Hadoop

An Elephant can't jump. But can carry heavy load.

A 20 page introduction to hadoop and friends.

Prashant Sharma

Table of Contents

1. Introduction

1.1 What is distributed computing?

Multiple autonomous systems appear as one, interacting via a message passing interface, no single point of failure.

Challenges of Distributed computing.

1. Resource sharing. Access any data and utilize CPU resources across the system.
2. Openness. Extensions, interoperability, portability.
3. Concurrency. Allows concurrent access, update of shared resources.
4. Scalability. Handle extra load. like increase in users, etc..
5. Fault tolerance. by having provisions for redundancy and recovery.
6. Heterogeneity. Different Operating systems, different hardware, Middleware system allows this.
7. Transparency. Should appear as a whole instead of collection of computers.
8. **Biggest challenge** is to hide the details and complexity of accomplishing above challenges from the user and to have a common unified interface to interact with it. Which is where **hadoop** comes in.

1.2 What is **hadoop**? (Name of a toy elephant actually)

Hadoop is a framework which provides open source libraries for distributed computing using simple single map-reduce interface and its own distributed filesystem called HDFS. It facilitates scalability and takes cares of detecting and handling failures.

1.3 How does Hadoop eliminate complexities?

Hadoop has components which take care of all complexities for us and by using a simple map reduce framework we are able to harness the power of distributed computing without having to worry about complexities like fault tolerance, data loss.

It has replication mechanism for data recovery and job scheduling and blacklisting of faulty nodes by a configurable blacklisting policy.

Following are major components.

1. Map-reduce (Job Tracker and task tracker)
2. Namenode and Secondary namenode (A HDFS NameNode stores Edit logs and File system Image).
3. Datanode (Runs on slaves)
4. JobTracker (Runs on server)
5. TaskTracker (Runs on slaves)

1.4 What is **map-reduce**?

The map reduce framework is introduced by google. A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs. (definition by Google paper on mapred)

This broadly consists of two mandatory functions to implement: “Map and reduce”. A map is a function which is executed on each key-value pair from an input split, does some processing and emits again a key and value pair. After map and before reduce can begin there is a phase called shuffle which copies and sorts the output on key and aggregates values.

These key and “aggregated value” pairs are captured by reduce and outputs a reduced key value pair . This process is also called aggregation as you get values aggregated for a particular key as input to reduce method. Again in reduce you may play around as you want with key-values and what you emit know is also key value pairs which are dumped directly to a file.

Now simply by expressing a problem in terms of map-reduce we can execute a task in parallel and distribute it across a broad cluster and be relieved of taking care of all complexities of distributed computing. Indeed “Life made easy”, had you tried doing the same thing with MPI libraries you can understand the complexity there scaling to thousands or even hundreds of nodes.

There is a lot more going in map-reduce than just map-reduce. But the beauty of the hadoop is that it takes care of most of those things and a user may not dig into details for simply running a job, though it is good if one has knowledge of those features and can help in tuning parameters and thus improve efficiency and fault tolerance.

1.5 What is HDFS?

Hadoop has its own implementation of distributed file system called hadoop distributed filesystem, which is coherent and provides all facilities of a file system. It implements ACLs and provides a subset of usual UNIX commands for accessing or querying the filesystem and if one mounts it as a fuse dfs then it is possible to access it as any other linux filesystem with standard unix commands.

1.6 What is Namenode?

A single point of failure for an HDFS installation. It contains information regarding a block’s location as well as the information of entire directory structure and files. By saying It is a single point of failure - I mean, if namenode goes down - whole filesystem is offline. Hadoop also has a secondary namenode which contains edit log, which in case of a failure of namenode can be used to replay all the actions of the filesystem and thus restore the state of the filesystem. A secondary namenode regularly contacts namenode and takes checkpointed snapshot images. At any time of failure these checkpointed images can be used to restore the namenode. Current efforts are going on to have high availability for Namenode.

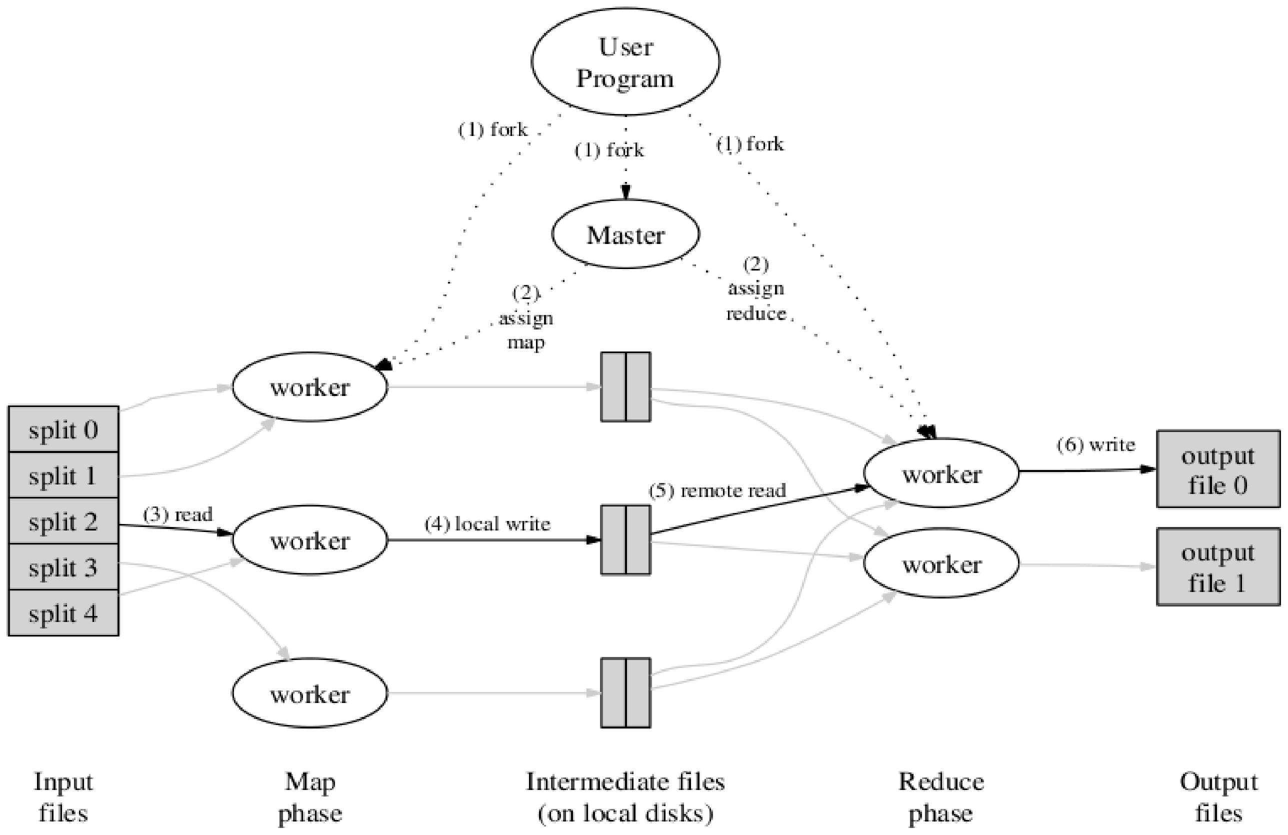
1.7 What is a datanode?

