MapReduce Tutorial

Table of contents

1. Purpose

This document comprehensively describes all user-facing facets of the Hadoop MapReduce framework and serves as a tutorial.

2. Prerequisites

Make sure Hadoop is installed, configured and running. See these guides:

- Single Node Setup for first-time users.
- Cluster Setup for large, distributed clusters.

3. Overview

Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

A MapReduce *job* usually splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the *reduce tasks*. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

Typically the compute nodes and the storage nodes are the same, that is, the MapReduce framework and the Hadoop Distributed File System (HDFS) are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

The MapReduce framework consists of a single master `JobTracker` and one slave `TaskTracker` per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

Minimally, applications specify the input/output locations and supply *map* and *reduce* functions via implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the *job configuration*. The Hadoop *job client* then submits the job (jar/executable etc.) and configuration to the `JobTracker` which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

Although the Hadoop framework is implemented in JavaTM, MapReduce applications need

not be written in Java.

- Hadoop Streaming is a utility which allows users to create and run jobs with any executables (e.g. shell utilities) as the mapper and/or the reducer.
- Hadoop Pipes is a SWIG- compatible *C++ API* to implement MapReduce applications (non JNITM based).

4. Inputs and Outputs

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

The `key` and `value` classes have to be serializable by the framework. Several serialization systems exists; the default serialization mechanism requires keys and values to implement the Writable interface. Additionally, the `key` classes must facilitate sorting by the framework; a straightforward means to do so is for them to implement the WritableComparable interface.

Input and Output types of a MapReduce job:

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

Note that the combine phase may run zero or more times in this process.

5. Example: WordCount v1.0

Before we jump into the details, lets walk through an example MapReduce application to get a flavour for how they work.

`WordCount` is a simple application that counts the number of occurences of each word in a given input set.

This example works with a pseudo-distributed (Single Node Setup) or fully-distributed (Cluster Setup) Hadoop installation.

5.1. Source Code

	WordCount.java
1.	`package org.myorg;`
2.	
3.	`import java.io.IOException;`

```
4.  import java.util.*;
5.
6.  import org.apache.hadoop.fs.Path;
7.  import org.apache.hadoop.conf.*;
8.  import org.apache.hadoop.io.*;
9.  import org.apache.hadoop.mapreduce.*;
10. import org.apache.hadoop.mapreduce.lib.input.*;
11. import org.apache.hadoop.mapreduce.lib.output.*;
12. import org.apache.hadoop.util.*;
13.
14. public class WordCount extends Configured impleme
15.
16.    public static class Map
17.         extends Mapper<LongWritable, Text, Text, I
18.       private final static IntWritable one = new I
19.       private Text word = new Text();
20.
21.       public void map(LongWritable key, Text value
22.           throws IOException, InterruptedException
23.         String line = value.toString();
24.         StringTokenizer tokenizer = new StringToke
25.         while (tokenizer.hasMoreTokens()) {
26.           word.set(tokenizer.nextToken());
27.           context.write(word, one);
28.         }
29.       }
30.    }
```

```
31.
32.     public static class Reduce
33.         extends Reducer<Text, IntWritable, Text, I
34.       public void reduce(Text key, Iterable<IntWri
35.           Context context) throws IOException, Int
36.
37.         int sum = 0;
38.         for (IntWritable val : values) {
39.           sum += val.get();
40.         }
41.         context.write(key, new IntWritable(sum));
42.       }
43.     }
44.
45.     public int run(String [] args) throws Exceptio
46.       Job job = new Job(getConf());
47.       job.setJarByClass(WordCount.class);
48.       job.setJobName("wordcount");
49.
50.       job.setOutputKeyClass(Text.class);
51.       job.setOutputValueClass(IntWritable.class);
52.
53.       job.setMapperClass(Map.class);
54.       job.setCombinerClass(Reduce.class);
55.       job.setReducerClass(Reduce.class);
56.
57.       job.setInputFormatClass(TextInputFormat.clas
```

```
58.     job.setOutputFormatClass(TextOutputFormat.cl
59.
60.     FileInputFormat.setInputPaths(job, new Path(
61.     FileOutputFormat.setOutputPath(job, new Path
62.
63.     boolean success = job.waitForCompletion(true
64.     return success ? 0 : 1;
65.   }
66.
67.   public static void main(String[] args) throws
68.     int ret = ToolRunner.run(new WordCount(), ar
69.     System.exit(ret);
70.   }
71. }
72.
```

5.2. Usage

Assuming `HADOOP_HOME` is the root of the installation and `HADOOP_VERSION` is the Hadoop version installed, compile `WordCount.java` and create a jar:

```
$ mkdir wordcount_classes
$ javac -classpath
${HADOOP_HOME}/hadoop-core-${HADOOP_VERSION}.jar:${HADOOP_HOME}/hadoop-ma
-d wordcount_classes WordCount.java
$ jar -cvf /user/joe/wordcount.jar -C wordcount_classes/ .
```

Assuming that:

- `/user/joe/wordcount/input` - input directory in HDFS
- `/user/joe/wordcount/output` - output directory in HDFS

Sample text-files as input:

