Hadoop

1. Introduction

History

At first there was Nutch, created by Doug Cutting, an open source web search engine. The architecture for Nutch could not support the billions of pages on the web. In 2003, Google published a paper describing the Google File System, and Nutch Distributed Filesystem (NDFS) was written based on GFS. In 2004, Google published a paper introducing MapReduce, and by 2005, Nutch developers have enabled Nutch algorithms to run using MapReduce and NDFS. In 2006 Hadoop was formed as an independent project. Cutting joined Yahoo!, which provided a team and resources to run Hadoop at web scale.

What is Hadoop?

A framework that allows of distributed processing of large data sets across clusters of commodity hardware using a simple programming model. Large data sets are data if the ranges of petabytes, and clusters of computers are minimally in the hundreds range.

2. HDFS

A distributed file system designed to store a large amount of data and provide access to this data to many clients distributed across the network. HDFS should store files reliably, and provide fast scalable access.

A HDFS cluster has two kinds of nodes, NameNode (master) and DataNodes (slaves). The NameNode manages the filesystem namespace: it maintains the filesystemtree and metadata for the files and directories. Files are broken up into blocks (typically 64MB) and blocks are stored in DataNodes. Blocks are replicated (default: 3) across DataNodes. NameNode failure is prevented by 1) configuring NameNode to write its persistent file state to multiple file systems, 2) secondary namenode that periodically merges the namespace image with the edit log and keeps a copy of this merged image.

Reading Data from a Hadoop URL

```
public class URLCat {
	static { // to make java recognize Hadoop's hdfs URL scheme
		URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
	}

	public static void main(String[] args) throws Exception {
		InputStream in = null;
		try {
			in = new URL(args[0].openStream();
			// copy bytes from URL, to console, buffer size used for copying, and
whether
			// to close the stream. False because we close the stream manually
below
			IOUtils.copyBytes(in, System.out, 4906, false);
		} finally {
			IOUtils.closeStream(in);
		}
	}
}
```

Reading Data Using the FileSystem API

A file in the Hadoop filesystem is represented by a Hadoop Path object (e.g. Hdfs:// localhost/user/hadoop/somefile.txt)

FileSystem is a general filesystem API, the first step is to retrieve an instance for the filesystem we want to use, in this case HDFS. There are two static factory methods for getting a FileSystem instance:

```
public static FileSystem get(Configuration conf) throws IOException
public static FileSystem get(URI uri, Configuration conf) throws IOException
```

A Configuration object encapsulates a client or server's configuration, set using the configuration files read from the classpath, conf/core-site.xml. First method returns the default filesystem stated in the configuration file. The second uses the given URI's scheme and authority to determine the filesystem to use, falling back to the default if no scheme is specified.

With a FileSystem instance created, we invoke an open() method to get the input stream:

```
public FSDataInputStream open(Path f) throws IOException
public abstract FSDataInputStream open(Path f, int bufferSize) throws IOException
```

```
public class FileSystemCat {
	public static void main(String[] args) throws Exception {
		String uri = args[0];
		Configuration conf = new Configuration();
		FileSystem fs = FileSystem.get(URI.create(uri), conf);
		InputStream in = null;
		try {
			in = fs.open(new Path(uri));
			IOUtils.copyBytes(in, System.out, 4906, false);
		} finally {
			IOUtils.closeStream(in);
		}
	}
}
```

The open method on FileSystem returns a FSDataInputStream rather than the standard java.io class. FSDataInputStream is a specilization of java.io.DataInputStream which implements Seekable and PositionedReadable, supports random access. Seekable interface permits seeking to an arbitrary, absolute position to a file via method seek(). It has a method getPos() for querying the current offsent from the start of the file.

```
		FSDataInputStream in = null;
		try {
			in = fs.open(new Path(uri));
			IOUtils.copyBytes(in, System.out, 4096,f alse);
			in.seek(0); // go back to start of file
			IOUtils.copyBytes(in,  System.out, 4096, false);
		} finally {
			IOUtils.closeStream(in);
		}
```

PositionedReadable interface has methods:
-read() to read a number of bytes from a given position in the file into the buffer at the given offset in the buffer, returns the number of bytes read
-readFully() will read number of bytes into the buffer unless the end of file is reached
Both methods preserve the current offset in the file.
Seek is a relatively expensive operation and should be used sparingly.