Datanode stores actual blocks of data and stores and retrieves blocks when asked. They periodically report back to Namenode with list of blocks they are storing.

1.8 What is a Jobtracker and tasktracker?

There is one JobTracker(is also a single point of failure) running on a master node and several tasktracker running on slave nodes. Each tasktracker has multiple task-instances running and every task tracker reports to jobtracker in the form of heart beat at regular intervals which also carries message of the progress of the current job it is executing and idle if it has finished executing. Jobtracker schedules jobs and takes care of failed ones by re-executing them on some other nodes. In Mrv2 efforts are made to have high availability for Jobtracker, which would definitely change the way it has been.

2. How map-reduce work?



MapReduce: Simplified Data Processing on Large Clusters

by google.

2.1 **Introduction**.

These are basic steps of a typical map reduce program as described by Google Map-reduce paper. We will understand this taking inverted index as example. An inverted index is same as the one that appear at the back of the book , where each word is listed and then location where it occurs.

Its main usage is to build indexes for search engines.

Suppose you were to build a search engine with inverted index as the index. Now the conventional way is to build the inverted index in a large map(Data structure) and update the map by reading the documents and updating the index.

Limitation of this approach: If the no of documents is large then disk I/o will become a bottle neck. And what if data is in PBs?
 will it scale?

2.2 Map-reduce is the answer.

The map function parses each document, and emits a sequence of <word, document ID> pairs. Where word is the key , as the name says its unique and different document ID with same word are merged when passed to reduce function as input, this is done in sort and shuffle phase.
The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

2.3 An example program which puts inverted index in action using Hadoop 0.20.203 API.

```
package testhadoop;

import java.io.IOException;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.lib.input.*;
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;
import org.apache.hadoop.util.GenericOptionsParser;

/**
 * Here we find the inverted index of a corpus. you can use the wikipedia
 * corpus. Read the hadoop quickstart guide for installation intruction. Map
 * :emits word as key and filename as value. Reduce :emtis word and occurances
 * in filenames.
 *
 * @author Prashant
 *
 */
public class InvertedIndex {
```

```java
/**
 * Mapper is the Abstract class which need to be extended to write a mapper.
 * We specify the input key and value format and output key and value
 * formats in <Inkey,InVal,Outkey,OutVal>. So in the mapper we chose
 * <Object,Text,Text,Text> Remember we can only use the writable implemented
 * classes for key and value pairs(Serialization issues discussed later)
 * emits(Word,Filename in which it occurs).
 *
 * @author Prashant
 */
public static class WFMapper extends Mapper<Object, Text, Text, Text> {

    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
        Text word = new Text();
        // Tokenize each line on basis of \",. \t\n you should add more
        // symbols if you are using HTML or XML corpus.
        StringTokenizer itr = new StringTokenizer(value.toString(),
                "\",. \t\n");
        // Here we used the context object to retrieve the file name of the
        // map is working on.
        String file = new String(((FileSplit) (context.getInputSplit()))
                .getPath().toString());
        Text FN = new Text(file);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            // Emits intermediate key and value pairs.
            context.write(word, FN);

        }
    }
}

/**
 * Almost same concept for reducer as well as mapper(Read WFMapper
 * documentation) Emits (Word,"<Filename>,<Filename>....")
 * Here we store the file names in a hashtable and increment the count to augment the index.
 */
public static class WFReducer extends Reducer<Text, Text, Text, Text> {
    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {

        Hashtable<String, Long> table = new Hashtable<String, Long>();
        for (Text val : values) {
            if (table.containsKey(val.toString())) {
                Long temp = table.get(val.toString());
                temp = temp.longValue() + 1;
                table.put(val.toString(), temp);
            } else
                table.put(val.toString(), new Long(1));
        }
        String result = "";
        Enumeration<String> e = table.keys();
        while (e.hasMoreElements()) {
            String tempkey = e.nextElement().toString();
            Long tempvalue = (Long) table.get(tempkey);
            result = result + "< " + tempkey + ", " + tempvalue.toString()
                    + " > ";

        }
        context.write(key, new Text(result));

    }
}
```

