not usually present in this file.

Then make the directories necessary:

```
user@EachMachine$ mkdir -p $HOME/hdfs/data

user@namenode$ mkdir -p $HOME/hdfs/name
```

The user who owns the Hadoop instances will need to have read and write access

to each of these directories. It is not necessary for all users to have access to these directories. Set permissions with chmod as appropriate. In a large-scale environment, it is recommended that you create a user named "hadoop" on each node for the express purpose of owning and running Hadoop tasks. For a single individual's machine, it is perfectly acceptable to run Hadoop under your own username. It is not recommended that you run Hadoop as root.

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. When the NameNode starts up, or a checkpoint is triggered by a configurable threshold, 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. The purpose of a checkpoint is to make sure that HDFS has a consistent view of the file system metadata by taking a snapshot of the file system metadata and saving it to FsImage. Even though it is efficient to read a FsImage, it is not efficient to make incremental edits directly to a FsImage. Instead of modifying FsImage for each edit, we persist the edits in the Editlog. During the checkpoint the changes from Editlog are applied to the FsImage. A checkpoint can be triggered at a given time interval (dfs.namenode.checkpoint.period) expressed in seconds, or after a given number of filesystem transactions have accumulated (dfs.namenode.checkpoint.txns). If both of these properties are set, the first threshold to be reached triggers a checkpoint.

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. The report is called 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.

Using HDFS Programmatically

While HDFS can be manipulated explicitly through user commands, or implicitly as the input to or output from a Hadoop MapReduce job, you can also work with HDFS inside your own Java applications.

```
1: import java.io.File;

2: import java.io.IOException;

3:

4: import org.apache.hadoop.conf.Configuration;

5: import org.apache.hadoop.fs.FileSystem;

6: import org.apache.hadoop.fs.FSDataInputStream;

7: import org.apache.hadoop.fs.FSDataOutputStream;
```

```
8: import org.apache.hadoop.fs.Path;
9:
10: public class HDFSHelloWorld {
11:
12:   public static final String theFilename = "hello.txt";
13:   public static final String message = "Hello, world!\n";
14:
15:   public static void main (String [] args) throws IOException {
16:
17:     Configuration conf = new Configuration();
18:     FileSystem fs = FileSystem.get(conf);
19:
20:     Path filenamePath = new Path(theFilename);
21:
22:     try {
23:       if (fs.exists(filenamePath)) {
24:         // remove the file first
25:         fs.delete(filenamePath);
26:       }
27:
28:       FSDataOutputStream out = fs.create(filenamePath);
29:       out.writeUTF(message;
30:       out.close();
```

```
31:
32:       FSDataInputStream in = fs.open(filenamePath);
33:       String messageIn = in.readUTF();
34:       System.out.print(messageIn);
35:       in.close();
46:     } catch (IOException ioe) {
47:       System.err.println("IOException during operation: " + ioe.toString());
48:       System.exit(1);
49:     }
40:   }
41: }
```

This program creates a file named hello.txt, writes a short message into it, then reads it back and prints it to the screen. If the file already existed, it is deleted first.

First we get a handle to an abstract FileSystem object, as specified by the application configuration. The Configuration object created uses the default parameters.

```
17:     Configuration conf = new Configuration();
18:     FileSystem fs = FileSystem.get(conf);
```

The FileSystem interface actually provides a generic abstraction suitable for use in several file systems. Depending on the Hadoop configuration, this may use HDFS or the local file system or a different one altogether. If this test program is launched via the ordinary 'java *classname*' command line, it may not find conf/hadoop-site.xml and will use the local file system. To ensure that it uses the proper Hadoop configuration, launch this program through Hadoop by putting it in a jar and running:

$HADOOP_HOME/bin/hadoop jar *yourjar* HDFSHelloWorld

Regardless of how you launch the program and which file system it connects to, writing to a file is done in the same way:

```
28:     FSDataOutputStream out = fs.create(filenamePath);

29:     out.writeUTF(message);

30:     out.close();
```

First we create the file with the fs.create() call, which returns an FSDataOutputStream used to write data into the file. We then write the information using ordinary stream writing functions; FSDataOutputStreamextends the java.io.DataOutputStream class. When we are done with the file, we close the stream with out.close().

This call to fs.create() will overwrite the file if it already exists, but for sake of example, this program explicitly removes the file first anyway (note that depending on this explicit prior removal is technically a race condition). Testing for whether a file exists and removing an existing file are performed by lines 23-26:

```
23:     if (fs.exists(filenamePath)) {

24:       // remove the file first

25:       fs.delete(filenamePath);

26:     }
```

Other operations such as copying, moving, and renaming are equally straightforward operations on Pathobjects performed by the FileSystem.

Finally, we re-open the file for read, and pull the bytes from the file, converting them to a UTF-8 encoded string in the process, and print to the screen:

```
32:     FSDataInputStream in = fs.open(filenamePath);

33:     String messageIn = in.readUTF();

34:     System.out.print(messageIn);

35:     in.close();
```

The fs.open() method returns an FSDataInputStream, which subclasses java.io.DataInputStream. Data can be read from the stream using the readUTF() operation, as on line 33. When we are done with the stream, we call close() to free the handle associated with the file.

Replica Statement

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 the local machine if the writer is on a datanode, otherwise on a random datanode, another replica 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.

If the replication factor is greater than 3, the placement of the 4th and following replicas are determined randomly while keeping the number of replicas per rack below the upper limit (which is basically (replicas - 1) / racks + 2).

Because the NameNode does not allow DataNodes to have multiple replicas of the same block, maximum number of replicas created is the total number of DataNodes at that time.

After the support for Storage Types and Storage Policies was added to HDFS, the NameNode takes the policy into account for replica placement in addition to the rack awareness described above. The NameNode chooses nodes based on rack awareness at first, then checks that the candidate node have storage required by the policy associated with the file. If the candidate node does not have the storage type, the NameNode looks for another node. If enough nodes to place replicas can not be found in the first path, the NameNode looks for nodes having fallback storage types in the second path.

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 HDFS cluster spans multiple data centers, then a replica that is resident in the local data center is preferred over any remote replica.

Unit 3

MapReduce is a programming model designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks. MapReduce programs are written in a particular style influenced by *functional programming* constructs, specifically idioms for processing lists of data. Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

A MapReduce *job* usually splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the *reduce tasks*. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks. Typically the compute nodes and the storage nodes are the same, that is, the MapReduce framework and the Hadoop Distributed File System are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

The MapReduce framework consists of a single master ResourceManager, one slave NodeManager per cluster-node, and MRAppMaster per application Minimally, applications specify the input/output locations and supply *map* and *reduce* functions via implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the *job configuration*.

The Hadoop *job client* then submits the job (jar/executable etc.) and configuration to the ResourceManager which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

MapReduce Basics

MapReduce programs are designed to compute large volumes of data in a parallel fashion. This requires dividing the workload across a large number of machines. This model would not scale to large clusters (hundreds or thousands of nodes) if the components were allowed to share data arbitrarily. The communication overhead required to keep the data on the nodes synchronized at all times would prevent the system from performing reliably or efficiently at large scale.

Instead, all data elements in MapReduce are *immutable*, meaning that they cannot be updated. If in a mapping task you change an input (key, value) pair, it does not get reflected back in the input files; communication occurs only by generating new output (key, value) pairs which are then forwarded by the Hadoop system into the next phase of execution.

MapReduce programs are designed to compute large volumes of data in a parallel fashion. This requires dividing the workload across a large number of machines. This model would not scale to large clusters (hundreds or thousands of nodes) if the components were allowed to share data arbitrarily. The communication overhead required to keep the data on the nodes synchronized at all times would prevent the system from performing reliably or efficiently at large scale.

Instead, all data elements in MapReduce are *immutable*, meaning that they cannot be updated. If in a mapping task you change an input (key, value) pair, it does not get reflected back in the input files; communication occurs only by generating new output (key, value) pairs which are then forwarded by the Hadoop system into the next phase of execution.