Writing Data

FileSystem class has a number of methods for creating a file, the simplest being:

```
public FSDataOutputStream create(Path f) throws IOException
```

the create method creates any parent directories that do not exist, which might cause unintended behaviour, hence exist() should be called first to check for existence of directories.
There is also an overloaded method for passing a callback interface, Progressable, so that applications can be notified of the progress of data being written to the outputstream.
We can also append to an existing file using the overloaded append() method

```
public FSDataOutputStream append(Path f) throws IOException
```

```
public class FileCopyWithProgress {
        public static void main(String[] args) throws Exception {
                String localSrc = args[0];
                String dest = args[1];
                InputStream in = new BufferedInputStream(new
FileInputStream(localSrc));

                Configuration conf = new Configuration();
                FileSystem fs = FileSystem.get(URI.create(dst), conf);
                OutputStream out = fs.create(new Path(dest), new Progressable() {
                        public void progress() {
                                System.out.prin("."); // display a . every time the
progress()
// method is called by Hadoop, which is after 64K packet of data is written
                        }
                });
                IOUtils.copyBytes(in, out, 4096, true);
        }
}
```

Querying the FileSystem

The method getFileStatus() on FileSystem provides a way of getting FileStatus object for a single file or directory

```
String uri = args[0];
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(URI.create(uri), conf);
Path dir = new Path("/dir);
FileStatus stat = fs.getFileStatus(dir);
assertThat(stat.getPath().toUri().getPath(), is("/dir));
```

listStatus() is used to list the contents of the directories

```
Path[] paths = new Path[args.length];
for (int i=0, i<paths.lenth; i++) {
        paths[i] = new Path(args[i]);
}
FileStatus[] status = fs.listStatus(paths);
Path[] listedPaths = FileUtil.stat2Paths(status);
for (Path p : listedPaths) {
        System.out.prinln(p);
}
```

globStatus() methods returns an array of FileStatus objects whos path match the supplied pattern, Hadoop supports the same set of glob characters as Unix bash

Deleting Data

```
public boolean delete(Path f, boolean recursive) throws IOException
```

Data Flow

<u>How a File Read is performed</u>

The client opens the file by calling open() on the FileSystem object, which for HDFS is an instance of DistributedFileSystem

```
FileSystem fs = FileSystem.get(URI.create(uri), conf);
```

FSDataInputStream in = fs.open(new Path(uri));

DistributedFileSystem calls the namenode, using RPC, to determine locations of the blocks for the first few blocks of the file. For each block, namenode returns the addresses, datanodes are sorted according to proximity to the client. The DistributedFileSystem returns a FSDataInputStream to the client to read data from, FSDataInputStream wraps a DFSInputStream, which manages the datanode and namenode I/O. Client then calls read() on the stream. DFSInputStream connects to the first datanode for the first block of file. Data is streamed from the datanode to the client, which calls read() repeatedly until the end of block, then DFSInputStream will close the connection and find the best datanode for the next block. DFSInputStream will also call the datanode locations for next batch of blocks to be read from namenode. When the client is finished reading, it calls close() on FSDataInputStream.

How a File Write is performed

The client creates a file by calling create() on DistributedFileSystem, which makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it. The namenode checks to see if file does not already exist and that the client has permissions to create a file. If the checks pass, the DistributedFileSystem returns a FSDataOutputStream, which wraps a DFSOutputStream,for the client to start writing data to. As the client writes data, DFSOutputStream splits it into packet which it writes to an internal queue, called the data queue. The data queue is consumed by the DataStreamer, which ask the namenode to pick and allocate new suitable datanodes to store the replicas. The list of datanodes forms a pipeline. The DataStreamer streams the packets to the first datanode in the pipeline, which stores the packet ad forwards it down the pipeline. DFSOutputStream also maintains an internal queue of packets waiting to be acknowledged by datanodes. A packet is removed from the ack queue only when it has been acknowledge by all the datanodes in the pipeline. When the client has finished writing data, it calls close() on the stream.