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

```
/user/joe/wordcount/input/file01
/user/joe/wordcount/input/file02
$ bin/hadoop fs -cat /user/joe/wordcount/input/file01
Hello World Bye World
$ bin/hadoop fs -cat /user/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

Run the application:

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

Output:

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

5.3. Bundling a data payload with your application

Applications can specify a comma-separated list of paths which would be present in the current working directory of the task using the option `-files`. The `-libjars` option allows applications to add jars to the classpaths of the maps and reduces. The option `-archives` allows them to pass comma separated list of archives as arguments. These archives are unarchived and a link with name of the archive is created in the current working directory of tasks. The mechanism that provides this functionality is called the *distributed cache*. More details about the command line options surrounding job launching and control of the distributed cache are available at Hadoop Commands Guide.

Hadoop ships with some example code in a jar precompiled for you; one of these is (another) wordcount program. Here's an example invocation of the `wordcount` example with `-libjars`, `-files` and `-archives`:
`hadoop jar hadoop-examples.jar wordcount -files cachefile.txt -libjars mylib.jar -archives myarchive.zip input output` Here, myarchive.zip will be placed and unzipped into a directory by the name "myarchive.zip"

Users can specify a different symbolic name for files and archives passed through -files and -archives option, using #.

For example, `hadoop jar hadoop-examples.jar wordcount -files`

`dir1/dict.txt#dict1,dir2/dict.txt#dict2 -archives mytar.tgz#tgzdir input output` Here, the files dir1/dict.txt and dir2/dict.txt can be accessed by tasks using the symbolic names dict1 and dict2 respectively. And the archive mytar.tgz will be placed and unarchived into a directory by the name tgzdir.

The distributed cache is also described in greater detail further down in this tutorial.

5.4. Walk-through

This section describes the operation of the `WordCount` application shown earlier in this tutorial.

The `Mapper` implementation (lines 16-30), via the `map` method (lines 21-29), processes one line at a time, as provided by the specified `TextInputFormat` (line 57). It then splits the line into tokens separated by whitespaces, via the `StringTokenizer`, and emits a key-value pair of `< <word>, 1>`.

For the given sample input the first map emits:
```
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
```

The second map emits:
```
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>
```

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

`WordCount` also specifies a `combiner` (line 54). Hence, the output of each map is passed through the local combiner (which is same as the `Reducer` as per the job configuration) for local aggregation, after being sorted on the *key*s.

The output of the first map:
```
< Bye, 1>
< Hello, 1>
< World, 2>
```

The output of the second map:
```
< Goodbye, 1>
< Hadoop, 2>
```

```
< Hello, 1>
```

The Reducer implementation (lines 32-43), via the `reduce` method (lines 34-42) just sums up the values, which are the occurence counts for each key (i.e. words in this example).

Thus the output of the job is:

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

The `run` method specifies various facets of the job, such as the input/output paths (passed via the command line), key/value types, input/output formats etc., in the Job. It then calls the `Job.waitForCompletion()` (line 63) to submit the job to Hadoop and monitor its progress.

We'll learn more about Job, Mapper, Tool and other interfaces and classes a bit later in the tutorial.

6. MapReduce - User Interfaces

This section provides a reasonable amount of detail on every user-facing aspect of the MapReduce framwork. This should help users implement, configure and tune their jobs in a fine-grained manner. However, please note that the javadoc for each class/interface remains the most comprehensive documentation available; this is only meant to be a tutorial.

Let us first take the Mapper and Reducer classes. Applications typically extend them to provide the `map` and `reduce` methods.

We will then discuss other core classes including Job, Partitioner, Context, InputFormat, OutputFormat, OutputCommitter and others.

Finally, we will wrap up by discussing some useful features of the framework such as the `DistributedCache`, `IsolationRunner` etc.

6.1. Payload

Applications typically extend the `Mapper` and `Reducer` classes to provide the `map` and `reduce` methods. These form the core of the job.

6.1.1. Mapper

`Mapper` maps input key/value pairs to a set of intermediate key/value pairs.

Maps are the individual tasks that transform input records into intermediate records. The transformed intermediate records do not need to be of the same type as the input records. A given input pair may map to zero or many output pairs.

The Hadoop MapReduce framework spawns one map task for each `InputSplit` generated by the `InputFormat` for the job. An `InputSplit` is a logical representation of a unit of input work for a map task; e.g., a filename and a byte range within that file to process. The `InputFormat` is responsible for enumerating the `InputSplits`, and producing a `RecordReader` which will turn those logical work units into actual physical input records.

Overall, `Mapper` implementations are specified in the `Job`, a client-side class that describes the job's configuration and interfaces with the cluster on behalf of the client program. The `Mapper` itself then is instantiated in the running job, and is passed a `MapContext` object which it can use to configure itself. The `Mapper` contains a `run()` method which calls its `setup()` method once, its `map()` method for each input record, and finally its `cleanup()` method. All of these methods (including `run()` itself) can be overridden with your own code. If you do not override any methods (leaving even map as-is), it will act as the *identity function*, emitting each input record as a separate output.

The `Context` object allows the mapper to interact with the rest of the Hadoop system. It includes configuration data for the job, as well as interfaces which allow it to emit output. The `getConfiguration()` method returns a `Configuration` which contains configuration data for your program. You can set arbitrary (key, value) pairs of configuration data in your `Job`, e.g. with `Job.getConfiguration().set("myKey", "myVal")`, and then retrieve this data in your mapper with `Context.getConfiguration().get("myKey")`. This sort of functionality is typically done in the Mapper's `setup()` method.

The `Mapper.run()` method then calls `map(KeyInType, ValInType, Context)` for each key/value pair in the `InputSplit` for that task. Note that in the WordCount program's map() method, we then emit our output data via the `Context` argument, using its `write()` method.

Applications can then override the Mapper's `cleanup()` method to perform any required teardown operations.

Output pairs do not need to be of the same types as input pairs. A given input pair may map to zero or many output pairs. Output pairs are collected with calls to `Context.write(KeyOutType, ValOutType)`.

Applications can also use the `Context` to report progress, set application-level status

messages and update `Counters`, or just indicate that they are alive.

All intermediate values associated with a given output key are subsequently grouped by the framework, and passed to the `Reducer`(s) to determine the final output. Users can control the grouping by specifying a `Comparator` via `Job.setGroupingComparatorClass(Class)`. If a grouping comparator is not specified, then all values with the same key will be presented by an unordered `Iterable` to a call to the `Reducer.reduce()` method.

The `Mapper` outputs are sorted and partitioned per `Reducer`. The total number of partitions is the same as the number of reduce tasks for the job. Users can control which keys (and hence records) go to which `Reducer` by implementing a custom `Partitioner`.

Users can optionally specify a `combiner`, via `Job.setCombinerClass(Class)`, to perform local aggregation of the intermediate outputs, which helps to cut down the amount of data transferred from the `Mapper` to the `Reducer`.

The intermediate, sorted outputs are always stored in a simple (key-len, key, value-len, value) format. Applications can control if, and how, the intermediate outputs are to be compressed and the CompressionCodec to be used via the `Job`.

6.1.1.1. How Many Maps?

The number of maps is usually driven by the total size of the inputs, that is, the total number of blocks of the input files.

The right level of parallelism for maps seems to be around 10-100 maps per-node, although it has been set up to 300 maps for very cpu-light map tasks. Task setup takes awhile, so it is best if the maps take at least a minute to execute.

Thus, if you expect 10TB of input data and have a blocksize of `128MB`, you'll end up with 82,000 maps, unless the `mapreduce.job.maps` parameter (which only provides a hint to the framework) is used to set it even higher. Ultimately, the number of tasks is controlled by the number of splits returned by the `InputFormat.getSplits()` method (which you can override).

6.1.2. Reducer

`Reducer` reduces a set of intermediate values which share a key to a (usually smaller) set of values.

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

The API of `Reducer` is very similar to that of `Mapper`; there's a `run()` method that receives a `Context` containing the job's configuration as well as interfacing methods that return data from the reducer itself back to the framework. The `run()` method calls `setup()` once, `reduce()` once for each key associated with the reduce task, and `cleanup()` once at the end. Each of these methods can access the job's configuration data by using `Context.getConfiguration()`.

As in `Mapper`, any or all of these methods can be overridden with custom implementations. If none of these methods are overridden, the default reducer operation is the identity function; values are passed through without further processing.

The heart of `Reducer` is its `reduce()` method. This is called once per key; the second argument is an `Iterable` which returns all the values associated with that key. In the WordCount example, this is all of the 1's or other partial counts associated with a given word. The Reducer should emit its final output (key, value) pairs with the `Context.write()` method. It may emit 0, 1, or more (key, value) pairs for each input.

`Reducer` has 3 primary phases: shuffle, sort and reduce.

6.1.2.1. Shuffle

Input to the `Reducer` is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.

6.1.2.2. Sort

The framework groups `Reducer` inputs by keys (since different mappers may have output the same key) in this stage.

The shuffle and sort phases occur simultaneously; while map-outputs are being fetched they are merged.

Secondary Sort

If equivalence rules for grouping the intermediate keys are required to be different from those for grouping keys before reduction, then one may specify a `Comparator` via Job.setGroupingComparatorClass(Class). Since this can be used to control how intermediate keys are grouped, these can be used in conjunction to simulate *secondary sort on values*.

6.1.2.3. Reduce

In this phase the `reduce(MapOutKeyType, Iterable<MapOutValType>, Context)` method is called for each `<key, (list of values)>` pair in the grouped

inputs.

The output of the reduce task is typically written to the FileSystem via `Context.write(ReduceOutKeyType, ReduceOutValType)`.

Applications can use the `Context` to report progress, set application-level status messages and update `Counters`, or just indicate that they are alive.

The output of the `Reducer` is *not sorted*.

6.1.2.4. How Many Reduces?

The right number of reduces seems to be `0.95` or `1.75` multiplied by (<*no. of nodes*> * `mapreduce.tasktracker.reduce.tasks.maximum`).

With `0.95` all of the reduces can launch immediately and start transfering map outputs as the maps finish. With `1.75` the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.

Increasing the number of reduces increases the framework overhead, but increases load balancing and lowers the cost of failures.

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

6.1.2.5. Reducer NONE

It is legal to set the number of reduce-tasks to *zero* if no reduction is desired.

In this case the outputs of the map-tasks go directly to the `FileSystem`, into the output path set by setOutputPath(Path). The framework does not sort the map-outputs before writing them out to the `FileSystem`.

6.1.2.6. Mark-Reset

While applications iterate through the values for a given key, it is possible to mark the current position and later reset the iterator to this position and continue the iteration process. The corresponding methods are `mark()` and `reset()`.

`mark()` and `reset()` can be called any number of times during the iteration cycle. The `reset()` method will reset the iterator to the last record before a call to the previous `mark()`.

This functionality is available only with the new context based reduce iterator.

The following code snippet demonstrates the use of this functionality.

Source Code

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

   MarkableIterator<IntWritable> mitr = new
MarkableIterator<IntWritable>(values.iterator());

   // Mark the position
   mitr.mark();

   while (mitr.hasNext()) {
     i = mitr.next();
     // Do the necessary processing
   }

   // Reset
   mitr.reset();

   // Iterate all over again. Since mark was called before the first
   // call to mitr.next() in this example, we will iterate over all
   // the values now
   while (mitr.hasNext()) {
     i = mitr.next();
     // Do the necessary processing
   }

}
```