LIST PROCESSING

Conceptually, MapReduce programs transform lists of input data elements into lists of output data elements. A MapReduce program will do this twice, using two different list processing idioms: *map*, and *reduce*. These terms are taken from several list processing languages such as LISP, Scheme, or ML.

Inputs and Outputs

The MapReduce framework operates exclusively on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.

The key and value classes have to be serializable by the framework and hence need to implement the Writable interface. Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework.

Input and Output types of a MapReduce job:

(input) <k1, v1> -> **map** -> <k2, v2> -> **combine** -> <k2, v2> -> **reduce** -> <k3, v3> (output)

Word count program using Map Reduce is as follows:

```
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;
import org.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);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Applications typically implement the Mapper and Reducer interfaces to provide the map and reduce methods. These form the core of the job. Mapper maps input key/value pairs to a set of intermediate key/value pairs. Maps are the individual tasks that transform input records into intermediate records. The transformed intermediate records do not need to be of the same type as the input records. A given input pair may map to zero or many output pairs. The Hadoop MapReduce framework spawns one map task for each InputSplit generated by the InputFormat for the job. Overall, mapper implementations are passed to the job via Job.setMapperClass (Class) method. The framework then calls map(WritableComparable, Writable, Context) for each key/value pair in the InputSplit for that task. Applications can then override

the cleanup(Context) method to perform any required cleanup.

Output pairs do not need to be of the same types as input pairs. A given input pair may map to zero or many output pairs. Output pairs are collected with calls to context.write(WritableComparable, Writable).Applications can use the Counter to report its statistics. All intermediate values associated with a given output key are subsequently grouped by the framework, and passed to the Reducer(s) to determine the final output. Users can control the grouping by specifying a Comparator Comparator via Job.setGroupingComparatorClass (Class). The Mapper outputs are sorted and then partitioned per Reducer. The total number of partitions is the same as the number of reduce tasks for the job. Users can control which keys (and hence records) go to which Reducer by implementing a custom Partitioner. Users can optionally specify a combiner, via Job.setCombinerClass(Class), to perform local aggregation of the intermediate outputs, which helps to cut down the amount of data transferred from the Mapper to the Reducer. The intermediate, sorted outputs are always stored in a simple (key-len, key, value-len, value) format. Applications can control if, and how, the intermediate outputs are to be compressed and the CompressionCodec to be used via the Configuration.

Reducer reduces a set of intermediate values which share a key to a smaller set of values.

The number of reduces for the job is set by the user via Job.setNumReduceTasks (int).

Overall, Reducer implementations are passed the Job for the job via the Job.setReducerClass(Class) method and can override it to initialize themselves. The framework then calls reduce (WritableComparable, Iterable<Writable>, Context) method for each <key, (list of values)> pair in the grouped inputs. Applications can then override the cleanup(Context) method to perform any required cleanup.

Reducer has 3 primary phases: shuffle, sort and reduce. Input to the Reducer is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.

If equivalence rules for grouping the intermediate keys are required to be different from those for grouping keys before reduction, then one may specify a Comparator via Job.setSortComparatorClass (Class).

Since Job.setGroupingComparatorClass (Class) can be used to control how intermediate keys are grouped, these can be used in conjunction to simulate *secondary sort on values*.

In this phase the reduce(WritableComparable, Iterable<Writable>, Context) method is called for each <key, (list of values)> pair in the grouped inputs. The output of the reduce task is typically written to the FileSystem via Context.write(WritableComparable, Writable).Applications can use the Counter to report its statistics.The output of the Reducer is *not sorted*.

The right number of reduces seems to be 0.95 or 1.75 multiplied by (<*no. of nodes*> * <*no. of maximum containers per node*>).With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish. With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.Increasing the number of reduces increases the framework overhead, but increases load balancing and lowers the cost of failures.

The scaling factors above are slightly less than whole numbers to reserve a few reduce slots in the framework for speculative-tasks and failed tasks.

It is legal to set the number of reduce-tasks to *zero* if no reduction is desired. In this case the outputs of the map-tasks go directly to the FileSystem, into the output path set by FileOutputFormat.setOutputPath (Job, Path). The framework does not sort the map-outputs before writing them out to the FileSystem.

Partitioner partitions the key space. Partitioner controls the partitioning of the keys of the intermediate map-outputs. The key (or a subset of the key) is used to derive

the partition, typically by a *hash function*. The total number of partitions is the same as the number of reduce tasks for the job. Hence this controls which of the m reduce tasks the intermediate key (and hence the record) is sent to for reduction. HashPartitioner is the default Partitioner. Counter is a facility for MapReduce applications to report its statistics.

Mapper and Reducer implementations can use the Counter to report statistics. Hadoop MapReduce comes bundled with a library of generally useful mappers, reducers, and partitioners. Job represents a MapReduce job configuration.Job is the primary interface for a user to describe a MapReduce job to the Hadoop framework for execution. The framework tries to faithfully execute the job as described by Job, however:

- Some configuration parameters may have been marked as final by administrators and hence cannot be altered.
- While some job parameters are straight-forward to set (e.g. Job.setNumReduceTasks (int)) , other parameters interact subtly with the rest of the framework and/or job configuration and are more complex to set (e.g. Configuration. Set(JobContext.NUM_MAPS, int)).

Optionally, Job is used to specify other advanced facets of the job such as the Comparator to be used, files to be put in the DistributedCache, whether intermediate and/or job outputs are to be compressed (and how), whether job tasks can be executed in a *speculative* manner Of course, users can use Configuration.set(String, String)/ Configuration.get(String) to set/get arbitrary parameters needed by applications. However, use the DistributedCache for large amounts of (read-only) data.

The MRAppMaster executes the Mapper/Reducer *task* as a child process in a separate jvm.

The child-task inherits the environment of the parent MRAppMaster. The user can

specify additional options to the child-jvm via the mapreduce.{map|reduce}.java.opts and configuration parameter in the Job such as non-standard paths for the run-time linker to search shared libraries via -Djava.library.path=<> etc. If the mapreduce.{map|reduce}.java.opts parameters contains the symbol *@taskid@* it is interpolated with value of taskid of the MapReduce task.

Here is an example with multiple arguments and substitutions, showing jvm GC logging, and start of a passwordless JVM JMX agent so that it can connect with jconsole and the likes to watch child memory, threads and get thread dumps. It also sets the maximum heap-size of the map and reduce child jvm to 512MB & 1024MB respectively. It also adds an additional path to the java.library.path of the child-jvm.

```
<property>
  <name>mapreduce.map.java.opts</name>
  <value>
  -Xmx512M -Djava.library.path=/home/mycompany/lib -verbose:gc -
Xloggc:/tmp/@taskid@.gc
  -Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.ssl=false
  </value>
</property>