Coherency Model

This model describes the data visibility of reads and writes for a file. After creating a file, it is visible in the filesystem namespace. However, any content written to the file is not guaranteed to be visible, even if the entire stream is flushed. Once more than a block's worth of data has been written, the first block will be visible to new readers. HDFS provides a method for forcing all buffers to be synchronized to the datanodes via the sync() method on FSDataOutputStream, in the event of a crash that data will not be lost. There is a trade-off between data robustness and throughput

Parallel copying with distcp

Hadoop comes with distcp that is used for copying large amounts of data to and from Hadoop filesystems in paralle, e.g. Transfering data between two HDFS clusters

% hadoop distcp hdfs://namenode1/foo hdfs://namenode2/bar

will copy the /foo directory into /bar director, hence namenode2 will have bar/foo. Distcp is implemented as a MapReduce job with only a map phase.

Hadoop Archives

HDFS stores small files inefficiently, since each file is stored in a block and block metadata is held in memory by the namenode. A large number of small files can eat up a lot of memory. Hardoop Archives (HAR files), packs files into HDFS blocks more efficiently, reducing memory usage while still allowing transparent access to files. HAR files can also be used as input to MapReduce.

% hadoop archive -archiveName files.har /my/files /my

(keyword, specify archive name, files to archive, output folder)

3. Hadoop I/O

Data Integrity

Error-detecting code CRC-32 (cyclic redundancy check), which computes a 32-bit integer checksum for input of 512bytes. Datanodes are responsible for verifying the data they receive before storing the data and its checksums. A DataBlockScanner is run periodically to verify that all blocks stored on the datanode are not corrupted due "bit rot"

Compression

Includes DEFLATE, gzip, ZIP, bzip2, LZO. All algorithms exhibit a space/time trade-off. Nine different options: -1 means optizmized for speed and -9 means optimized for space.

Codec

An implementation of compression-depression algorithms. In Hadoop, a codec is represented by an implementation of the CompressionCodec interface. Compression Codec has two methods which allow us to easiy compress or decompress data: createOutputStream(OutputStream out) method to create a CompressionOutputStream to which you write your uncompressed data to have it written in compressed form to the underlying stream

```
public class StreamCompressor {
        public static void main(String[] args) throws Exception {
                String codecClassname = args[0];
                Class<?> codecClass = Class.forName(codecClassname);
                Configuration conf = new Configuration();
                CompressionCodec codec = (CompressionCodec)
ReflectionUtils.newInstance(codecClass, conf);
// use ReflectionUtils to construct a new instance of  coded
                CompressionOutputStream out =
codec.createOutputStream(System.out);
// obtain a compression wrapper around System.out
                IOUtils.copyBytes(System.in, out, 4906, false);
                out.finish();
        }
}
```

Infering CompressionCodecs using CompressionCodecFactory

```
CompressionCodecFactory factory = new CompressionCodecFactory(conf);
CompressionCodec codec = factory.getCodec(inputPath);
```

Using Compression in MapReduce

If the input files are compressed, they will be automatically decompressed as they are read by MapReduce, using the filename extension to determine the codec to use. To compress the output of MapReduce job, in the job configuration, set the mapred.output.compress to true and mapred.output.compress.codec to the classname of the codec to use.

```
conf.setBoolean("mapred.output.compress", true);
conf.setClass("mapred.output.compression.codec", GzipCodec.class,
CompressionCodec.class);
```

Map outputs can be compressed to speed up data transfer to reducer nodes

```
conf.setCompressMapOutput(true);
conf.setMapOutputCompressorClass(GzipCodec.class);
```

Serialization

Process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage. Hadoop uses its own serialization format, Writables.

The Writable Interface

The Writable interface defines two methods, write(DataOutput out) for writing its state to a DataOutput binary stream, and readFields(DataInput in) for reading its state from a DataInput binary stream. E.g. IntWritable, a wrapper for Java int.
IntWritable implements the WritableComparable interface, this interface extends the Writable and java.lang.Comparable interfaces.