6.1.3. Partitioner

`Partitioner` partitions the key space.

Partitioner controls the partitioning of the keys of the intermediate map-outputs. The key (or a subset of the key) is used to derive the partition, typically by a *hash function*. The total number of partitions is the same as the number of reduce tasks for the job. Hence this controls which of the `m` reduce tasks the intermediate key (and hence the record) is sent to for reduction.

`HashPartitioner` is the default `Partitioner`.

6.1.4. Reporting Progress

Via the mapper or reducer's Context, MapReduce applications can report progress, set application-level status messages and update `Counters`.

`Mapper` and `Reducer` implementations can use the `Context` to report progress or just indicate that they are alive. In scenarios where the application takes a significant amount of time to process individual key/value pairs, this is crucial since the framework might assume that the task has timed-out and kill that task. Another way to avoid this is to set the configuration parameter `mapreduce.task.timeout` to a high-enough value (or even set it to *zero* for no time-outs).

Applications can also update `Counters` using the `Context`.

Hadoop MapReduce comes bundled with a library of generally useful mappers, reducers, and partitioners in the `org.apache.hadoop.mapreduce.lib` package.

6.2. Job Configuration

The `Job` represents a MapReduce job configuration. The actual state for this object is written to an underlying instance of Configuration.

`Job` is the primary interface for a user to describe a MapReduce job to the Hadoop framework for execution. The framework tries to faithfully execute the job as described by `Job`, however:

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

The `Job` is typically used to specify the `Mapper`, combiner (if any), `Partitioner`, `Reducer`, `InputFormat`, `OutputFormat` and `OutputCommitter` implementations.

`Job` also indicates the set of input files (setInputPaths(Job, Path...) /addInputPath(Job, Path)) and (setInputPaths(Job, String) /addInputPaths(Job, String)) and where the output files should be written (setOutputPath(Path)).