<property>
  <name>mapreduce.reduce.java.opts</name>
  <value>
  -Xmx1024M -Djava.library.path=/home/mycompany/lib -verbose:gc -
Xloggc:/tmp/@taskid@.gc
  -Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.ssl=false
  </value>
</property>
```

Users/admins can also specify the maximum virtual memory of the launched child-task, and any sub-process it launches recursively, using mapreduce memory.mb. Note that the value set here is a per process limit. The value for mapreduce.{map|reduce}. memory.mb should be specified in mega

bytes (MB). And also the value must be greater than or equal to the -Xmx passed to JavaVM, else the VM might not start.

Note: mapreduce.{map|reduce}.java.opts are used only for configuring the launched child tasks from MRAppMaster. The memory available to some parts of the framework is also configurable. In map and reduce tasks, performance may be influenced by adjusting parameters influencing the concurrency of operations and the frequency with which data will hit disk. Monitoring the filesystem counters for a job- particularly relative to byte counts from the map and into the reduce- is invaluable to the tuning of these parameters.

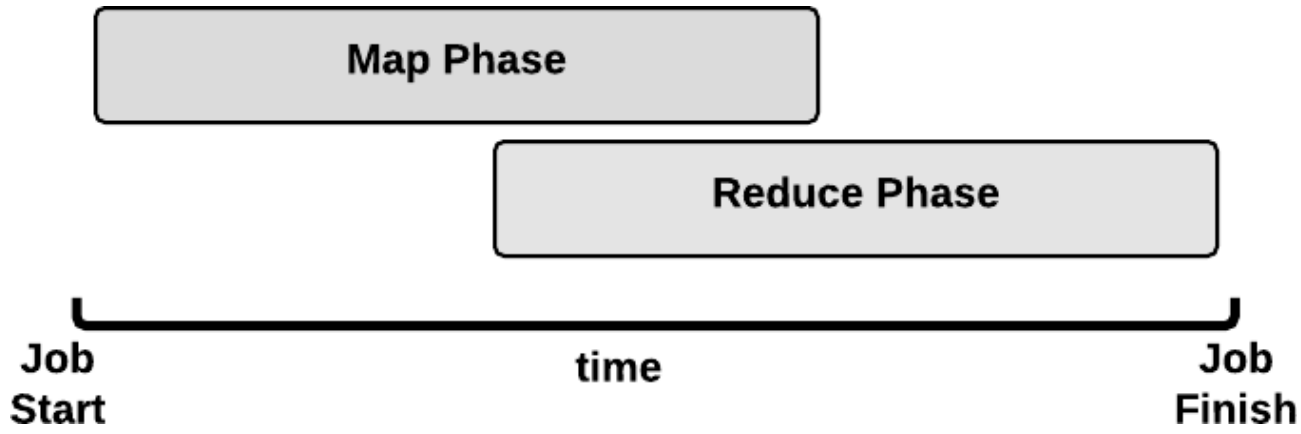
A record emitted from a map will be serialized into a buffer and metadata will be stored into accounting buffers. As described in the following options, when either the serialization buffer or the metadata exceed a threshold, the contents of the buffers will be sorted and written to disk in the background while the map continues to output records. If either buffer fills completely while the spill is in progress, the map thread will block. When the map is finished, any remaining records are written to disk and all on-disk segments are merged into a single file. Minimizing the number of spills to disk can decrease map time, but a larger buffer also decreases the memory available to the mapper.

Unit 4

Anatomy of a MapReduce Job

In MapReduce, a YARN application is called a **Job**. The implementation of the Application Master provided by the MapReduce framework is called **MRAppMaster**.

Timeline of a MapReduce Job



This is the timeline of a MapReduce Job execution:

- **Map Phase**: several **Map Tasks** are executed
- **Reduce Phase**: several **Reduce Tasks** are executed

Notice that the Reduce Phase may start before the end of Map Phase. Hence, an interleaving between them is possible.

Map Phase

We now focus our discussion on the Map Phase. A key decision is how many MapTasks the Application Master needs to start for the current job.

What does the user give us?

Let's take a step back. When a client submits an application, several kinds of information are provided to the YARN infrastucture. In particular:

- a configuration: this may be partial (some parameters are not specified by the user) and in this case the default values are used for

the job. Notice that these default values may be the ones chosen by a Hadoop provider like Amanzon.

- a JAR containing:
  - a map() implementation
  - a combiner implementation
  - a reduce() implementation
- input and output information:
  - input directory: is the input directory on HDFS? On S3? **How many files?**
  - output directory: where will we store the output? On HDFS? On S3?

The number of files inside the input directory is used for deciding the number of Map Tasks of a job.

How many Map Tasks?

The Application Master will launch one MapTask for each map split. Typically, there is a map split for each input file. If the input file is too big (bigger than the HDFS block size) then we have two or more map splits associated to the same input file. This is the pseudocode used inside the method getSplits() of the FileInputFormat class:

```
num_splits = 0
for each input file f:
   remaining = f.length
   while remaining / split_size > split_slope:
      num_splits += 1
      remaining -= split_size
```

where:

```
split_slope = 1.1
split_size =~ dfs.blocksize
```

Notice that the configuration parameter mapreduce.job.maps is ignored in MRv2 (in the past it was just an hint).

MapTask Launch

The MapReduce Application Master asks to the Resource Manager for Containers needed by the Job: one MapTask container request for each MapTask (map split).

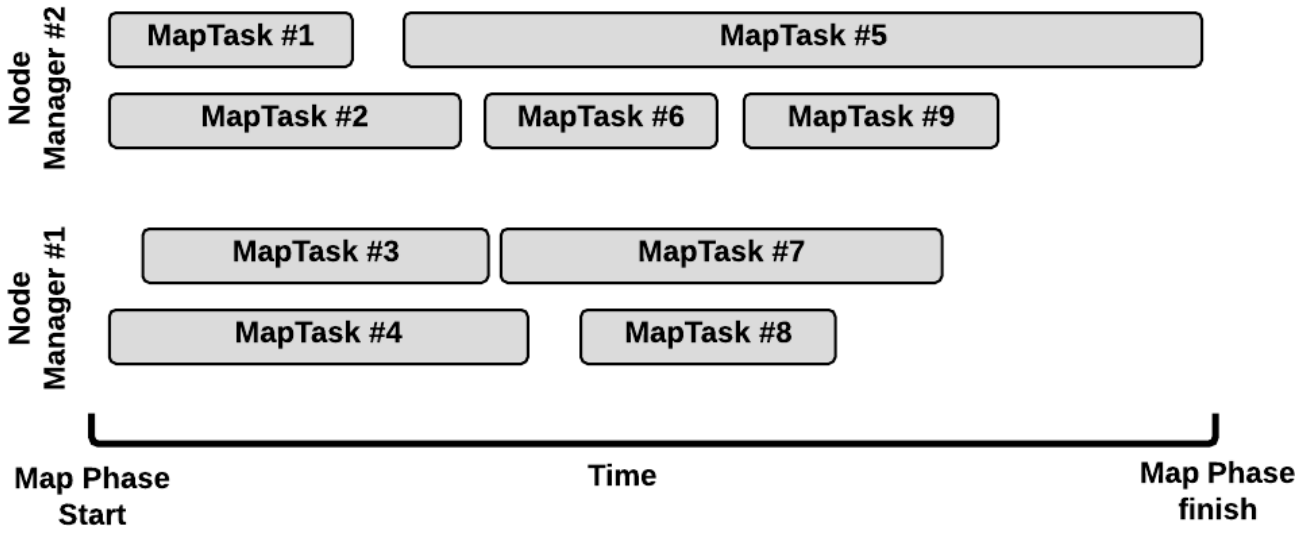
A container request for a MapTask tries to exploit data locality of the map split. The Application Master asks for:

- a container located on the same Node Manager where the map split is stored (a map split may be stored on multiple nodes due to the HDFS replication factor);
- otherwise, a container located on a Node Manager in the same rack where the the map split is stored;
- otherwise, a container on any other Node Manager of the cluster

This is just an hint to the Resource Scheduler. The Resource Scheduler is free to ignore data locality if the suggested assignment is in conflict with the Resouce Scheduler's goal.

When a Container is assigned to the Application Master, the MapTask is launched.

Map Phase: example of an execution scenario



This is a possible execution scenario of the Map Phase:

- there are two Node Managers: each Node Manager has 2GB of RAM (NM capacity) and each MapTask requires 1GB, we can run in parallel

2 containers on each Node Manager (this is the best scenario, the Resource Scheduler may decide differently)

- there are no other YARN applications running in the cluster
- our job has 8 map splits (e.g., there are 7 files inside the input directory, but only one of them is bigger than the HDFS block size so we split it into 2 map splits): we need to run 8 Map Tasks.

Map Task Execution Timeline

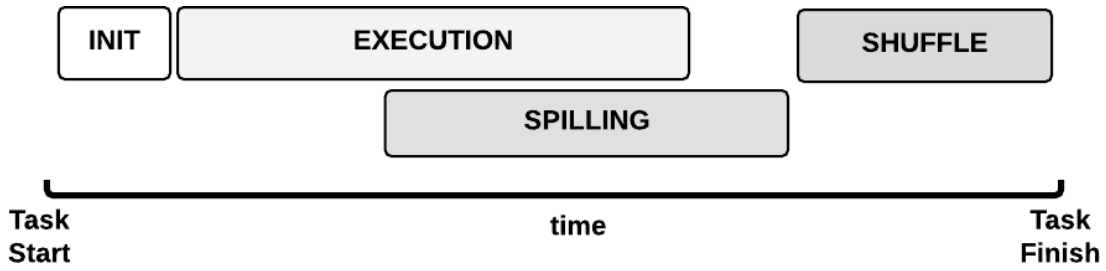


Fig. Map Task execution timeline:

- **INIT** phase: we setup the Map Task
- **EXECUTION** phase: for each (key, value) tuple inside the map split we run the `map()` function
- **SPILLING** phase: the map output is stored in an in-memory buffer; when this buffer is *almost* full then we start (in parallel) the spilling phase in order to remove data from it
- **SHUFFLE** phase: at the end of the spilling phase, we merge all the map outputs and package them for the reduce phase

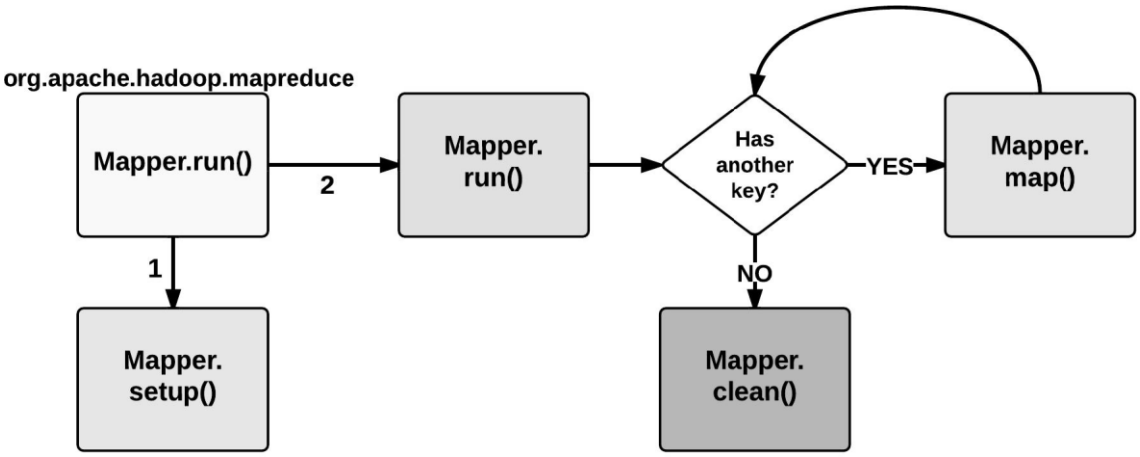
MapTask: INIT

During the INIT phase, we:

1. create a context (TaskAttemptContext.class)
2. create an instance of the user Mapper.class
3. setup the input (e.g., InputFormat.class, InputSplit.class, RecordReader.class)
4. setup the output (NewOutputCollector.class)
5. create a mapper context (MapContext.class, Mapper.Context.class)

6. initialize the input, e.g.:
7. create a SplitLineReader.class object
8. create a HdfsDataInputStream.class object

MapTask: EXECUTION

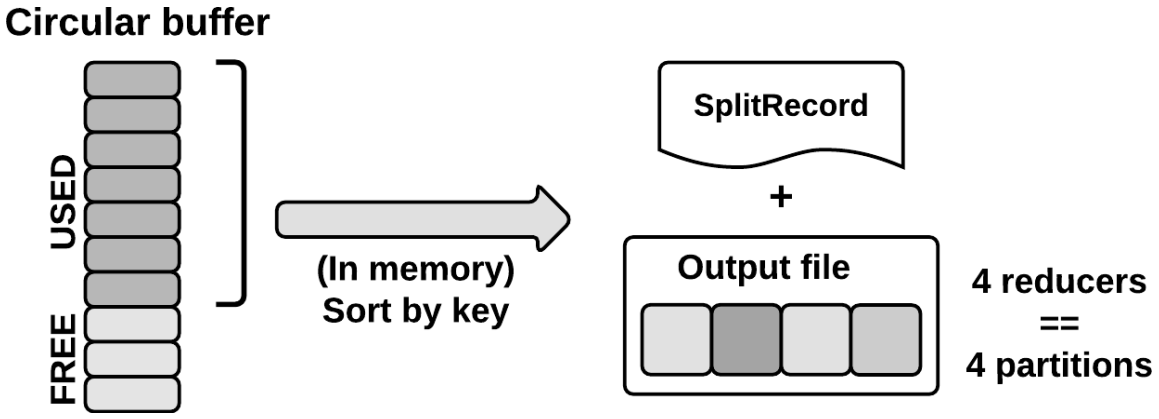


The EXECUTION phase is performed by the run method of the Mapper class. The user can override it, but by default it will start by calling the setup method: this function by default does not do anything useful but can be override by the user in order to setup the Task (e.g., initialize class variables). After the setup, for each <key, value> tuple contained in the map split, the map() is invoked. Therefore, map() receives: a key a value, and a mapper context. Using the context, a mapstores its output to a buffer.

Notice that the map split is fetched chuck by chunk (e.g., 64KB) and each chunk is split in several (key, value) tuples (e.g., using SplitLineReader.class).

This is done inside the Mapper.Context.nextKeyValue method. When the map split has been completely processed, the run function calls the clean method: by default, no action is performed but the user may decide to override it.

MapTask: SPILLING



As seen in the EXECUTING phase, the map will write (using Mapper.Context.write()) its output into a circular in-memory buffer (MapTask.MapOutputBuffer). The size of this buffer is fixed and determined by the configuration parameter mapreduce.task.io.sort.mb (default: 100MB).

Whenever this circular buffer is *almost* full (mapreduce.map.sort.spill.percent: 80% by default), the SPILLING phase is performed (in parallel using a separate thread). Notice that if the splilling thread is too slow and the buffer is 100% full, then the map() cannot be executed and thus it has to wait.

The SPILLING thread performs the following actions:

1. it creates a SpillRecord and FSOutputStream (local filesystem)
2. in-memory sorts the used chunk of the buffer: the output tuples are sorted by (partitionIdx, key) using a quicksort algorithm.
3. the sorted output is split into partitions: one partition for each ReduceTask of the job (see later).
4. Partitions are sequentially written into the local file.

How Many Reduce Tasks?

The number of ReduceTasks for the job is decided by the configuration parameter `mapreduce.job.reduces`.

What is the partitionIdx associated to an output tuple?

The paritionIdx of an output tuple is the index of a partition. It is decided inside the `Mapper.Context.write()`:

```
partitionIdx = (key.hashCode() & Integer.MAX_VALUE) % numReducers
```

It is stored as metadata in the circular buffer alongside the output tuple. The user can customize the partitioner by setting the configuration parameter `mapreduce.job.partitioner.class`.

When do we apply the combiner?

If the user specifies a combiner then the SPILLING thread, before writing the tuples to the file (4), executes the combiner on the tuples contained in each partition. Basically, we:

1. create an instance of the user `Reducer.class` (the one specified for the combiner!)
2. create a `Reducer.Context`: the output will be stored on the local filesystem
3. execute `Reduce.run()`: see Reduce Task description

The combiner typically use the same implementation of the standard `reduce()` function and thus can be seen as a local reducer.

MapTask: end of EXECUTION

At the end of the EXECUTION phase, the SPILLING thread is triggered for the last time. In more detail, we:

1. sort and spill the remaining unspilled tuples

2. start the SHUFFLE phase

Notice that for each time the buffer was almost full, we get one spill file (SpillReciord + output file). Each Spill file contains several partitions (segments).

MapReduce Performance Tuning Tutorial

Performance tuning in **Hadoop** will help in optimizing the Hadoop cluster performance. This tutorial on Hadoop MapReduce performance tuning will provide you ways for improving your Hadoop cluster performance and get the best result from your programming in Hadoop. It will cover 7 important concepts like Memory Tuning in Hadoop, Map Disk spill in Hadoop, tuning mapper tasks, Speculative execution in **Big data** hadoop and many other related concepts for Hadoop MapReduce performance tuning.

Hadoop MapReduce Performance Tuning

Hadoop performance tuning will help you in optimizing your Hadoop cluster performance and make it better to provide best results while doing Hadoop programming in **Big Data** companies. To perform the same, you need to repeat the process given below till desired output is achieved at optimal way.

The first step in hadoop performance tuning is to run Hadoop job, Identify the bottlenecks and address them using below methods to get the highest performance. You need to repeat above step till a level of performance is achieved.

Tuning Hadoop Run-time Parameters

There are many options provided by Hadoop on CPU, memory, disk, and network for performance tuning. Most Hadoop tasks are not CPU bounded, what is most considered is to optimize usage of memory and disk spills. Let us get into the details in this Hadoop performance tuning in Tuning Hadoop Run-time parameters.

Memory Tuning

The most general and common rule for memory tuning in MapReduce performance tuning is: use as much memory as you can without triggering swapping. The parameter for task memory is ***mapred.child.java.opts*** that can be put in your configuration file.

You can also monitor memory usage on the server using Ganglia, **Cloudera** manager, or Nagios for better memory performance.

Minimize the Map Disk Spill

Disk IO is usually the performance bottleneck in Hadoop. There are a lot of parameters you can tune for minimizing spilling like:

- Compression of mapper output
- Usage of 70% of heap memory ion mapper for spill buffer

But do you think frequent spilling is a good idea?

It's highly suggested not to spill more than once as if you spill once, you need to re-read and re-write all data: 3x the IO.

Tuning Mapper Tasks

The number of **mapper** tasks is set implicitly unlike **reducer** tasks. The most common hadoop performance tuning way for the mapper is controlling the amount of mapper and the size of each job. When dealing with large files, Hadoop split the file into smaller chunks so that mapper can run it in parallel. However, initializing new mapper job usually takes few seconds that is also an overhead to be minimized. Below are the suggestions for the same:

- Reuse jvm task
- Aim for map tasks running 1-3 minutes each. For this if the average mapper running time is lesser than one minute, increase the ***mapred.min.split.size***, to allocate less mappers in slot and thus reduce the mapper initializing overhead.

- Use Combine file input format for bunch of smaller files.

Tuning Application Specific Performance

Let's now discuss the tips to improve the Application specific performance in Hadoop.

Minimize your Mapper Output

Minimizing the mapper output can improve the general performance a lot as this is sensitive to disk IO, network IO, and memory sensitivity on shuffle phase.

For achieving this, below are the suggestions:

- Filter the records on mapper side instead of reducer side.
- Use minimal data to form your map output key and map output value in Map Reduce.
- Compress mapper output

Balancing Reducer's Loading

Unbalanced reducer tasks create another performance issue. Some reducers take most of the output from mapper and ran extremely long compare to other reducers.

Below are the methods to do the same:

- Implement a better hash function in Partitioner class.
- Write a preprocess job to separate keys using MultipleOutputs. Then use another map-reduce job to process the special keys that cause the problem.

When tasks take long time to finish the execution, it affects the MapReduce jobs. This problem is being solved by the approach of speculative execution by backing up slow tasks on alternate machines. You need to set the configuration parameters '***mapreduce.map.tasks.speculative.execution***' and '***mapreduce.reduce.tasks.speculative.execution***' to true for enabling speculative execution. This will reduce the job execution time if the task progress is slow due to memory unavailability.

Unit-5

Apache Mahout is an open source project that is primarily used in producing scalable machine learning algorithms. 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:

- Recommendation
- Classification

- 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 Naive 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.

Apache Mahout is a highly scalable machine learning library that enables developers to use optimized algorithms. Mahout implements popular

machine learning techniques such as recommendation, classification, and clustering. Therefore, it is prudent to have a brief section on machine learning before we move further.

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

Recommendation

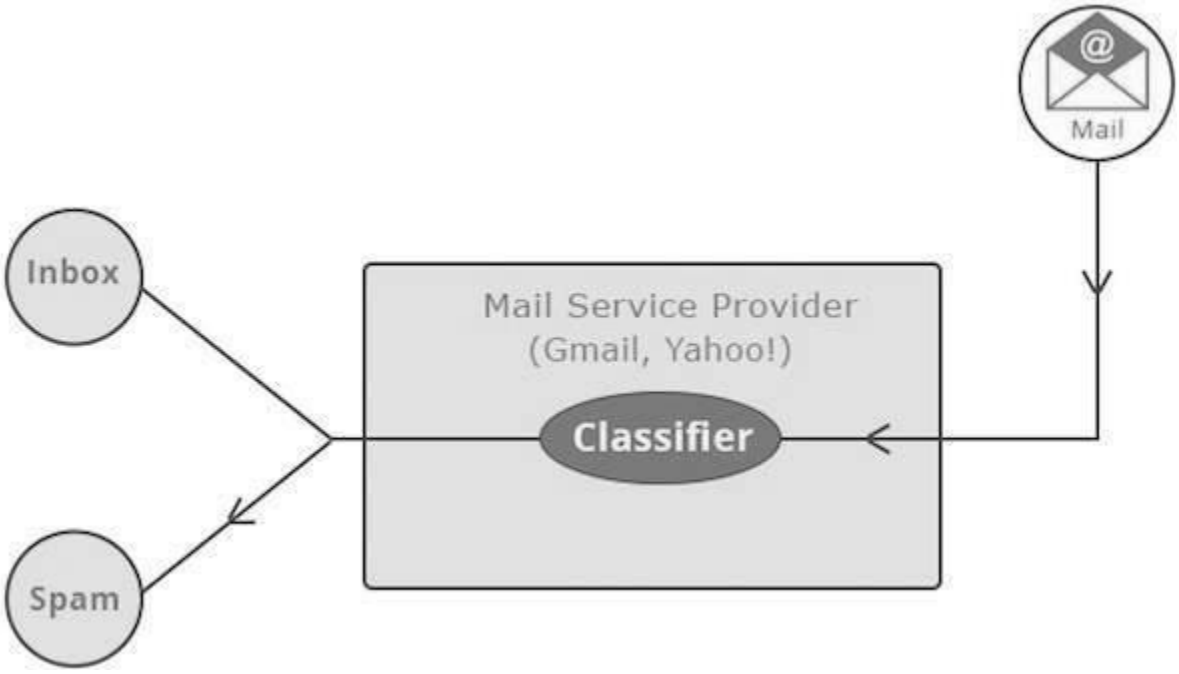
Recommendation is a popular technique that provides close recommendations based on user information such as previous purchases, clicks, and ratings.

- 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".

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.



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.

Java and Hadoop are the prerequisites of mahout. Below given are the steps to download and install Java, Hadoop, and Mahout.

Pre-Installation Setup

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

Creating a User

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

- Open root using the command "su".
- Create a user from the root account using the command **"useradd username"**.
- Now you can open an existing user account using the command **"su username"**.
- Open the Linux terminal and type the following commands to create a user.

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

SSH Setup and Key Generation

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

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

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

Verifying ssh

```
ssh localhost
```

Installing Java

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

```
$ java -version
```

It should produce the following output.

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

If you don't have Java installed in your system, then follow the steps given below for installing Java.

Step 1

Download java (JDK <latest version> - X64.tar.gz) by visiting the following link: Oracle

Then **jdk-7u71-linux-x64.tar.gz is downloaded** onto your system.

Step 2

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

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

Step 3

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

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

Step 4

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

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

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

Downloading Hadoop

After installing Java, you need to install Hadoop initially. Verify the existence of Hadoop using "Hadoop version" command as shown below.