Writable Classes

These include BooleanWritable, ByteWritable, IntWritable, VintWritable, FloatWritable, LongWritable, VlongWritable, DoubleWritable, Text, NullWritable, BytesWritable, MD5Hash, ObjectWritable, GenericWritable.

Text

Text is a Writable for UTF-8 sequences. Indexing for Text class is in terms of position in the encoded byte sequence, not the Unicode character in the string. CharAt() returns an int representing a Unicode code point. Iterating over Unicode characters in Text is complicated by the use of byte offsets for indexing. First the Text object is turned into a java.nio.ByteBuffer, then repeatedly call the bytesToCodePoint() static method on Text with the buffer. This method extracts the next code point as an int and updates the position in the buffer. Text is mutable, a Text instance can be reused by calling set() on it.

BytesWritable is a wrapper for an array of binary data. Its serialized format is an integer field (4 bytes) that specifies the number of bytes to follow, followed by the bytes themselves.

NullWritable is a special type of Wrtiable as it has a zero-length serialization. No bytes are written to or read from the stream. It is used as a placeholder.

Custom Writable

Custom Writable class must implement WritableComparable. Constructors (including Default constructors), set(), get(), write(), readFields(), compareTo(), ovewrite the hasCode(), equals() and toString().

File-Based Data Structures

For some applications, a specialized data structure is required. Hadoop has developed a number of higher-level containers for situations where puttting each blob of binary data into its own file doesn't scale.

SequenceFile

To create a sequence file, use one of its createWriter() static methods, which returns a SequenceFile.Writer instance, there are serveral overloaded versions but they require you to specify a stream to write to, a Configuration object, and the key and value types.

SequenceFile.Writer writer = SequenceFile.createWriter(fs, conf, path, key.getClass(), value.getClass());

Keys and values can use any serialization framework. To read a sequence file from beginning to end, create an instance of SequenceFile.Reader and iterate over records by repeatedly invoking one of the next() methods. This next() method takes a key and a value argument and reads the next key and value in the stream into these variables. The sequence file consists of a header followed by one or more records. The first three bytes of the file are bytes SEQ, followed by a single byte representing the version number. The header contains other fields including names of the key and value classes, compression details, user-defined metadata and the sync marker. If no compression is enabled, each record is made up of the record length, the key length, the key and the value. The length fields are written as four-byte integers. The format for record compression is identical to no compression, except that value bytes are compressed. Block compressions compresses multiple records at once. The format of a block is a field indicating the number of records in the block, followed by four compressed fields: key length, keys, value lengths, value.

MapFile

A sorted SequenceFile with an index to permit lookups by key. Writing to a MapFile is similar to writing to a SequenceFile, creating a MapFile.Writer instance and calling the append() method to add the entries in order. Keys must be instances of WritableComparable and values of Writable. MapFile is actually a directory containing two SequenceFiles, data and index. Data file contains all of the entries in order, and index file contains a fraction of the keys and a mapping from the keys to that key's offset in the data file. Iterating the entries in order in a MapFile by creating a MapFile.Reader and calling next() until it reaches false when it reaches the end of file. A random access lookup can be performed by calling get(). A MapFile can be created from a SequenceFile by calling the fix() method on MapFile, which recreates the index for a MapFile.

4. MapReduce*

A programming model for parallel data processing. MapReduce breaks up the processing into two main phase: map phase and reduce phase. Each phase has key-value pairs as input and output, the data type is determined by the programmer.

Example data: Google search query log stored as text files. Each record contains information including IP address of Google user, query entered, date, time, etc. Each record is on a single line and is stored without delimiters. E.g. 192.168.1.24“what is hadoop”199112202300 (IP, query, YYMMDDhhmm).

Example processing: to extract the query and year to examine trends

Original data (**Bold** indicates fields we are interested in):
192.168.1.24**“what is hadoop”1991**12202300
192.168.1.24**“what is hadoop”1991**12212300
192.168.1.24**“what is apache”1991**12212300
192.168.1.24**“what is apache pig”1992**12222300

A MapReduce Job is a unit of work that a client wants to perform, in this case is to extract the query and year from a large number of log files.