Optionally, `Job` is used to specify other advanced facets of the job such as the `Comparator` to be used, files to be put in the `DistributedCache`, whether intermediate and/or job outputs are to be compressed (and how), debugging via user-provided scripts, whether job tasks can be executed in a *speculative* manner (setMapSpeculativeExecution(boolean))/(setReduceSpeculativeExecution(boolean)) , maximum number of attempts per task (setMaxMapAttempts(int)/setMaxReduceAttempts(int)) , percentage of tasks failure which can be tolerated by the job (Job.getConfiguration().setInt(Job.MAP_FAILURES_MAX_PERCENT, int)/Job.getConfiguration().setInt(Job.REDUCE_FAILURES_MAX_PERCENT, int)), etc.

Of course, users can use `Job.getConfiguration()` to get access to the underlying configuration state, and can then use set(String, String)/get(String, String) to set/get arbitrary parameters needed by applications. However, use the `DistributedCache` for large amounts of (read-only) data.

6.3. Task Execution & Environment

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

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

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

```
<property>
 <name>mapreduce.map.java.opts</name>
 <value>
```

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

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

6.3.1. Configuring Memory Requirements For A Job

MapReduce tasks are launched with some default memory limits that are provided by the system or by the cluster's administrators. Memory intensive jobs might need to use more than these default values. Hadoop has some configuration options that allow these to be changed. Without such modifications, memory intensive jobs could fail due to `OutOfMemory` errors in tasks or could get killed when the limits are enforced by the system. This section describes the various options that can be used to configure specific memory requirements.

- `mapreduce.{map|reduce}.java.opts`: If the task requires more Java heap space, this option must be used. The value of this option should pass the desired heap using the JVM option -Xmx. For example, to use 1G of heap space, the option should be passed in as -Xmx1024m. Note that other JVM options are also passed using the same option. Hence, append the heap space option along with other options already configured.
- `mapreduce.{map|reduce}.ulimit`: The slaves where tasks are run could be configured with a ulimit value that applies a limit to every process that is launched on the slave. If the task, or any child that the task launches (like in streaming), requires more than the configured limit, this option must be used. The value is given in kilobytes. For example, to increase the ulimit to 1G, the option should be set to 1048576. Note that this value is a per process limit. Since it applies to the JVM as well, the heap space given to the JVM through the `mapreduce.{map|reduce}.java.opts` should be less than the value configured for the ulimit. Otherwise the JVM will not start.
- `mapreduce.{map|reduce}.memory.mb`: In some environments, administrators might have configured a total limit on the virtual memory used by the entire process tree for a task, including all processes launched recursively by the task or its children, like in

streaming. More details about this can be found in the section on Monitoring Task Memory Usage in the Cluster SetUp guide. If a task requires more virtual memory for its entire tree, this option must be used. The value is given in MB. For example, to set the limit to 1G, the option should be set to 1024. Note that this value does not automatically influence the per process ulimit or heap space. Hence, you may need to set those parameters as well (as described above) in order to give your tasks the right amount of memory.

- `mapreduce.{map|reduce}.memory.physical.mb`: This parameter is similar to `mapreduce.{map|reduce}.memory.mb`, except it specifies how much physical memory is required by a task for its entire tree of processes. The parameter is applicable if administrators have configured a total limit on the physical memory used by all MapReduce tasks.

As seen above, each of the options can be specified separately for map and reduce tasks. It is typically the case that the different types of tasks have different memory requirements. Hence different values can be set for the corresponding options.

The memory available to some parts of the framework is also configurable. In map and reduce tasks, performance may be influenced by adjusting parameters influencing the concurrency of operations and the frequency with which data will hit disk. Monitoring the filesystem counters for a job- particularly relative to byte counts from the map and into the reduce- is invaluable to the tuning of these parameters.

Note: The memory related configuration options described above are used only for configuring the launched child tasks from the tasktracker. Configuring the memory options for daemons is documented under Configuring the Environment of the Hadoop Daemons (Cluster Setup).

6.3.2. Map Parameters

A record emitted from a map and its metadata will be serialized into a buffer. As described in the following options, when the record data exceed a threshold, the contents of this buffer will be sorted and written to disk in the background (a "spill") while the map continues to output records. If the remainder of the buffer fills during the spill, the map thread will block. When the map is finished, any buffered records are written to disk and all on-disk segments are merged into a single file. Minimizing the number of spills to disk *can* decrease map time, but a larger buffer also decreases the memory available to the mapper.

Name	Type	Description
mapreduce.task.io.sort.mb	int	The cumulative size of the serialization and accounting buffers storing records emitted

		from the map, in megabytes.
mapreduce.map.sort.spill.percen	float	This is the threshold for the accounting and serialization buffer. When this percentage of the `io.sort.mb` has filled, its contents will be spilled to disk in the background. Note that a higher value may decrease the number of- or even eliminate-merges, but will also increase the probability of the map task getting blocked. The lowest average map times are usually obtained by accurately estimating the size of the map output and preventing multiple spills.

Other notes

- If the spill threshold is exceeded while a spill is in progress, collection will continue until the spill is finished. For example, if `mapreduce.map.sort.spill.percent` is set to 0.33, and the remainder of the buffer is filled while the spill runs, the next spill will include all the collected records, or 0.66 of the buffer, and will not generate additional spills. In other words, the thresholds are defining triggers, not blocking.
- A record larger than the serialization buffer will first trigger a spill, then be spilled to a separate file. It is undefined whether or not this record will first pass through the combiner.

6.3.3. Shuffle/Reduce Parameters

As described previously, each reduce fetches the output assigned to it by the Partitioner via HTTP into memory and periodically merges these outputs to disk. If intermediate compression of map outputs is turned on, each output is decompressed into memory. The following options affect the frequency of these merges to disk prior to the reduce and the memory allocated to map output during the reduce.