```
hadoop version
```

It should produce the following output:

```
Hadoop 2.6.0
Compiled by jenkins on 2014-11-13T21:10Z
Compiled with protoc 2.5.0
From source with checksum 18e43357c8f927c0695f1e9522859d6a
This command was run using /home/hadoop/hadoop/share/hadoop/common/hadoopcommon-2.6.0.jar
```

If your system is unable to locate Hadoop, then download Hadoop and have it installed on your system. Follow the commands given below to do so.

Download and extract hadoop-2.6.0 from apache software foundation using the following commands.

```
$ su
password:
# cd /usr/local
# wget http://mirrors.advancedhosters.com/apache/hadoop/common/hadoop-
2.6.0/hadoop-2.6.0-src.tar.gz
# tar xzf hadoop-2.6.0-src.tar.gz
# mv hadoop-2.6.0/* hadoop/
# exit
```

Installing Hadoop

Install Hadoop in any of the required modes. Here, we are demonstrating HBase functionalities in pseudo-distributed mode, therefore install Hadoop in pseudo-distributed mode.

Follow the steps given below to install **Hadoop 2.4.1** on your system.

Step 1: Setting up Hadoop

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

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

export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native

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

Now, apply all changes into the currently running system.

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

Step 2: Hadoop Configuration

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

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

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

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

Given below are the list of files which you have to edit to configure Hadoop.

core-site.xml

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

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

```
<configuration>

   <property>

      <name>fs.default.name</name>

      <value>hdfs://localhost:9000</value>

   </property>

</configuration>
```

hdfs-site.xm

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

Let us assume the following data:

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

(In the below given path /hadoop/ is the user name.
hadoopinfra/hdfs/namenode is the directory created by hdfs file system.)
namenode path = //home/hadoop/hadoopinfra/hdfs/namenode

(hadoopinfra/hdfs/datanode is the directory created by hdfs file system.)
datanode path = //home/hadoop/hadoopinfra/hdfs/datanode
```

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

```
<configuration>

   <property>

      <name>dfs.replication</name>

      <value>1</value>

   </property>


   <property>

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

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

    <property>
        <name>dfs.data.dir</name>
        <value>file:///home/hadoop/hadoopinfra/hdfs/datanode</value>
    </property>
</configuration>
```

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

mapred-site.xml

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

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

mapred-site.xml

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

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

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

```
<configuration>

   <property>

      <name>mapreduce.framework.name</name>

      <value>yarn</value>

   </property>

</configuration>
```

Verifying Hadoop Installation

The following steps are used to verify the Hadoop installation.

Step 1: Name Node Setup

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

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

The expected result is as follows:

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

Step 2: Verifying Hadoop dfs

The following command is used to start dfs. This command starts your Hadoop file system.

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

The expected output is as follows:

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

Step 3: Verifying Yarn Script

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

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

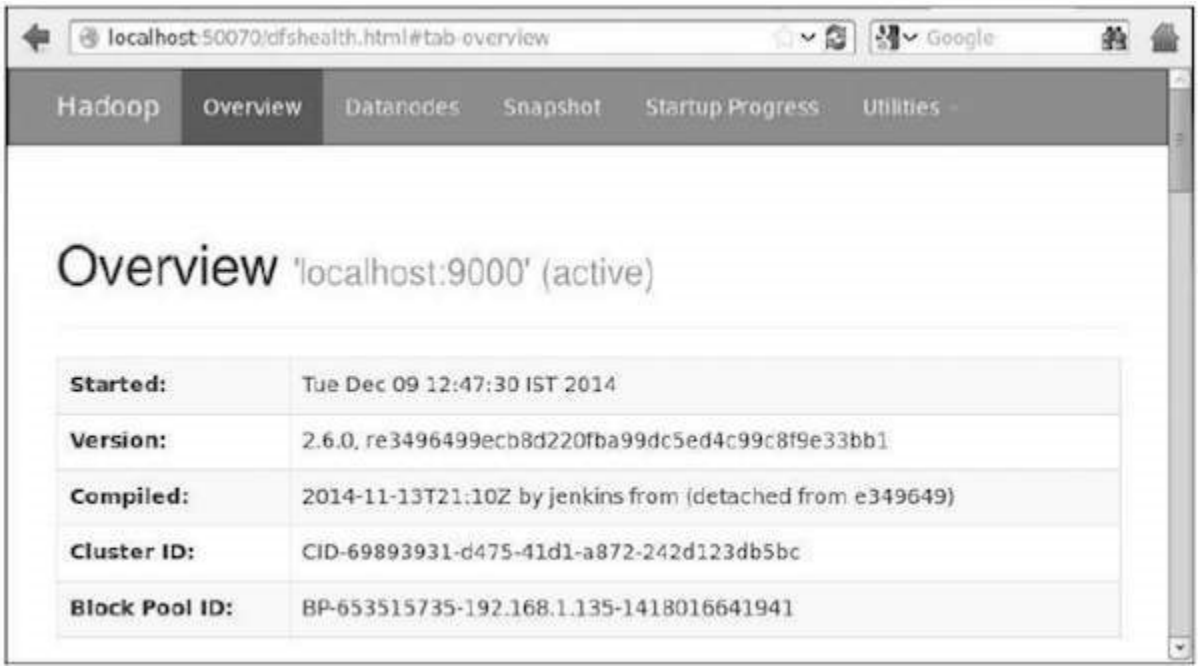
The expected output is as follows:

```
starting yarn daemons
starting resource manager, logging to /home/hadoop/hadoop-2.4.1/logs/yarn-
hadoop-resourcemanager-localhost.out
localhost: starting node manager, logging to /home/hadoop/hadoop-
2.4.1/logs/yarn-hadoop-nodemanager-localhost.out
```

Step 4: Accessing Hadoop on Browser

The default port number to access hadoop is 50070. Use the following URL to get Hadoop services on your browser.

```
http://localhost:50070/
```



Step 5: Verify All Applications for Cluster

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

```
http://localhost:8088/
```

Downloading Mahout

Mahout is available in the website Mahout. Download Mahout from the link provided in the website. Here is the screenshot of the website.

Step 1

Download Apache mahout from the link http://mirror.nexcess.net/apache/mahout/ using the following command.

```
[Hadoop@localhost ~]$ wget
http://mirror.nexcess.net/apache/mahout/0.9/mahout-distribution-0.9.tar.gz
```

Then **mahout-distribution-0.9.tar.gz** will be downloaded in your system.

Step2

Browse through the folder where **mahout-distribution-0.9.tar.gz** is stored and extract the downloaded jar file as shown below.

```
[Hadoop@localhost ~]$ tar zxvf mahout-distribution-0.9.tar.gz
```

Maven Repository

Given below is the pom.xml to build Apache Mahout using Eclipse.

```
<dependency>
    <groupId>org.apache.mahout</groupId>
    <artifactId>mahout-core</artifactId>
    <version>0.9</version>
</dependency>

<dependency>
    <groupId>org.apache.mahout</groupId>
    <artifactId>mahout-math</artifactId>
    <version>${mahout.version}</version>
</dependency>

<dependency>
    <groupId>org.apache.mahout</groupId>
    <artifactId>mahout-integration</artifactId>
    <version>${mahout.version}</version>
```

```
</dependency>
```

Mahout Recommender Engine

Mahout has a non-distributed, non-Hadoop-based recommender engine. You should pass a text document having user preferences for items. And the output of this engine would be the estimated preferences of a particular user for other items.