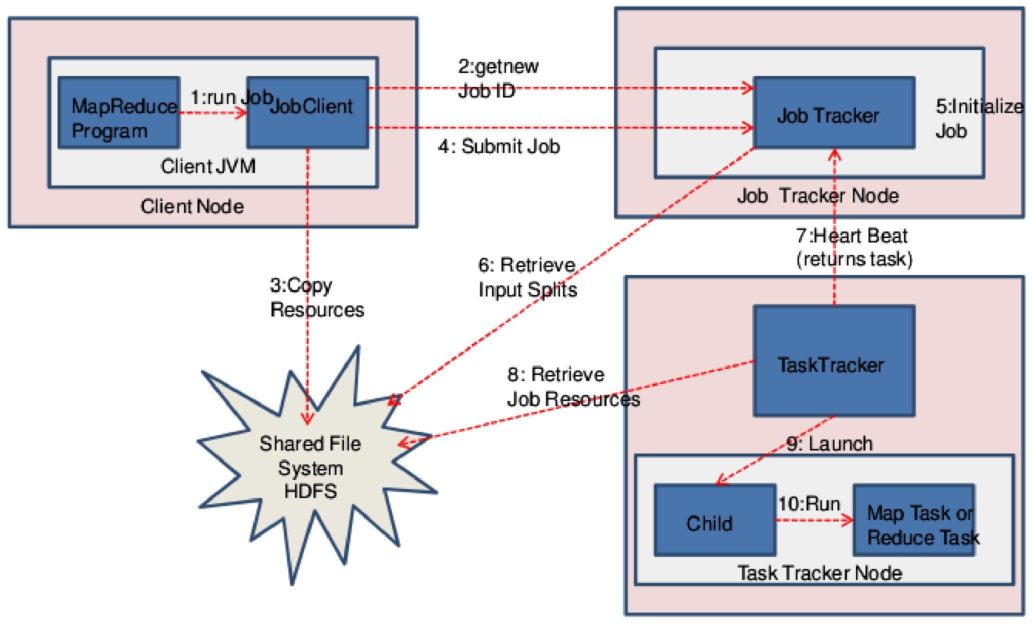
```
    public static void main(String[] args) throws Exception {
        /**
         * Load the configurations into the configuration object(From XML that
         * you Setup while you have setup hadoop)
         */
        Configuration conf = new Configuration();
        // pass the arguements to Hadoop utility for options parsing
        String[] otherArgs = new GenericOptionsParser(conf, args)
                .getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("Usage: invertedIndex <in> <out>");
            System.exit(2);
        }
        // Create Job object from configuration.
        Job job = new Job(conf, "Inverted index");

        job.setJarByClass(InvertedIndex.class);
        job.setMapperClass(WFMapper.class);
        /**
         * Why do we use a combiner when its optional? Well a combiner helps in
         * reducing the output at the mapper end itself and thus bandwidth load
         * is reduced over the network and also increases the efficiency of the
         * reducer.
         */
        job.setCombinerClass(WFReducer.class); // We used the same class for
                                                // combiner as reducer. Although
                                                // its possible to write a
                                                // seperate.
        job.setReducerClass(WFReducer.class);

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

2.4 How Hadoop runs Map-reduce?.

If you are curious about what goes on behind the scenes when you deployed a map-red program and saw a dump of spam printed on stderr(which is Info of your job's status and not error-If everything went fine).



The diagram above is self explanatory and tells us about how map reduce works.

2.4.1 Submit Job

- Asks the Job Tracker for a new ID
- Checks output spec of the Job. Checks o/p Dir. If exists, throws error. Job is not submitted.
- Pass JobConf to JobClient.runJob() or submitJob()
- runJob() blocks, submitJob() does not. (Asynchronous and synchronous ways of submitting a job.)
- JobClient: Determines proper division of input into InputSplits
- Submits the job to JobTracker.
- Computes input split for the job. Splits cannot be computed(inputs does't exist), error is thrown. Job is not submitted
- Copies the resources needed to run the job to HDFS in a directory named specified by "mapred.system.dir".
- Job jar file. Copied with a high replication factor, factor of Can be set by "mapred.submit.replication" property.
- Tells the JobTracker. Job is ready

2.4.2 Job Initialization

- Puts the job in internal Queue
- Job Scheduler will pickup and initialize it

    - Create a Job object and job being run
    - Encapsulate its tasks
    - Book keeping info to track tasks status and progress
- Create list of tasks to run
- Retrieves number of input splits computed by the JobClient from the shared filesystem
- Creates one map task for each split.
- Scheduler creates the Reduce tasks and assigns them to taskTracker.
    - No. of reduce tasks is determined by the map.reduce.tasks.
- Tasks ID’s are given for each task

2.4.3 Task Assignment.

- Task Assignment
- Task trackers send heartbeats to JobTracker via RPC.
- Task tracker indicates readines for a new task
- Job Tracker will allocate a Task
- Job Tracker communicates the task in a response to a heartbeat return
- Choosing a Task Tracker
    - Job Tracker must choose a Task for a TaskTracker
    - Uses scheduler to choose a task from
    - Job Scheduling algorithms – >default one based on priority and FIFO.

2.4.4 Task Execution

- Task tracker has been assigned the task
- Next step is to run the task
- Localizes the job by copying the jar file from the "mapred.system.dir" to job specific directories and copies any other files required.
- Creates a local working dir for the task, un-jars the contents of the jar onto this dir
- Creates an instance of TaskRunner to run the task
- Task runner launches a new JVM to run each task
    - To avoid Task tracker to fail, if any bugs in MapReduce tasks
    - Only the child JVM exits in case of a problem
- TaskTracker.Child.main():
    - Sets up the child TaskInProgress attempt
    - Reads XML configuration
    - Connects back to necessary MapReduce components via RPC
    - Uses TaskRunner to launch user process

3. Hadoop Streaming.

Hadoop streaming is a utility that comes with the Hadoop distribution. The utility allows you to create and run map/reduce jobs with any executable or script as the mapper and/or the reducer.

3.1 A simple example run.

```
$HADOOP_HOME/bin/hadoop  jar $HADOOP_HOME/hadoop-streaming.jar \
   -input myInputDirs \
   -output myOutputDir \
   -mapper /bin/cat \
   -reducer /bin/wc
```

3.2 How it works?

Mapper Side: When an executable is specified for mappers, each mapper task will launch the executable as a separate process when the mapper is initialized. As the mapper task runs, it converts its inputs into lines and feed the lines to the stdin of the process. In the meantime, the mapper collects the line oriented outputs from the stdout of the process and converts each line into a key/value pair, which is collected as the output of the mapper.

Similarly for **reducer** task each reducer task gets as input the converted form of key value pair of the map task into stdin readable input and output of the executable is converted to key value pairs.

3.3 Features.

You can specify internal class as a mapper instead of an executable like this.

```
-mapper org.apache.hadoop.mapred.lib.IdentityMapper \
```

Input ouput format classes can be specified like this.

```
-inputformat JavaClassName
-outputformat JavaClassName
-partitioner JavaClassName
-combiner JavaClassName
```

You can specify JobConf parameters.