The client calls the runJob() method on JobClient, which creates a new JobClient instance and calls submitJob() on it. SubmitJob() asks the jobtracker for a new job ID, checks the output specification of the job, computes the input splits for the job, copies the resources needed to run the job to the jobtracker's filesystem. Having submitted the job, runJob() polls the job's progress once a second and reports the progress to the console.

When the JobTracker receives a call to its submitJob() method, it puts it into and internal queue where the job scheduler will pick it up and initialize it. The job scheduler retrieves the number of input splits and generates one map tasks for each split. The JobTrackerw will choose a task and send it to the TaskTracker for execution. TaskTrakers periodically send heartbeat methods to the JobTracker. JobTracker takes into account data-locality when assigning map tasks, where data is on the same node or near the TaskTracker. The TaskTracker 1) localizes the job JAR and any required files by copying it from the shared filesystem, 2) creates a local working directory for the class, 3) creates an instance of TaskRunner to run the task. Each map tasks runs one map function for each record (in our example: line) in the split.

An InputFormat (default: TextInputFormat) defines how the files (query log files) are split and read. The RecordReader loads the data from the files and convert it into key-value pairs. In this case, by choosing a TextInputFormat, each line in the log file becomes our text value. The key is the byte offset of the beginning of the line from the beginning of the file, and this is large unimportant and can be ignored.

TextInputFormat will present these key-value pairs to the map function:
(0, 192.168.1.24**“what is hadoop”1991**12202300)

(43, 192.168.1.24**“what is hadoop”1991**12202300)
(86, 192.168.1.24**“what is apache”1991**12212300)
(129, 192.168.1.24**“what is apache pig”1992**12222300)

Map function will extract the query and year fields as a key-value pair:
(1991, “what is hadoop”)
(1991, “what is hadoop”)
(1991, “what is apache”)
(1992, “what is pig”)
OutputCollector collects each pair and stores this information in the a memory buffer with a dfault size of 100MB. When the contents of the buffer reaches a threshold (io.sort.spill.percent defaults 80%). If a Combiner (optional) is specified, it is run when the number of spills reaches a specified amount (min.num.spills.for.combine default 3). The data is then divided into partitions corresponding to the reducers that they will ultimately be sent to. A background thread will start to spill the contents to disk. A compression codec can be specified in this case to reduce the file size of map outputs.

The output from the map function resides in the local disk of the tasktracker that ran the map task. As map tasks complete successfully, their notify their parent tasktracker of the status, which notifies the jobtracker. A thread in the reducer periodically asks the jobtracker for map output locations until it has retrieved them all. Since map tasks may finish at different times, reduce tasks start copying map's output as soon as each completes. Reduce tasks can copy map outputs in parallel (mapred.reduce.parallel.copies default 5) to their buffer (mapred.job.shuffle.input.buffer.percent) if the outputs are small enough. When all map outputs are copied, the reduce task moves into the sort phase, which merges the map outputs and maintains their sort ordering.

will be processed by the Map Reduce framework before being sent to the reduce function. The various key-value pairs will be sent to individual reduce nodes by their key values, identical keys are sent to the same machine (process known as shuffling), and this machine is decided on by the Partitioner.
Node 1: (1991, “what is apache”)
(1991, “what is hadoop”)
(1991, “what is hadoop”)

Node 2: (1992, “what is pig”)

These intermediate key-value pairs are then sorted on individual reduce nodes.
(1991, [“what is apache”, “what is hadoop”, “what is hadoop”])
(1992, [“what is pig”])

The shuffled and sorted pairs are then sent as input to the reduce function. The reduce function will iterate through the keys and pick the query that has the most frequency:
(1991, “what is hadoop”)
(1992, “what is pig”)
OutputCollector collects each pair.

These final pairs are written to output files in a manner defined by OutputFormat. The RecordReader writes the records.

When the JobTracker receives a notification that the last task is complete, the status for the job is changed to “successful”. Hence when the JobClient polls for status and learns that the job is completed, it prints the message to the client and returns the runJob() method. Clean-up.