Example

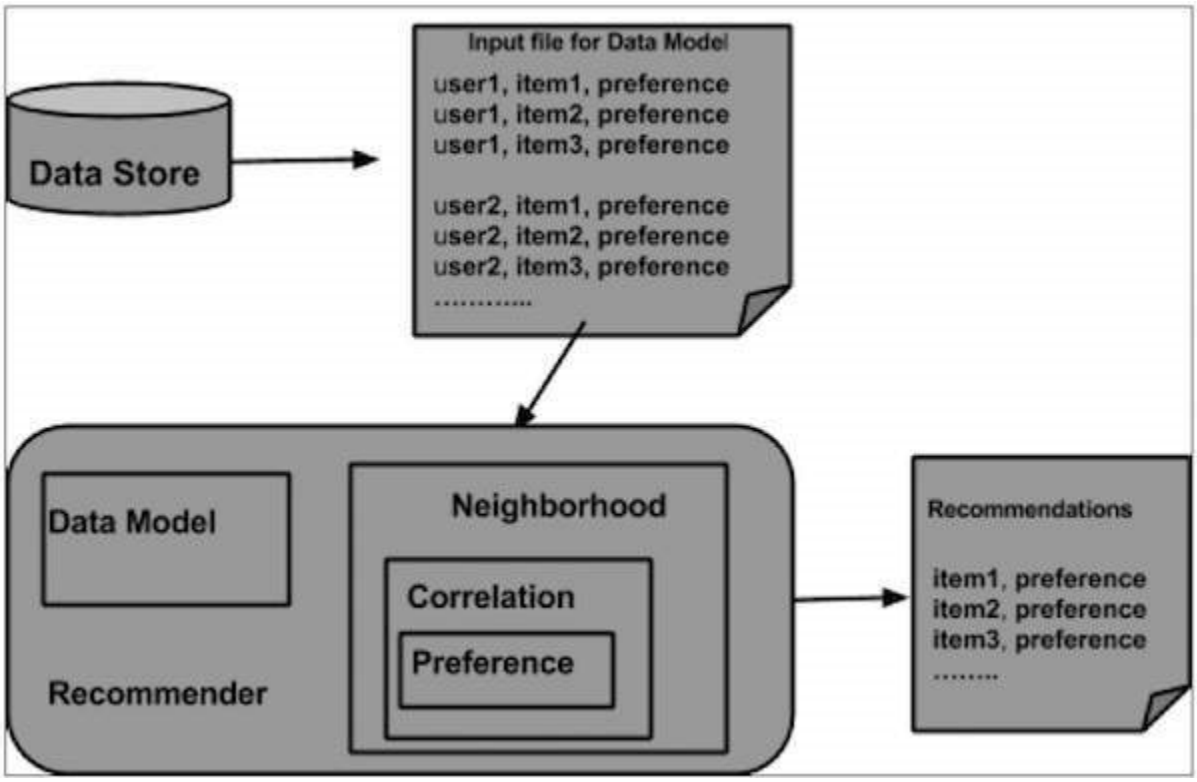
Consider a website that sells consumer goods such as mobiles, gadgets, and their accessories. If we want to implement the features of Mahout in such a site, then we can build a recommender engine. This engine analyzes past purchase data of the users and recommends new products based on that.

The components provided by Mahout to build a recommender engine are as follows:

- DataModel
- UserSimilarity
- ItemSimilarity
- UserNeighborhood
- Recommender

From the data store, the data model is prepared and is passed as an input to the recommender engine. The Recommender engine generates the recommendations for a particular user. Given below is the architecture of recommender engine.

Architecture of Recommender Engine



Building a Recommender using Mahout

Here are the steps to develop a simple recommender:

Step1: Create DataModel Object

The constructor of **PearsonCorrelationSimilarity** class requires a data model object, which holds a file that contains the Users, Items, and Preferences details of a product. Here is the sample data model file:

```
1,00,1.0
1,01,2.0
1,02,5.0
1,03,5.0
1,04,5.0

2,00,1.0
2,01,2.0
2,05,5.0
2,06,4.5
2,02,5.0

3,01,2.5
3,02,5.0
3,03,4.0
3,04,3.0

4,00,5.0
4,01,5.0
```

```
4,02,5.0
4,03,0.0
```

The **DataModel** object requires the file object, which contains the path of the input file. Create the **DataModel** object as shown below.

```
DataModel datamodel = new FileDataModel(new File("input file"));
```

Step2: Create UserSimilarity Object

Create **UserSimilarity** object using **PearsonCorrelationSimilarity** class as shown below:

```
UserSimilarity similarity = new PearsonCorrelationSimilarity(datamodel);
```

Step3: Create UserNeighborhood object

This object computes a "neighborhood" of users like a given user. There are two types of neighborhoods:

- **NearestNUserNeighborhood** - This class computes a neighborhood consisting of the nearest *n* users to a given user. "Nearest" is defined by the given UserSimilarity.
- **ThresholdUserNeighborhood** - This class computes a neighborhood consisting of all the users whose similarity to the given user meets or exceeds a certain threshold. Similarity is defined by the given UserSimilarity.

Here we are using **ThresholdUserNeighborhood** and set the limit of preference to 3.0.

```
UserNeighborhood neighborhood = new ThresholdUserNeighborhood(3.0, similarity, model);
```

Step4: Create Recommender Object

Create **UserbasedRecomender** object. Pass all the above created objects to its constructor as shown below.

```
UserBasedRecommender recommender = new GenericUserBasedRecommender(model, neighborhood,
similarity);
```

Step5: Recommend Items to a User

Recommend products to a user using the recommend() method of **Recommender** interface. This method requires two parameters. The first represents the user id of the user to whom we need to send the recommendations, and the second represents the number of

recommendations to be sent. Here is the usage of **recommender()** method:

```
List<RecommendedItem> recommendations = recommender.recommend(2, 3);

for (RecommendedItem recommendation : recommendations) {
   System.out.println(recommendation);
 }
```

Example Program

Given below is an example program to set recommendation. Prepare the recommendations for the user with user id 2.

```
import java.io.File;

import java.util.List;


import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;

import org.apache.mahout.cf.taste.impl.neighborhood.ThresholdUserNeighborhood;

import org.apache.mahout.cf.taste.impl.recommender.GenericUserBasedRecommender;

import org.apache.mahout.cf.taste.impl.similarity.PearsonCorrelationSimilarity;


import org.apache.mahout.cf.taste.model.DataModel;

import org.apache.mahout.cf.taste.neighborhood.UserNeighborhood;


import org.apache.mahout.cf.taste.recommender.RecommendedItem;

import org.apache.mahout.cf.taste.recommender.UserBasedRecommender;


import org.apache.mahout.cf.taste.similarity.UserSimilarity;


public class Recommender {

   public static void main(String args[]){

      try{
```

```
            //Creating data model

            DataModel datamodel = new FileDataModel(new File("data")); //data


            //Creating UserSimilarity object.

            UserSimilarity usersimilarity = new PearsonCorrelationSimilarity(datamodel);


            //Creating UserNeighbourHHood object.

            UserNeighborhood userneighborhood = new ThresholdUserNeighborhood(3.0,
usersimilarity, datamodel);


            //Create UserRecomender

            UserBasedRecommender recommender = new
GenericUserBasedRecommender(datamodel, userneighborhood, usersimilarity);


            List<RecommendedItem> recommendations = recommender.recommend(2, 3);


            for (RecommendedItem recommendation : recommendations) {

                System.out.println(recommendation);

            }


        }catch(Exception e){}


    }

   }
```

Compile the program using the following commands:

```
javac Recommender.java
java Recommender
```

It should produce the following output:

```
RecommendedItem [item:3, value:4.5]
RecommendedItem [item:4, value:4.0]
```

Clustering is the procedure to organize elements or items of a given collection into groups based on the similarity between the items. For example, the applications related to online news publishing group their news articles using clustering.