```
$HADOOP_HOME/bin/hadoop  jar $HADOOP_HOME/hadoop-streaming.jar \
   -input myInputDirs \
   -output myOutputDir \
   -mapper org.apache.hadoop.mapred.lib.IdentityMapper\
   -reducer /bin/wc \
   -jobconf mapred.reduce.tasks=2
```

4. Hadoop Distributed File System

4.1 **Introduction**.

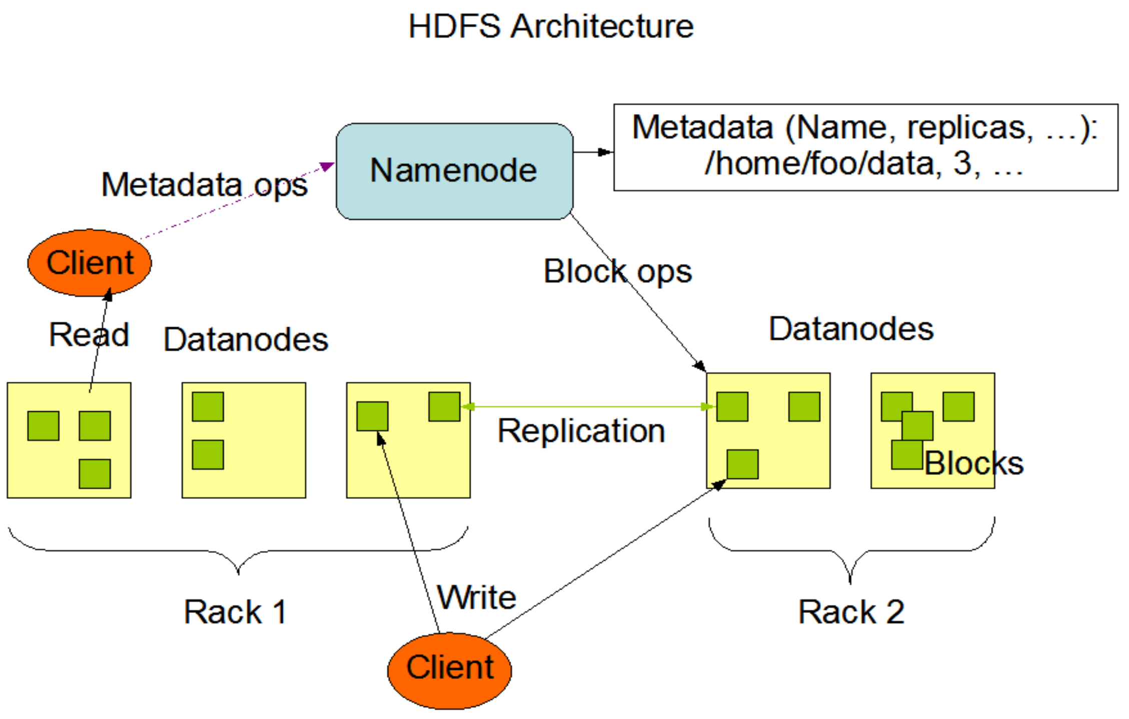
HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

- It has large block size (default 64mb) for storage to compensate for seek time to network bandwidth. So very large files for storage are ideal.

- Streaming data access. Write once and read many times architecture. Since files are large time to read is significant parameter than seek to first record.
- Commodity hardware. It is designed to run on commodity hardware which may fail. HDFS is capable of handling it.

4.2 What HDFS can not do?

- Low latency data access. It is not optimized for low latency data access it trades latency to increase the throughput of the data.
- Lots of small files. Since block size is 64 MB and lots of small files(will waste blocks) will increase the memory requirements of namenode.
- Multiple writers and arbitrary modification. There is no support for multiple writers in HDFS and files are written to by a single writer after end of each file.



4.3 Anatomy of HDFS !

4.3.1 Filesystem Metadata.

- The HDFS namespace is stored by Namenode.
- Namenode uses a transaction log called the EditLog to record every change that occurs to the filesystem meta data.
- For example, creating a new file. Change replication factor of a file
- EditLog is stored in the Namenode’s local filesystem
- Entire filesystem namespace including mapping of blocks to files and file system properties is stored in a file FsImage. Stored in Namenode’s local filesystem.

4.3.2 Anatomy of write.

- DFSOutputStream splits data into packets.
- Writes into an internal queue.
- DataStreamer asks namenode to get list of data nodes and uses the internal data queue.
- Name node gives a list of data nodes for the pipeline.
- Maintains internal queue of packets waiting to be acknowledged.

4.3.3 Anatomy of a read.

- Name node returns the locations of blocks for first few blocks of the file
- Data nodes list is sorted according to their proximity to the client
- FSDataInputStream wraps DFSInputStream, which manages datanode and namenode I/O
- Read is called repeatedly on the datanode till end of the block is reached
- Finds the next DataNode for the next datablock
- All happens transparently to the client
- Calls close after finishing reading the data

4.4 **Accessibility**

HDFS can be accessed from applications in many different ways. Natively, HDFS provides a Java API for applications to use. A C language wrapper for this Java API is also available. In addition, an HTTP browser can also be used to browse the files of an HDFS instance. Or can be mounted as unix filesystem.

4.4.1 DFS shell

HDFS allows user data to be organized in the form of files and directories. It provides a commandline interface called *DFSShell* that lets a user interact with the data in HDFS. The syntax of this command set is similar to other shells (e.g. bash, csh) that users are already familiar with. Here are some sample action/command pairs:

Action	Command
Create a directory named /foodir	bin/hadoop dfs -mkdir /foodir
View the contents of a file named /foodir/myfile.txt	bin/hadoop dfs -cat /foodir/myfile.txt

DFSShell is targeted for applications that need a scripting language to interact with the stored data.

4.4.2 DFS Admin

The *DFSAdmin* command set is used for administering an HDFS cluster. These are commands that are used only by an HDFS administrator. Here are some sample action/command pairs:

Action	Command

Put a cluster in SafeMode	bin/hadoop dfsadmin -safemode enter
Generate a list of Datanodes	bin/hadoop dfsadmin -report
Decommission Datanode datanodename	bin/hadoop dfsadmin -decommission datanodename

4.4.3 Browser Interface

A typical HDFS install configures a web server to expose the HDFS namespace through a configurable TCP port. This allows a user to navigate the HDFS namespace and view the contents of its files using a web browser.

4.4.4 Mountable HDFS: Please visit the wiki for more details MountableHDFS:

5. Serialization.

5.1 Introduction.

Serialization is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage.

Expectation from a serialization interface.

- Compact . To utilize bandwidth efficiently.
- Fast. Reduced processing overhead of serializing and deserializing.
- Extensible. Easily enhanceable protocols.
- Interoperable. Support for different languages.

Hadoop has writable interface which has all of those features except interoperability which is implemented in Avro.

There are following predefined implementations available for WritableComparable.

1. IntWritable
2. LongWritable
3. DoubleWritable
4. VLongWritable. Variable size, stores as much as needed. 1-9 bytes storage
5. VIntWritable. Less used ! as it is pretty much represented by Vlong.
6. BooleanWritable
7. FloatWritable
8. BytesWritable.
9. NullWritable. Well this does not store anything and may be used when we do not want to give anything as key or value. It has also got one important usage: For example we want to write a seq.

file and do not want it be stored in key and value pairs, then we can give key as NullWritable object and since it stores nothing, all values will be merged by reduce method into one single instance.

10. MD5Hash

11. ObjectWritable

12. GenericWritable

Apart from the above there are four Writable Collection types
1. ArrayWritable
2. TwoDArrayWritable
3. MapWritable
4. SortedMapWritable

5.2 Write your own composite writable.

Besides predefined “writables” we can implement WritableComparable to serialize a class and use as key and value. As it is important for any class to be used, as a key and/or value in map-reduce - to be serialized.

5.3 An example explained on serialization and having custom Writables from hadoop repos. (See comments)

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package testhadoop;

import java.io.DataInput;
import java.io.DataOutput;
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.LongWritable;
import org.apache.hadoop.io.RawComparator;
import org.apache.hadoop.io.Text;
```

```java
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Partitioner;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.util.GenericOptionsParser;

/**
 * This is an example Hadoop Map/Reduce application.
 * It reads the text input files that must contain two integers per a line.
 * The output is sorted by the first and second number and grouped on the
 * first number.
 *
 * To run: bin/hadoop jar build/hadoop-examples.jar secondarysort
 *            <i>in-dir</i> <i>out-dir</i>
 */

/*
 * In this example the use of composite key is demonstrated where since there is no default
implementation of a composite key. We had to override methods from WritableComparable.
 *
 * Mapclass:   Mapclass simply reads the line from input and emits the pair as Intpair(left,right)  and
value as right reducer class that just emits the sum of *the input values.

 */
public class SecondarySort2 {

  /**
   * Define a pair of integers that are writable.
   * They are serialized in a byte comparable format.
   */
  public static class IntPair
                    implements WritableComparable<IntPair> {
    private int first = 0;
    private int second = 0;

    /**
     * Set the left and right values.
     */
    public void set(int left, int right) {
      first = left;
      second = right;
    }
    public int getFirst() {
      return first;
    }
    public int getSecond() {
      return second;
    }
    /**
     * Read the two integers.
     * Encoded as: MIN_VALUE -> 0, 0 -> -MIN_VALUE, MAX_VALUE-> -1
     */
    @Override
    public void readFields(DataInput in) throws IOException {
      first = in.readInt() + Integer.MIN_VALUE;
      second = in.readInt() + Integer.MIN_VALUE;
    }
    @Override
    public void write(DataOutput out) throws IOException {
      out.writeInt(first - Integer.MIN_VALUE);
      out.writeInt(second - Integer.MIN_VALUE);
    }
    @Override
    public int hashCode() {
      return first * 157 + second;
    }

    @Override
    public boolean equals(Object right) {
      if (right instanceof IntPair) {
        IntPair r = (IntPair) right;
        return r.first == first && r.second == second;
      } else {
        return false;
      }
    }
    /** A Comparator that compares serialized IntPair. */
    public static class Comparator extends WritableComparator {
      public Comparator() {
        super(IntPair.class);
      }

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

    static {                                        // register this comparator
      WritableComparator.define(IntPair.class, new Comparator());
    }

    /** Compare on the basis of first first! then second. */
    @Override
    public int compareTo(IntPair o) {
      if (first != o.first) {
        return first < o.first ? -1 : 1;
      } else if (second != o.second) {
        return second < o.second ? -1 : 1;
      } else {
        return 0;
      }
    }
  }

  /**
   * Partition based on the first part of the pair. We will need to override the partitioner
   * as we cannot go for default Hashpartitioner. Since we have our own implementation of key and hash
function.
   */
 /* Partion function (first*127 MOD (noOfPartition)).*/
  public static class FirstPartitioner extends Partitioner<IntPair,IntWritable>{
    @Override
    public int getPartition(IntPair key, IntWritable value,
                            int numPartitions) {
      return Math.abs(key.getFirst() * 127) % numPartitions;
    }
  }



  /**
   * Read two integers from each line and generate a key, value pair
   * as ((left, right), right).
   */
  /* Mapclass simply reads the line from input and emits the pair as Intpair(left,right) as key and
value as right*/
  public static class MapClass
         extends Mapper<LongWritable, Text, IntPair, IntWritable> {

    private final IntPair key = new IntPair();
    private final IntWritable value = new IntWritable();

    @Override
    public void map(LongWritable inKey, Text inValue,
                    Context context) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(inValue.toString());
      int left = 0;
      int right = 0;
      if (itr.hasMoreTokens()) {
        left = Integer.parseInt(itr.nextToken());
        if (itr.hasMoreTokens()) {
          right = Integer.parseInt(itr.nextToken());
        }
        key.set(left, right);
        value.set(right);
        context.write(key, value);
      }
    }
  }

  /**
   * A reducer class that just emits the sum of the input values.
   */
  public static class Reduce
         extends Reducer<IntPair, IntWritable, Text, IntWritable> {

    private final Text first = new Text();

    @Override
    public void reduce(IntPair key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      first.set(Integer.toString(key.getFirst()));
      for(IntWritable value: values) {
        context.write(first, value);
      }
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: secondarysort2 <in> <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "secondary sort");
    job.setJarByClass(SecondarySort2.class);
    job.setMapperClass(MapClass.class);
    job.setReducerClass(Reduce.class);

    // group and partition by the first int in the pair
    job.setPartitionerClass(FirstPartitioner.class);

    // the map output is IntPair, IntWritable
    job.setMapOutputKeyClass(IntPair.class);
    job.setMapOutputValueClass(IntWritable.class);

    // the reduce output is Text, IntWritable
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }

}
```