Name	Type	Description
mapreduce.task.io.sort.factor	int	Specifies the number of segments on disk to be merged at the same time. It limits the number of open files and compression codecs during the merge. If the number of files

		exceeds this limit, the merge will proceed in several passes. Though this limit also applies to the map, most jobs should be configured so that hitting this limit is unlikely there.
mapreduce.reduce.merge.inmem	int	The number of sorted map outputs fetched into memory before being merged to disk. Like the spill thresholds in the preceding note, this is not defining a unit of partition, but a trigger. In practice, this is usually set very high (1000) or disabled (0), since merging in-memory segments is often less expensive than merging from disk (see notes following this table). This threshold influences only the frequency of in-memory merges during the shuffle.
mapreduce.reduce.shuffle.merge	float	The memory threshold for fetched map outputs before an in-memory merge is started, expressed as a percentage of memory allocated to storing map outputs in memory. Since map outputs that can't fit in memory can be stalled, setting this high may decrease parallelism between the fetch and merge. Conversely, values as high as 1.0 have been effective for reduces whose input can fit entirely in memory. This parameter influences only the frequency of in-memory merges during the shuffle.
mapreduce.reduce.shuffle.input.b	float	The percentage of memory- relative to the maximum heapsize as typically specified in `mapreduce.reduce.java.opts`- that can be allocated to storing

		map outputs during the shuffle. Though some memory should be set aside for the framework, in general it is advantageous to set this high enough to store large and numerous map outputs.
mapreduce.reduce.input.buffer.p	float	The percentage of memory relative to the maximum heapsize in which map outputs may be retained during the reduce. When the reduce begins, map outputs will be merged to disk until those that remain are under the resource limit this defines. By default, all map outputs are merged to disk before the reduce begins to maximize the memory available to the reduce. For less memory-intensive reduces, this should be increased to avoid trips to disk.

Other notes

- If a map output is larger than 25 percent of the memory allocated to copying map outputs, it will be written directly to disk without first staging through memory.
- When running with a combiner, the reasoning about high merge thresholds and large buffers may not hold. For merges started before all map outputs have been fetched, the combiner is run while spilling to disk. In some cases, one can obtain better reduce times by spending resources combining map outputs- making disk spills small and parallelizing spilling and fetching- rather than aggressively increasing buffer sizes.
- When merging in-memory map outputs to disk to begin the reduce, if an intermediate merge is necessary because there are segments to spill and at least `mapreduce.task.io.sort.factor` segments already on disk, the in-memory map outputs will be part of the intermediate merge.

6.3.4. Directory Structure

The task tracker has local directory, `${mapreduce.cluster.local.dir}/taskTracker/` to create localized cache and localized job. It can define multiple local directories (spanning multiple disks) and then each filename is assigned to a semi-random local directory. When the job starts, task tracker

creates a localized job directory relative to the local directory specified in the configuration. Thus the task tracker directory structure looks as following:

- `${mapreduce.cluster.local.dir}/taskTracker/distcache/` : The public distributed cache for the jobs of all users. This directory holds the localized public distributed cache. Thus localized public distributed cache is shared among all the tasks and jobs of all users.
- `${mapreduce.cluster.local.dir}/taskTracker/$user/distcache/` : The private distributed cache for the jobs of the specific user. This directory holds the localized private distributed cache. Thus localized private distributed cache is shared among all the tasks and jobs of the specific user only. It is not accessible to jobs of other users.
- `${mapreduce.cluster.local.dir}/taskTracker/$user/jobcache/$jobid/` : The localized job directory
  - `${mapreduce.cluster.local.dir}/taskTracker/$user/jobcache/$jobid/wor` : The job-specific shared directory. The tasks can use this space as scratch space and share files among them. This directory is exposed to the users through the configuration property `mapreduce.job.local.dir`. It is available as System property also. So, users (streaming etc.) can call `System.getProperty("mapreduce.job.local.dir")` to access the directory.
  - `${mapreduce.cluster.local.dir}/taskTracker/$user/jobcache/$jobid/jar` : The jars directory, which has the job jar file and expanded jar. The `job.jar` is the application's jar file that is automatically distributed to each machine. Any library jars that are dependencies of the application code may be packaged inside this jar in a `lib/` directory. This directory is extracted from `job.jar` and its contents are automatically added to the classpath for each task. The job.jar location is accessible to the application through the API Job.getJar() . To access the unjarred directory, Job.getJar().getParent() can be called.
  - `${mapreduce.cluster.local.dir}/taskTracker/$user/jobcache/$jobid/job` : The job.xml file, the generic job configuration, localized for the job.
  - `${mapreduce.cluster.local.dir}/taskTracker/$user/jobcache/$jobid/$ta` : The task directory for each task attempt. Each task directory again has the following structure :
    - `${mapreduce.cluster.local.dir}/taskTracker/$user/jobcache/$jobid/` : A job.xml file, task localized job configuration, Task localization means that properties have been set that are specific to this particular task within the job. The properties localized for each task are described below.
    - `${mapreduce.cluster.local.dir}/taskTracker/$user/jobcache/$jobid/` : A directory for intermediate output files. This contains the temporary map reduce data generated by the framework such as map output files etc.

- `${mapreduce.cluster.local.dir}/taskTracker/$user/jobcache/$jobid/` : The curernt working directory of the task. With jvm reuse enabled for tasks, this directory will be the directory on which the jvm has started
- `${mapreduce.cluster.local.dir}/taskTracker/$user/jobcache/$jobid/` : The temporary directory for the task. (User can specify the property `mapreduce.task.tmp.dir` to set the value of temporary directory for map and reduce tasks. This defaults to `./tmp`. If the value is not an absolute path, it is prepended with task's working directory. Otherwise, it is directly assigned. The directory will be created if it doesn't exist. Then, the child java tasks are executed with option `-Djava.io.tmpdir='the absolute path of the tmp dir'`. Pipes and streaming are set with environment variable, `TMPDIR='the absolute path of the tmp dir'`). This directory is created, if `mapreduce.task.tmp.dir` has the value `./tmp`

6.3.5. Task JVM Reuse

Jobs can enable task JVMs to be reused by specifying the job configuration `mapreduce.job.jvm.numtasks`. If the value is 1 (the default), then JVMs are not reused (i.e. 1 task per JVM). If it is -1, there is no limit to the number of tasks a JVM can run (of the same job). One can also specify some value greater than 1 using the api `Job.getConfiguration().setInt(Job.JVM_NUM_TASKS_TO_RUN, int)`.