Failure

Task Failure

Due to hanging, crashing, exception. Child JVM reports the error to parent tasktracker before exiting, entered into user log files. TaskTracker will mark the task attempt as failed and frees up a slot to run another task. Hanging tasks are killed after a user-defined time (mapred.task.timeout). When the JobTracker is informed that a task has failed, it will reschedule the execution of the task. A value can be configured to determine them maximum number of times a task can fail before it is not retried anymore (mapred.map.max.attempts and mapred.reduce.max.attempts)

TaskTracker Failure

TaskTracker can fail by crashing or running very slowly, which will stop sending heartbeats to the JobTracker (mapred.tasktracker.expiry.interval) JobTracker remove it from its pool of tasktrackers and arrange for tasks perviously ran on that TaskTracker to be rerun if they belong to an incomplete job. Tasks in progress are also rescheduled.

JobTracker Failure

Most serious failure. Usage of ZooKeeper.

Speculative Execution

When a particular task have failed to make as much progress, on average, as other tasks from the job, a speculative task is launched. When a task is completed, any duplicate tasks will be killed.

5. Developing MapReduce Applications

Configuration API

Components in Hadoop are configured using an instance of the Configuration class. Configurations read their properties from XML files, e.g. Core-default.xml and core-site.xml. Properties defined in resources that are added later override the earlier configuration. Properties marked as final cannot be overidden.

```
Configuration conf = new Configuration()
conf.addResource("configuration-1.xml")
conf.addResource("configuration-2.xml")
```

Configuration properties can be defined in terms of other properties, or system properties, e.g. ${default-block-size}

Writing a Unit Test

Since outputs are written to an OutputCollector, rather than simply being return from the method call, the OutputCollector needs to be replaced with a mock so that it outputs can be verified. One way is to use Mockito.

```
Public class MaxTemperatureMapperTest {
        @Test
        public void processValidRecord() throws IOException {
                MaxTemperatureMapper = new MaxTemperatureMapper();
                Text value = new Text("65432145168231" + "654198645498321");
                OutputCollector<Text, IntWritable> output =
mock(OutputCollector.class);
                mapper.map(null, value, output, null);
                verify(output).collect(new Text("32"), new IntWritable(49));
        }
}
```

This test passes a weather record to the mapper, then checks the output year and reading. To create a mock OutputCollector we call Mockito's mock() method, passing the class of the type we want to mock. We then invoke the mappers map method, and verify that the mock object was called with the correct method and arguments using Mockito's verify() method.

Running Locally on Test Data

Using the Tool interface to write a driver for the MapReduce job.

```
Public class MaxTemperatureDriver extends Configured implements Tool {
        public int run(String[] args) throws Exception {
                if (args.length != 2){
                        System.err.printf("....");
                        ToolRunner.printGenericCommandUsage(System.err);
                        return -1;
                }
        Job conf = new JobConf(getConf(), getClass());
        conf.setJobName("Max Temperature");
        // set file input/output format, input/ouput keyclass, mapper/reducer class etc
        JobClient.runJob(conf);
```

```
        return 0;
        }
        public static void main(String[] args) throws Exception {
                int exitCode = ToolRunner.run(new MaxTemperatureDriver(), args);
                System.exit(exitCode);
        }
}
```

Running Dependent Jobs

Linear chain:

```
        JobClient.runJob(conf1);
        JobClient.runJob(conf2);
```

MapReduce Types and Formats

An input split is a chunk of input that is processed by a single map. Each split is divided into records and the map processes each record. InputSplits does not contain the input data, its a reference to the data (location and size). InputSplits are created by the InputFormat. The JobClient calls the getSplits() method, passing the number of map tasks as an argument in the method. Splits are then calculated and sent to the jobtracker, which uses their localtions to schedule map tasks to process on tasktrackers. On a tasktracker, the map task passes the split to the getRecordReader() method on InputFormat to obtain a RecordReader for the split. The RecordReader iterates over the records and map tasks uses one to generate record key-value pairs, which it passes to the map function.

FileInputFormat is the base class for all implementations of InputFormat that use files as their data source. It provides a place to define which files are included as input to the job, and an implementation for generating splits for the input files (this job is performed by subclasses). Input paths can be set using FileInputFormat's static addInputPath/s() method.
CombineFileInputFormat works better with small files (recall that 1 map task is called for 1 split, hence many task for many splits), it packs many small files into each split so that each mapper has more to process. It is an abstract class so code will have to be written.