5.4 Why Java Object Serialization is not so efficient compared to other Serialization frameworks?

5.4.1 Java Serialization does not meet the criteria of Serialization format.

1)compact

2)fast

3)extensible

4)interoperable

5.4.2 Java Serialization is not compact.

Java writes the classname of each object being written to the stream, this is true of classes that implement java.io.Serializable or java.io.Externalizable. Subsequent instances of the same class write a reference handle to the first occurrence, which occupies only 5 bytes. However, reference handles don’t work well with random access, since the referent class may occur at any point in the preceding stream that is, there is state stored in the stream. Even worse, reference handles play havoc with sorting records in a serialized stream, since the first record of a particular class is distinguished and must be treated as a special case. All of these problems are avoided by not writing the classname to the stream at all, which is the approach that Writable takes. This makes the assumption that the client knows the expected type. The result is that the format is considerably more compact than Java Serialization, and random access and sorting work as expected since each record is independent of the others (so there is no stream state).

5.4.3 Java Serialization is not fast.

Java Serialization is a general-purpose mechanism for serializing graphs of objects, so it necessarily has some overhead for serialization and deserialization operations.
While Map-Reduce job at its core serializes and deserializes billions of records of different types, thus benefiting in terms of memory and bandwidth by not allocating new objects.

5.4.4 Java Serialization is not extensible.

In terms of extensibility, Java Serialization has some support for evolving a type, but it is hard to use effectively (Writables have no support: the programmer has to manage them himself).

5.4.5 Java Serialization is not interoperable.

In principle, other languages could interpret the Java Serialization stream protocol (defined by the Java Object Serialization Specification), but in practice there are no widely used implementations in other languages, so it is a Java-only solution. The situation is the same for Writables.

5.4.6 Serialization IDL

There are many serialization frameworks that approach the serialization in a different way; rather than defining types through code. It allows them to define in a language-neutral, declarative fashion using Interface description language(IDL). The system generates types for different languages, which encourages interoperability.
Avro is one of the serialization framework which uses IDL mechanism. Please see appendix for details.

6. Distributed Cache.

6.1 Introdution.

A facility provided by map-reduce framework to distribute the files explicitly specified using --files option across the cluster and kept in cache for processing. Generally all extra files needed by map reduce tasks should be distributed this way to save network bandwidth.