6.3.6. Configured Parameters

The following properties are localized in the job configuration for each task's execution:

Name	Type	Description
mapreduce.job.id	String	The job id
mapreduce.job.jar	String	job.jar location in job directory
mapreduce.job.local.dir	String	The job specific shared scratch space
mapreduce.task.id	String	The task id
mapreduce.task.attempt.id	String	The task attempt id
mapreduce.task.ismap	boolean	Is this a map task
mapreduce.task.partition	int	The id of the task within the job
mapreduce.map.input.file	String	The filename that the map is reading from

mapreduce.map.input.start	long	The offset of the start of the map input split
mapreduce.map.input.length	long	The number of bytes in the map input split
mapreduce.task.output.dir	String	The task's temporary output directory

Note: During the execution of a streaming job, the names of the "mapred" parameters are transformed. The dots (.) become underscores (_). For example, mapreduce.job.id becomes mapreduce.job.id and mapreduce.job.jar becomes mapreduce.job.jar. To get the values in a streaming job's mapper/reducer use the parameter names with the underscores.

6.3.7. Task Logs

The standard output (stdout) and error (stderr) streams of the task are read by the TaskTracker and logged to `${HADOOP_LOG_DIR}/userlogs`

6.3.8. Distributing Libraries

The DistributedCache can also be used to distribute both jars and native libraries for use in the map and/or reduce tasks. The child-jvm always has its *current working directory* added to the `java.library.path` and `LD_LIBRARY_PATH`. And hence the cached libraries can be loaded via System.loadLibrary or System.load. More details on how to load shared libraries through distributed cache are documented under Building Native Hadoop Libraries.

6.4. Job Submission and Monitoring

The `Job` is the primary interface by which user-job interacts with the `JobTracker`.

`Job` provides facilities to submit jobs, track their progress, access component-tasks' reports and logs, get the MapReduce cluster's status information and so on.

The job submission process involves:

1. Checking the input and output specifications of the job.
2. Computing the `InputSplit` values for the job.
3. Setting up the requisite accounting information for the `DistributedCache` of the job, if necessary.
4. Copying the job's jar and configuration to the MapReduce system directory on the `FileSystem`.
5. Submitting the job to the `JobTracker` and optionally monitoring it's status.

User can view the history log summary for a given history file using the following command

```
$ bin/hadoop job -history history-file
```
This command will print job details, failed and killed tip details.
More details about the job such as successful tasks and task attempts made for each task can be viewed using the following command
```
$ bin/hadoop job -history all history-file
```

User can use OutputLogFilter to filter log files from the output directory listing.

Normally the user creates the application, describes various facets of the job via `Job`, and then uses the `waitForCompletion()` method to submit the job and monitor its progress.

6.4.1. Job Control

Users may need to chain MapReduce jobs to accomplish complex tasks which cannot be done via a single MapReduce job. This is fairly easy since the output of the job typically goes to distributed file-system, and the output, in turn, can be used as the input for the next job.

However, this also means that the onus on ensuring jobs are complete (success/failure) lies squarely on the clients. In such cases, the various job-control options are:

- `Job.waitForCompletion()` : Submits the job and returns only after the job has completed.
- `Job.submit()` : Only submits the job;, then poll the other methods of `Job` such as `isComplete()`, `isSuccessful()`, etc. to query status and make scheduling decisions.
- `Job.getConfiguration().set(Job.END_NOTIFICATION_URL, String)` : Sets up a notification upon job-completion, thus avoiding polling.

6.4.2. Job Authorization

Job level authorization is enabled on the cluster, if the configuration `mapreduce.cluster.job-authorization-enabled` is set to true. When enabled, access control checks are done by the JobTracker and the TaskTracker before allowing users to view job details or to modify a job using Map/Reduce APIs, CLI or web user interfaces.

A job submitter can specify access control lists for viewing or modifying a job via the configuration properties `mapreduce.job.acl-view-job` and `mapreduce.job.acl-modify-job` respectively. By default, nobody is given access in these properties.

However, irrespective of the ACLs configured, a job's owner, the superuser and the members of an admin configured supergroup (`mapreduce.cluster.permissions.supergroup`) always have access to view

and modify a job.

A job view ACL authorizes users against the configured `mapreduce.job.acl-view-job` before returning possibly sensitive information about a job, like:

- job level counters
- task level counters
- tasks's diagnostic information
- task logs displayed on the TaskTracker web UI
- job.xml showed by the JobTracker's web UI

Other information about a job, like its status and its profile, is accessible to all users, without requiring authorization.

A job modification ACL authorizes users against the configured `mapreduce.job.acl-modify-job` before allowing modifications to jobs, like:

- killing a job
- killing/failing a task of a job
- setting the priority of a job

These operations are also protected by the queue level ACL, "acl-administer-jobs", configured via mapred-queue-acls.xml. The caller will be authorized against both queue level ACLs and job level ACLs, depending on what is enabled.

The format of a job level ACL is the same as the format for a queue level ACL as defined in the Cluster Setup documentation.

6.5. Job Input

InputFormat describes the input-specification for a MapReduce job.

The MapReduce framework relies on the `InputFormat` of the job to:

1. Validate the input-specification of the job.
2. Split-up the input file(s) into logical `InputSplit` instances, each of which is then assigned to an individual `Mapper`.
3. Provide the `RecordReader` implementation used to glean input records from the logical `InputSplit` for processing by the `Mapper`.

The default behavior of file-based `InputFormat` implementations, typically sub-classes of FileInputFormat, is to split the input into *logical* `InputSplit` instances based on the total size, in bytes, of the input files. However, the `FileSystem` blocksize of the input files is treated as an upper bound for input splits. A lower bound on the split size can be set via

`mapreduce.input.fileinputformat.split.minsize`.

Clearly, logical splits based on input-size is insufficient for many applications since record boundaries must be respected. In such cases, the application should implement a `RecordReader`, who is responsible for respecting record-boundaries and presents a record-oriented view of the logical `InputSplit` to the individual task.

TextInputFormat is the default `InputFormat`.