Text Input

TextInputFormat, each record is a line of input, the key (LongWritable) is a byte offset within the file of the the beginning of the line, the value is the contents of the line (Text) , excluding any line terminators.
KeyValueTextInputFormat, each record is a line of input, the key is the sequence before the delimiter (key.value.separator.in.input.line default \t) and the value is the sequence after the delimiter.
NlineInputFormat, specifies the lines of input each mapper receives, exactly the same as TextInputFormat
StreamInputFormat, StreamXmlRecordReader to read XML documents

Binary Input

SequenceFileInputFormat, stores sequences of binary key-value pairs. Splittable (with sync points), support compression and can store arbitrary types using serialization frameworks.
Multiple Inputs, allows you to sepcify the InputFormat and Mapper on a per-path basis

MultipleInputs.addInputPath(conf, InputPath, TextInputFormat.class, MapperType1.classs);
MultipleInputs.addInputPath(conf, InputPath, TextInputFormat.class, MapperType2.classs);
Database Input (and Output), compatible with Hbase's TableInputFormat/ TableOutputFormat

Output Formats

TextOutputFormat, writes records as lines of text, key-value pair separated by character (mapred.textoutputformat.separator default \t)

Binary Output

SequenceFileOutputFormat, writes sequence files for output, good choice if the output forms an input for another MapReduce job.
MapFileOutputFormat, keys in a MapFile must be added in order.
Multiple Outputs, allows control over the naming of files, or to produce multiple files per reducer.
MultipleOutputFormat, allows you to write data to multiple files whose names are derived from the output keys and values using generateFileNameForKeyValue() method.
MultipleOutputs, emits different types for each output

6. Starting a Pseudo-Distribution Mode

After installation of Hadoop, edit configuration settings in `conf/hdfs-site.xml` or `conf/hadoop-site.xml` in earlier versions:

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
  </property>
  <property>
    <name>mapred.job.tracker</name>
    <value>hdfs://localhost:9001</value>
  </property>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
        <!-- set to 1 to reduce warnings when
        running on a single node -->
  </property>
</configuration>
```

Check that the command `% ssh localhost` does not require a password, if it does, set up passwordless ssh:

```
ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Try again, if it fails, figure out the error with ssh-agent.
A new distributed filesystem must be formatted with the command (on master node):

```
bin/hadoop namenode -format
```

Open conf/hadoop-env.sh and define JAVA_HOME in it. Then start Hadoop daemon with:

```
bin/start-all.sh
```

Web-based interface for namenode and jobtracker:

http://localhost:50070/
http://localhost:50030/

Input files are copied to the HDFS using:

```
bin/hadoop dfs -put <localsrc> <dst>
```

To shutdown:

```
bin/stop-all.sh
```

7. Fully-Distributed operation

Specify the hostname or IP address of master server in the value for fs.default.name, as hdfs://master.example.com/ in conf/core-site.xml.
The host and port of the master server in the value of mapred.job.tracker as master.example.com:port in conf/mapred-site.xml.
Directories for dfs.name.dir and dfs.data.dir in conf/hdfs-site.xml, these are local directories used to hold distributed filesystem data on the master and slave nodes respectively. dfs.data.dir maybe contain a space-or-comma-separated list of directory names, so that the data may be stored on multiple local devices.
mapred.local.dir, local directory where temporary Map/Reduce data is stored, may be a list of directories too.
mapred.map.tasks and mapred.reduce.tasks in conf/mapred-site.xml. As a rule of thumb, use 10x the number of slave processors for mapred.map.tasks, and 2x the number of slave processors for mapred.reduce.tasks.
Finally, list all slave hostnames or IP addresses in your conf/slaves file, one per line. Then format your filesystem and start your cluster on your master node, as above.

8. References

Heavily referenced from
Hadoop: The Definitive Guide by Tom White
Copyright 2009 Tom White
Published by O'Reilly Media, Inc.

http://hadoop.apache.org/