6.2 An Example usage:

```
    $hadoop jar job.jar XYZUsingDistributedCache -files input/somethingtobecached input/data
output
```

7. Securing the elephant.

In order to safe guard hadoop cluster against unauthorized access , which may lead to data loss or leak of important data, We would need some mechanism to ensure the access is only to an authorized entity with the rights as to what extent of security we would want to enforce for that user.

Hadoop uses kerberos for ensuring cluster security.

7.1 kerberos tickets.

Kerberos is computer network authentication protocol which assigns tickets to nodes communicating over insecure network to establish each other's identity in a secure manner.

There are three steps to access a service.

a)Authentication. Receives TGT (Ticket granting ticket) from authentication server.

b)Authorization. Contacts Ticket granting server with a TGT for a service ticket.

c)Service Request. Now use Service ticket can be used to access the resource.

7.2 example: of using kerberos.

```
$kinit
provide password for HADOOP@user:******
$hadoop fs -put anything
…...
```

A kerberos ticket is valid for 10 hours once received and can be renewed.

7.3 Delegation tokens.

Delegation token are used in hadoop in background so that user does not have to authenticate at every command by contacting KDC.

7.4 Further securing the elephant.

- Besides this in order to isolate on user rather than operating system.

set mapred.task.tracker.task-controller to org.apache.hadoop.mapred.LinuxTaskController.

- To enforce the ACLs. i.e. each user is able to access only his jobs.

set mapred.acls.enabled to true. and setting for each user mapred.job.acl-view-job and ...modify-job

Hadoop does not use encryption for RPC and transferring HDFS blocks to and from datanodes.

8. Hadoop Job Scheduling.

Now since we have jobs running we need somebody to take care of them. Hadoop has plug-gable schedulers and a default scheduler as well. These are necessary to provide users access to cluster resources and also control access and limit usage.

8.1Three schedulers:

Default scheduler:

- Single priority based queue of jobs
- Scheduling tries to balance map and reduce load on all tasktrackers in the cluster

Capacity Scheduler:

- Yahoo!’s scheduler

The **Capacity Scheduler** supports the following features:

- Support for multiple queues, where a job is submitted to a queue.
- Queues are allocated a fraction of the capacity of the grid in the sense that a certain capacity of resources will be at their disposal. All jobs submitted to a queue will have access to the capacity allocated to the queue.
- Free resources can be allocated to any queue beyond it's capacity. When there is demand for these resources from queues running below capacity at a future point in time, as tasks scheduled on these resources complete, they will be assigned to jobs on queues running below the capacity.
- Queues optionally support job priorities (disabled by default).
- Within a queue, jobs with higher priority will have access to the queue's resources before jobs with lower priority. However, once a job is running, it will not be preempted for a higher priority job, though new tasks from the higher priority job will be preferentially scheduled.

- In order to prevent one or more users from monopolizing its resources, each queue enforces a limit on the percentage of resources allocated to a user at any given time, if there is competition for them.
- Support for memory-intensive jobs, wherein a job can optionally specify higher memory-requirements than the default, and the tasks of the job will only be run on TaskTrackers that have enough memory to spare.

Fair Scheduler: (ensure fairness amongst users)

- Built by Facebook
- Multiple queues (pools) of jobs – sorted in FIFO or by fairness limits
- Each pool is guaranteed a minimum capacity and excess is shared by all jobs using a fairness algorithm
- Scheduler tries to ensure that over time, all jobs receive the same number of resources

Appendix 1A Avro Serialization.

Apache Avro is one of the serialization framework used in Hadoop. Avro offers lot of advantages compared to Java Serialization.
1)compact
2)fast
3)interoperable
4)extensible

Avro is a data serialization system. It provides rich data structures that are compact, and are transported in a binary data format. It provides container file to store persistent data. It provides Remote Procedure Call(RPC). It provides simple integration with dynamic languages. Code generation is not required to read or write data files nor to use or implement RPC protocols. Code generation as an optional optimization, only worth implementing for statically typed languages.

Avro relies on JSON schemas. Avro data is always serialized with its schema. Avro relies on a schema-based system that defines a data contract to be exchanged.When Avro data is read, the schema used when writing it is always present.The strategy employed by Avro, means that a minimal amount of data is generated, enabling faster transport. Avro is compatible with C, Java and Python and some more languages.

Avro Serialzation is compact**.** Since the schema is present when data is read, considerably less type information need be encoded with data, resulting in smaller serialization size.

Avro Serialization is fast. Since the schema is present when data is read, considerably less type information need be encoded with data, resulting in smaller serialization size.The smaller serialization size making it faster to transport to remote machines.

Avro Serialization is Interoperable
Avro schemas are defined with JSON . This facilitates implementation in languages that already have JSON libraries.

Appendix 1B

Documented examples from latest Apache hadoop distribution in the new hadoop 0.21 trunk version. Can be ported to hadoop 0.20.203 api and used.