If `TextInputFormat` is the `InputFormat` for a given job, the framework detects input-files with the *.gz* extensions and automatically decompresses them using the appropriate `CompressionCodec`. However, it must be noted that compressed files with the above extensions cannot be *split* and each compressed file is processed in its entirety by a single mapper.

6.5.1. InputSplit

InputSplit represents the data to be processed by an individual `Mapper`.

Typically `InputSplit` presents a byte-oriented view of the input, and it is the responsibility of `RecordReader` to process and present a record-oriented view.

FileSplit is the default `InputSplit`. It sets `mapreduce.map.input.file` to the path of the input file for the logical split.

6.5.2. RecordReader

RecordReader reads `<key, value>` pairs from an `InputSplit`.

Typically the `RecordReader` converts the byte-oriented view of the input, provided by the `InputSplit`, and presents a record-oriented to the `Mapper` implementations for processing. `RecordReader` thus assumes the responsibility of processing record boundaries and presents the tasks with keys and values.

6.6. Job Output

OutputFormat describes the output-specification for a MapReduce job.

The MapReduce framework relies on the `OutputFormat` of the job to:

1. Validate the output-specification of the job; for example, check that the output directory doesn't already exist.
2. Provide the `RecordWriter` implementation used to write the output files of the job. Output files are stored in a `FileSystem`.

`TextOutputFormat` is the default `OutputFormat`.

6.6.1. Lazy Output Creation

It is possible to delay creation of output until the first write attempt by using LazyOutputFormat. This is particularly useful in preventing the creation of zero byte files when there is no call to output.collect (or Context.write). This is achieved by calling the static method `setOutputFormatClass` of `LazyOutputFormat` with the intended `OutputFormat` as the argument. The following example shows how to delay creation of files when using the `TextOutputFormat`

```
import
org.apache.hadoop.mapreduce.lib.output.LazyOutputFormat;
LazyOutputFormat.setOutputFormatClass(job,
TextOutputFormat.class);
```

6.6.2. OutputCommitter

OutputCommitter describes the commit of task output for a MapReduce job.

The MapReduce framework relies on the `OutputCommitter` of the job to:

1. Setup the job during initialization. For example, create the temporary output directory for the job during the initialization of the job. Job setup is done by a separate task when the job is in PREP state and after initializing tasks. Once the setup task completes, the job will be moved to RUNNING state.
2. Cleanup the job after the job completion. For example, remove the temporary output directory after the job completion. Job cleanup is done by a separate task at the end of the job. Job is declared SUCCEDED/FAILED/KILLED after the cleanup task completes.
3. Setup the task temporary output. Task setup is done as part of the same task, during task initialization.
4. Check whether a task needs a commit. This is to avoid the commit procedure if a task does not need commit.
5. Commit of the task output. Once task is done, the task will commit it's output if required.
6. Discard the task commit. If the task has been failed/killed, the output will be cleaned-up. If task could not cleanup (in exception block), a separate task will be launched with same attempt-id to do the cleanup.

`FileOutputCommitter` is the default `OutputCommitter`. Job setup/cleanup tasks occupy map or reduce slots, whichever is free on the TaskTracker. And JobCleanup task, TaskCleanup tasks and JobSetup task have the highest priority, and in that order.

6.6.3. Task Side-Effect Files

In some applications, component tasks need to create and/or write to side-files, which differ from the actual job-output files.

In such cases there could be issues with two instances of the same `Mapper` or `Reducer` running simultaneously (for example, speculative tasks) trying to open and/or write to the same file (path) on the `FileSystem`. Hence the application-writer will have to pick unique names per task-attempt (using the attemptid, say `attempt_200709221812_0001_m_000000_0`), not just per task.

To avoid these issues the MapReduce framework, when the `OutputCommitter` is `FileOutputCommitter`, maintains a special `${mapreduce.output.fileoutputformat.outputdir}/_temporary/_${taskid}` sub-directory accessible via `${mapreduce.task.output.dir}` for each task-attempt on the `FileSystem` where the output of the task-attempt is stored. On successful completion of the task-attempt, the files in the `${mapreduce.output.fileoutputformat.outputdir}/_temporary/_${taskid}` (only) are *promoted* to `${mapreduce.output.fileoutputformat.outputdir}`. Of course, the framework discards the sub-directory of unsuccessful task-attempts. This process is completely transparent to the application.

The application-writer can take advantage of this feature by creating any side-files required in `${mapreduce.task.output.dir}` during execution of a task via FileOutputFormat.getWorkOutputPath(), and the framework will promote them similarly for succesful task-attempts, thus eliminating the need to pick unique paths per task-attempt.

Note: The value of `${mapreduce.task.output.dir}` during execution of a particular task-attempt is actually `${mapreduce.output.fileoutputformat.outputdir}/_temporary/_{$taskid}`, and this value is set by the MapReduce framework. So, just create any side-files in the path returned by FileOutputFormat.getWorkOutputPath() from MapReduce task to take advantage of this feature.

The entire discussion holds true for maps of jobs with reducer=NONE (i.e. 0 reduces) since output of the map, in that case, goes directly to HDFS.

6.6.4. RecordWriter

RecordWriter writes the output `<key, value>` pairs to an output file.

RecordWriter implementations write the job outputs to the `FileSystem`.

6.7. Other Useful Features

6.7.1. Submitting Jobs to Queues

Users submit jobs to Queues. Queues, as collection of jobs, allow the system to provide specific functionality. For example, queues use ACLs to control which users who can submit jobs to them. Queues are expected to be primarily used by Hadoop Schedulers.

Hadoop comes configured with a single mandatory queue, called 'default'. Queue names are defined in the `mapred.queue.names` property of the Hadoop site configuration. Some job schedulers, such as the Capacity Scheduler, support multiple queues.

A job defines the queue it needs to be submitted to through the `mapreduce.job.queuename` property. Setting the queue name is optional. If a job is submitted without an associated queue name, it is submitted to the 'default' queue.

6.7.2. Counters

`Counters` represent global counters, defined either by the MapReduce framework or applications. Each `Counter` can be of any `Enum` type. Counters of a particular `Enum` are bunched into groups of type `Counters.Group`.

Applications can define arbitrary `Counters` (of type `Enum`); get a `Counter` object from the task's Context with the `getCounter()` method, and then call the `Counter.increment(long)` method to increment its value locally. These counters are then globally aggregated by the framework.