Applications of Clustering

- Clustering is broadly used in many applications such as market research, pattern recognition, data analysis, and image processing.
- Clustering can help marketers discover distinct groups in their customer basis. And they can characterize their customer groups based on purchasing patterns.
- In the field of biology, it can be used to derive plant and animal taxonomies, categorize genes with similar functionality and gain insight into structures inherent in populations.
- Clustering helps in identification of areas of similar land use in an earth observation database.
- Clustering also helps in classifying documents on the web for information discovery.
- Clustering is used in outlier detection applications such as detection of credit card fraud.
- As a data mining function, Cluster Analysis serves as a tool to gain insight into the distribution of data to observe characteristics of each cluster.

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.

Starting Hadoop

Mahout works with Hadoop, hence make sure that the Hadoop server is up and running.

```
$ cd HADOOP_HOME/bin
$ start-all.sh
```

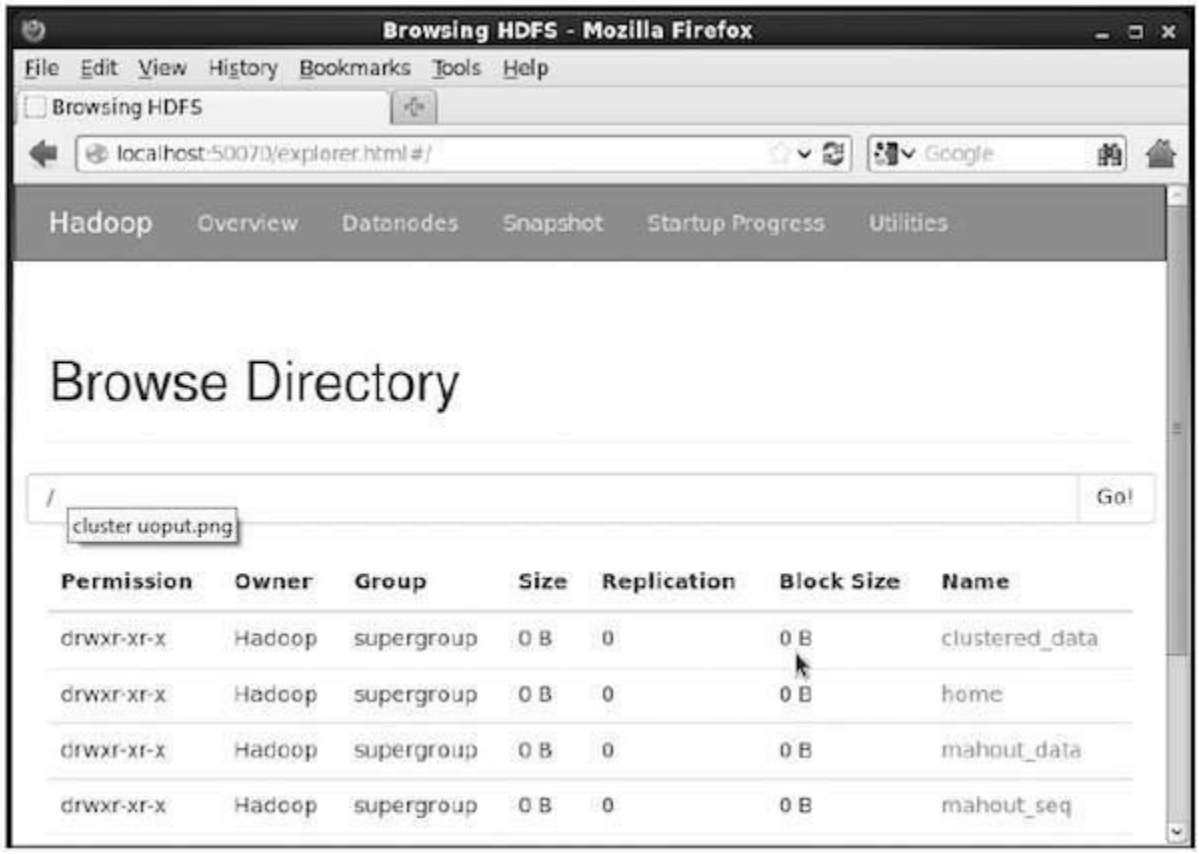
Preparing Input File Directories

Create directories in the Hadoop file system to store the input file, sequence files, and clustered data using the following command:

```
$ hadoop fs -p mkdir /mahout_data
$ hadoop fs -p mkdir /clustered_data
$ hadoop fs -p mkdir /mahout_seq
```

You can verify whether the directory is created using the hadoop web interface in the following URL - **http://localhost:50070/**

It gives you the output as shown below:



Copying Input File to HDFS

Now, copy the input data file from the Linux file system to mahout_data directory in the Hadoop File System as shown below. Assume your input file is mydata.txt and it is in the /home/Hadoop/data/ directory.

```
$ hadoop fs -put /home/Hadoop/data/mydata.txt /mahout_data/
```

Preparing the Sequence File

Mahout provides you a utility to convert the given input file in to a sequence file format. This utility requires two parameters.

- The input file directory where the original data resides.
- The output file directory where the clustered data is to be stored.

Given below is the help prompt of mahout **seqdirectory** utility.

Step 1: Browse to the Mahout home directory. You can get help of the utility as shown below:

```
[Hadoop@localhost bin]$ ./mahout seqdirectory --help
```

```
Job-Specific Options:
--input (-i) input Path to job input directory.
--output (-o) output The directory pathname for output.
--overwrite (-ow) If present, overwrite the output directory
```

Generate the sequence file using the utility using the following syntax:

```
mahout seqdirectory -i <input file path> -o <output directory>
```

Example

```
mahout seqdirectory
-i hdfs://localhost:9000/mahout_seq/
-o hdfs://localhost:9000/clustered_data/
```

Clustering Algorithms

Mahout supports two main algorithms for clustering namely:

- Canopy clustering
- K-means clustering

Canopy Clustering

Canopy clustering is a simple and fast technique used by Mahout for clustering purpose. The objects will be treated as points in a plain space. This technique is often used as an initial step in other clustering techniques such as k-means clustering. You can run a Canopy job using the following syntax:

```
mahout canopy -i <input vectors directory>
-o <output directory>
-t1 <threshold value 1>
-t2 <threshold value 2>
```

Canopy job requires an input file directory with the sequence file and an output directory where the clustered data is to be stored.

Example

```
mahout canopy -i hdfs://localhost:9000/mahout_seq/mydata.seq
-o hdfs://localhost:9000/clustered_data
-t1 20
-t2 30
```

You will get the clustered data generated in the given output directory.

K-means Clustering

K-means clustering is an important clustering algorithm. The k in k-means clustering algorithm represents the number of clusters the data is to be

divided into. For example, the k value specified to this algorithm is selected as 3, the algorithm is going to divide the data into 3 clusters.

Each object will be represented as vector in space. Initially k points will be chosen by the algorithm randomly and treated as centers, every object closest to each center are clustered. There are several algorithms for the distance measure and the user should choose the required one.

Creating Vector Files

- Unlike Canopy algorithm, the k-means algorithm requires vector files as input, therefore you have to create vector files.
- To generate vector files from sequence file format, Mahout provides the **seq2parse** utility.

Given below are some of the options of **seq2parse** utility. Create vector files using these options.

```
$MAHOUT_HOME/bin/mahout seq2sparse
--analyzerName (-a) analyzerName The class name of the analyzer
--chunkSize (-chunk) chunkSize      The chunkSize in MegaBytes.
--output (-o) output                The directory pathname for o/p
--input (-i) input                  Path to job input directory.
```

After creating vectors, proceed with k-means algorithm. The syntax to run k-means job is as follows:

```
mahout kmeans -i <input vectors directory>
-c  <input clusters directory>
-o  <output working directory>
-dm <Distance Measure technique>
-x  <maximum number of iterations>
-k <number of initial clusters>
```

K-means clustering job requires input vector directory, output clusters directory, distance measure, maximum number of iterations to be carried out, and an integer value representing the number of clusters the input data is to be divided into.