1.QMC pie estimator.

```java
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.hadoop.examples;

import java.io.IOException;
import java.math.BigDecimal;
import java.math.RoundingMode;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BooleanWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.SequenceFile.CompressionType;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
/** Firstly this is the only Mapreduce program that Prints output to the screen instead of file.
 * HaltonSequence:
         * Populates the arrays with Halton Sequence. i.e. in first iteraion when i=0 q is, 1 , 1/2 , 1/4 .... 63 nos.
         * variable k is moded for randomization, for second iteration when i=1 , q is, 1,1/3,1/9 ,
         * x is sum of all elements of series d * q.
  *nextPoint():
     * This simply undoes what the constructor has done to get the next point
  *QmcMapper creates size samples and checks if they are inside our outside by first subtracting 0.5 (which are center
       * coordinates (supposingly) from x and y and then putting it in equation of the circle and checking if it satisfies
       * (x^2 + y^2 > r^2) then it emits 2 values one is numInside and numoutside seperated by a true or false value as keys.
   *QmcReducer:
     * A reducer does not emit any thing it simply iterates over the keys of true and false and sums
     * the no of points and updates the variable numInside and numoutside , It has a seperate overriden
     * close method wherein it has written the output to the file reduce-out and as it has only one
     * reduce tasks (Thus it is possible else concurrency issues).
     *
     *overridden cleanup():seperate close to register the output to file.
     
   * estimate: calls specified no of mapper and 1 reducer and reads the output from the file written by
   * the reducer.
 *
 */


/**
 * A map/reduce program that estimates the value of Pi
 * using a quasi-Monte Carlo (qMC) method.
 * Arbitrary integrals can be approximated numerically by qMC methods.
 * In this example,
 * we use a qMC method to approximate the integral $I = \int_S f(x) dx$,
 * where $S=[0,1)^2$ is a unit square,
 * $x=(x_1,x_2)$ is a 2-dimensional point,
 * and $f$ is a function describing the inscribed circle of the square $S$,
 * $f(x)=1$ if $(2x_1-1)^2+(2x_2-1)^2 <= 1$ and $f(x)=0$, otherwise.
 * It is easy to see that Pi is equal to $4I$.
 * So an approximation of Pi is obtained once $I$ is evaluated numerically.
 *
 * There are better methods for computing Pi.
 * We emphasize numerical approximation of arbitrary integrals in this example.
 * For computing many digits of Pi, consider using bbp.
 *
 * The implementation is discussed below.
 *
 * Mapper:
 *   Generate points in a unit square
 *   and then count points inside/outside of the inscribed circle of the square.
 *
 * Reducer:
 *   Accumulate points inside/outside results from the mappers.
 *
 * Let numTotal = numInside + numOutside.
 * The fraction numInside/numTotal is a rational approximation of
 * the value (Area of the circle)/(Area of the square) = $I$,
 * where the area of the inscribed circle is Pi/4
```

```java
 * and the area of unit square is 1.
 * Finally, the estimated value of Pi is 4(numInside/numTotal).
 */
public class QuasiMonteCarlo extends Configured implements Tool {
  static final String DESCRIPTION
      = "A map/reduce program that estimates Pi using a quasi-Monte Carlo method.";
  /** tmp directory for input/output */
  static private final Path TMP_DIR = new Path(
      QuasiMonteCarlo.class.getSimpleName() + "_TMP_3_141592654");

  /** 2-dimensional Halton sequence {H(i)},
   * where H(i) is a 2-dimensional point and i >= 1 is the index.
   * Halton sequence is used to generate sample points for Pi estimation.
   */
  private static class HaltonSequence {
    /** Bases */
    static final int[] P = {2, 3};
    /** Maximum number of digits allowed */
    static final int[] K = {63, 40};

    private long index;
    private double[] x;
    private double[][] q;
    private int[][] d;

    /** Initialize to H(startindex),
     * so the sequence begins with H(startindex+1).
     */
    HaltonSequence(long startindex) {
      index = startindex;
      x = new double[K.length];
      q = new double[K.length][];
      d = new int[K.length][];
      for(int i = 0; i < K.length; i++) {
        q[i] = new double[K[i]];
        d[i] = new int[K[i]];
      }

      for(int i = 0; i < K.length; i++) {
        long k = index;
        x[i] = 0;

        for(int j = 0; j < K[i]; j++) {
          q[i][j] = (j == 0? 1.0: q[i][j-1])/P[i];
          d[i][j] = (int)(k % P[i]);
          k = (k - d[i][j])/P[i];
          x[i] += d[i][j] * q[i][j];
        }
      }
    }

    /** Compute next point.
     * Assume the current point is H(index).
      * Compute H(index+1).
     *
     * @return a 2-dimensional point with coordinates in [0,1)^2
     */
    double[] nextPoint() {
      index++;
      for(int i = 0; i < K.length; i++) {
        for(int j = 0; j < K[i]; j++) {
          d[i][j]++;
          x[i] += q[i][j];
          if (d[i][j] < P[i]) {
            break;
          }
          d[i][j] = 0;
          x[i] -= (j == 0? 1.0: q[i][j-1]);
        }
      }
      return x;
    }
  }

  /**
   * Mapper class for Pi estimation.
   * Generate points in a unit square
   * and then count points inside/outside of the inscribed circle of the square.
   */
  public static class QmcMapper extends
      Mapper<LongWritable, LongWritable, BooleanWritable, LongWritable> {

    /** Map method.
     * @param offset samples starting from the (offset+1)th sample.
     * @param size the number of samples for this map
     * @param context output {ture->numInside, false->numOutside}
     */
    public void map(LongWritable offset,
                    LongWritable size,
                    Context context)
        throws IOException, InterruptedException {

      final HaltonSequence haltonsequence = new HaltonSequence(offset.get());
      long numInside = 0L;
      long numOutside = 0L;

      for(long i = 0; i < size.get(); ) {
        //generate points in a unit square
        final double[] point = haltonsequence.nextPoint();

        //count points inside/outside of the inscribed circle of the square
        final double x = point[0] - 0.5;
        final double y = point[1] - 0.5;
        if (x*x + y*y > 0.25) {
          numOutside++;
        } else {
          numInside++;
        }

        //report status
        i++;
        if (i % 1000 == 0) {
          context.setStatus("Generated " + i + " samples.");
        }
      }

      //output map results
      context.write(new BooleanWritable(true), new LongWritable(numInside));
      context.write(new BooleanWritable(false), new LongWritable(numOutside));
    }
  }

  /**
   * Reducer class for Pi estimation.
   * Accumulate points inside/outside results from the mappers.
   */
  public static class QmcReducer extends
      Reducer<BooleanWritable, LongWritable, WritableComparable<?>, Writable> {
    
    private long numInside = 0;
    private long numOutside = 0;
      
    /**
     * Accumulate number of points inside/outside results from the mappers.
     * @param isInside Is the points inside? 
     * @param values An iterator to a list of point counts
     * @param context dummy, not used here.
     */
    public void reduce(BooleanWritable isInside,
        Iterable<LongWritable> values, Context context)
        throws IOException, InterruptedException {
      if (isInside.get()) {
        for (LongWritable val : values) {
          numInside += val.get();
        }
      } else {
        for (LongWritable val : values) {
          numOutside += val.get();
        }
      }
    }

    /**
     * Reduce task done, write output to a file.
     */
    @Override
    public void cleanup(Context context) throws IOException {
      //write output to a file
      Path outDir = new Path(TMP_DIR, "out");
      Path outFile = new Path(outDir, "reduce-out");
      Configuration conf = context.getConfiguration();
      FileSystem fileSys = FileSystem.get(conf);
      SequenceFile.Writer writer = SequenceFile.createWriter(fileSys, conf,
          outFile, LongWritable.class, LongWritable.class, 
          CompressionType.NONE);
      writer.append(new LongWritable(numInside), new LongWritable(numOutside));
      writer.close();
    }
  }

  /**
   * Run a map/reduce job for estimating Pi.
   *
   * @return the estimated value of Pi
   */
  public static BigDecimal estimatePi(int numMaps, long numPoints,
      Configuration conf
```

```java
      ) throws IOException, ClassNotFoundException, InterruptedException {
    Job job = new Job(conf);
    //setup job conf
    job.setJobName(QuasiMonteCarlo.class.getSimpleName());
    job.setJarByClass(QuasiMonteCarlo.class);

    job.setInputFormatClass(SequenceFileInputFormat.class);

    job.setOutputKeyClass(BooleanWritable.class);
    job.setOutputValueClass(LongWritable.class);
    job.setOutputFormatClass(SequenceFileOutputFormat.class);

    job.setMapperClass(QmcMapper.class);

    job.setReducerClass(QmcReducer.class);
    job.setNumReduceTasks(1);

    // turn off speculative execution, because DFS doesn't handle
    // multiple writers to the same file.
    job.setSpeculativeExecution(false);

    //setup input/output directories
    final Path inDir = new Path(TMP_DIR, "in");
    final Path outDir = new Path(TMP_DIR, "out");
    FileInputFormat.setInputPaths(job, inDir);
    FileOutputFormat.setOutputPath(job, outDir);

    final FileSystem fs = FileSystem.get(conf);
    if (fs.exists(TMP_DIR)) {
      throw new IOException("Tmp directory " + fs.makeQualified(TMP_DIR)
          + " already exists.  Please remove it first.");
    }
    if (!fs.mkdirs(inDir)) {
      throw new IOException("Cannot create input directory " + inDir);
    }

    try {
      //generate an input file for each map task
      for(int i=0; i < numMaps; ++i) {
        final Path file = new Path(inDir, "part"+i);
        final LongWritable offset = new LongWritable(i * numPoints);
        final LongWritable size = new LongWritable(numPoints);
        final SequenceFile.Writer writer = SequenceFile.createWriter(
            fs, conf, file,
            LongWritable.class, LongWritable.class, CompressionType.NONE);
        try {
          writer.append(offset, size);
        } finally {
          writer.close();
        }
        System.out.println("Wrote input for Map #"+i);
      }

      //start a map/reduce job
      System.out.println("Starting Job");
      final long startTime = System.currentTimeMillis();
      job.waitForCompletion(true);
      final double duration = (System.currentTimeMillis() - startTime)/1000.0;
      System.out.println("Job Finished in " + duration + " seconds");

      //read outputs
      Path inFile = new Path(outDir, "reduce-out");
      LongWritable numInside = new LongWritable();
      LongWritable numOutside = new LongWritable();
      SequenceFile.Reader reader = new SequenceFile.Reader(fs, inFile, conf);
      try {
        reader.next(numInside, numOutside);
      } finally {
        reader.close();
      }

      //compute estimated value
      final BigDecimal numTotal
          = BigDecimal.valueOf(numMaps).multiply(BigDecimal.valueOf(numPoints));
      return BigDecimal.valueOf(4).setScale(20)
          .multiply(BigDecimal.valueOf(numInside.get()))
          .divide(numTotal, RoundingMode.HALF_UP);
    } finally {
      fs.delete(TMP_DIR, true);
    }
  }

  /**
   * Parse arguments and then runs a map/reduce job.
   * Print output in standard out.
   *
   * @return a non-zero if there is an error.  Otherwise, return 0.
   */
  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: "+getClass().getName()+" <nMaps> <nSamples>");
      ToolRunner.printGenericCommandUsage(System.err);
      return 2;
    }

    final int nMaps = Integer.parseInt(args[0]);
    final long nSamples = Long.parseLong(args[1]);

    System.out.println("Number of Maps  = " + nMaps);
    System.out.println("Samples per Map = " + nSamples);

    System.out.println("Estimated value of Pi is "
        + estimatePi(nMaps, nSamples, getConf()));
    return 0;
  }

  /**
   * main method for running it as a stand alone command.
   */
  public static void main(String[] argv) throws Exception {
    System.exit(ToolRunner.run(null, new QuasiMonteCarlo(), argv));
  }
}

```

Grep example:

```java
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.apache.hadoop.examples;

import java.util.Random;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.map.InverseMapper;
import org.apache.hadoop.mapreduce.lib.map.RegexMapper;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.hadoop.mapreduce.lib.reduce.LongSumReducer;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
/** Grep search uses RegexMapper to read from the input that satisfies the regex
       * and then emits the count as value and word as key.
       *
       * Longsumreducer simply sums all the long values. which is the count emitted by all the
mapper
       * for that particular key.
       * It first searches by above procedure and since the output obtained above is sorted on
words it again
       * uses inversemapper class to sort the output on frequencies.
       */

/* Extracts matching regexs from input files and counts them. */
public class Grep extends Configured implements Tool {
  private Grep() {}                               // singleton

  public int run(String[] args) throws Exception {
    if (args.length < 3) {
      System.out.println("Grep <inDir> <outDir> <regex> [<group>]");
      ToolRunner.printGenericCommandUsage(System.out);
      return 2;
    }

    Path tempDir =
      new Path("grep-temp-"+
          Integer.toString(new Random().nextInt(Integer.MAX_VALUE)));

    Configuration conf = getConf();
    conf.set(RegexMapper.PATTERN, args[2]);
    if (args.length == 4)
      conf.set(RegexMapper.GROUP, args[3]);

    Job grepJob = new Job(conf);

    try {

      grepJob.setJobName("grep-search");

      FileInputFormat.setInputPaths(grepJob, args[0]);

      grepJob.setMapperClass(RegexMapper.class);

      grepJob.setCombinerClass(LongSumReducer.class);
      grepJob.setReducerClass(LongSumReducer.class);

      FileOutputFormat.setOutputPath(grepJob, tempDir);
      grepJob.setOutputFormatClass(SequenceFileOutputFormat.class);
      grepJob.setOutputKeyClass(Text.class);
      grepJob.setOutputValueClass(LongWritable.class);

      grepJob.waitForCompletion(true);

      Job sortJob = new Job(conf);
      sortJob.setJobName("grep-sort");

      FileInputFormat.setInputPaths(sortJob, tempDir);
      sortJob.setInputFormatClass(SequenceFileInputFormat.class);

      sortJob.setMapperClass(InverseMapper.class);

      sortJob.setNumReduceTasks(1);                 // write a single file
      FileOutputFormat.setOutputPath(sortJob, new Path(args[1]));
      sortJob.setSortComparatorClass(          // sort by decreasing freq
        LongWritable.DecreasingComparator.class);

      sortJob.waitForCompletion(true);
    }
    finally {
      FileSystem.get(conf).delete(tempDir, true);
    }
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new Grep(), args);
    System.exit(res);
  }

}

```

WordCount

```java
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.apache.hadoop.examples;

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;
import org.apache.hadoop.util.GenericOptionsParser;
/**
 * Word count Documentation.
 * TokenizerMapper: It takes as input each line from the input set and tokenize each word and emits each
                    * word as key and integer one as value.
 * IntSumReducer: It accepts each word as key and aggregates all values i.e. the ones mapper has emitted.
                    * So iterating over all the values and adding gives the count of that word.
 *
 *
 */

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();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }
    Job job = new Job(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(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```