6.7.3. DistributedCache

DistributedCache distributes application-specific, large, read-only files efficiently.

`DistributedCache` is a facility provided by the MapReduce framework to cache files (text, archives, jars and so on) needed by applications.

Applications specify the files to be cached via urls (hdfs://) in the `Job`. The `DistributedCache` assumes that the files specified via hdfs:// urls are already present on the `FileSystem`.

The framework will copy the necessary files to the slave node before any tasks for the job are executed on that node. Its efficiency stems from the fact that the files are only copied once per job and the ability to cache archives which are un-archived on the slaves.

`DistributedCache` tracks the modification timestamps of the cached files. Clearly the

cache files should not be modified by the application or externally while the job is executing.

`DistributedCache` can be used to distribute simple, read-only data/text files and more complex types such as archives and jars. Archives (zip, tar, tgz and tar.gz files) are *un-archived* at the slave nodes. Files have *execution permissions* set.

The files/archives can be distributed by setting the property `mapred.cache.{files|archives}`. If more than one file/archive has to be distributed, they can be added as comma separated paths. The properties can also be set by APIs DistributedCache.addCacheFile(URI,conf)/ DistributedCache.addCacheArchive(URI,conf) and DistributedCache.setCacheFiles(URIs,conf)/ DistributedCache.setCacheArchives(URIs,conf) where URI is of the form `hdfs://host:port/absolute-path#link-name`. In Streaming, the files can be distributed through command line option `-cacheFile/-cacheArchive`.

Optionally users can also direct the `DistributedCache` to *symlink* the cached file(s) into the `current working directory` of the task via the DistributedCache.createSymlink(Configuration) api. Or by setting the configuration property `mapreduce.job.cache.symlink.create` as `yes`. The DistributedCache will use the `fragment` of the URI as the name of the symlink. For example, the URI `hdfs://namenode:port/lib.so.1#lib.so` will have the symlink name as `lib.so` in task's cwd for the file `lib.so.1` in distributed cache.

The `DistributedCache` can also be used as a rudimentary software distribution mechanism for use in the map and/or reduce tasks. It can be used to distribute both jars and native libraries. The DistributedCache.addArchiveToClassPath(Path, Configuration) or DistributedCache.addFileToClassPath(Path, Configuration) api can be used to cache files/jars and also add them to the *classpath* of child-jvm. The same can be done by setting the configuration properties `mapreduce.job.classpath.{files|archives}`. Similarly the cached files that are symlinked into the working directory of the task can be used to distribute native libraries and load them.

6.7.3.1. Private and Public DistributedCache Files

DistributedCache files can be private or public, that determines how they can be shared on the slave nodes.

- "Private" DistributedCache files are cached in a local directory private to the user whose jobs need these files. These files are shared by all tasks and jobs of the specific user only and cannot be accessed by jobs of other users on the slaves. A DistributedCache file becomes private by virtue of its permissions on the file system where the files are uploaded, typically HDFS. If the file has no world readable access, or if the directory

path leading to the file has no world executable access for lookup, then the file becomes private.

- "Public" DistributedCache files are cached in a global directory and the file access is setup such that they are publicly visible to all users. These files can be shared by tasks and jobs of all users on the slaves. A DistributedCache file becomes public by virtue of its permissions on the file system where the files are uploaded, typically HDFS. If the file has world readable access, AND if the directory path leading to the file has world executable access for lookup, then the file becomes public. In other words, if the user intends to make a file publicly available to all users, the file permissions must be set to be world readable, and the directory permissions on the path leading to the file must be world executable.

The `DistributedCache` tracks modification timestamps of the cache files/archives. Clearly the cache files/archives should not be modified by the application or externally while the job is executing.

Here is an illustrative example on how to use the `DistributedCache`:
// Setting up the cache for the application 1. Copy the requisite files to the `FileSystem`:

```
$ bin/hadoop fs -copyFromLocal lookup.dat /myapp/lookup.dat
$ bin/hadoop fs -copyFromLocal map.zip /myapp/map.zip
$ bin/hadoop fs -copyFromLocal mylib.jar /myapp/mylib.jar
$ bin/hadoop fs -copyFromLocal mytar.tar /myapp/mytar.tar
$ bin/hadoop fs -copyFromLocal mytgz.tgz /myapp/mytgz.tgz
$ bin/hadoop fs -copyFromLocal mytargz.tar.gz
/myapp/mytargz.tar.gz
```

2. Setup the job

```
Job job = new Job(conf);
job.addCacheFile(new URI("/myapp/lookup.dat#lookup.dat"));
job.addCacheArchive(new URI("/myapp/map.zip"));
job.addFileToClassPath(new Path("/myapp/mylib.jar"));
job.addCacheArchive(new URI("/myapp/mytar.tar"));
job.addCacheArchive(new URI("/myapp/mytgz.tgz"));
job.addCacheArchive(new URI("/myapp/mytargz.tar.gz"));
```

3. Use the cached files in the `{@link org.apache.hadoop.mapreduce.Mapper}` or `{@link org.apache.hadoop.mapreduce.Reducer}`:

```
public static class MapClass extends Mapper<K, V, K, V> {
  private Path[] localArchives;
  private Path[] localFiles;
  public void setup(Context context) {
    // Get the cached archives/files
    localArchives = context.getLocalCacheArchives();
```

```
    localFiles = context.getLocalCacheFiles();
  }
  public void map(K key, V value, Context context) throws
IOException {
    // Use data from the cached archives/files here
    // ...
    // ...
    context.write(k, v);
  }
}
```

6.7.4. Tool

The Tool interface supports the handling of generic Hadoop command-line options.

`Tool` is the standard for any MapReduce tool or application. The application should delegate the handling of standard command-line options to GenericOptionsParser via ToolRunner.run(Tool, String[]) and only handle its custom arguments.

The generic Hadoop command-line options are:

```
-conf <configuration file>
-D <property=value>
-fs <local|namenode:port>
-jt <local|jobtracker:port>
```

6.7.5. IsolationRunner

IsolationRunner is a utility to help debug MapReduce programs.

To use the `IsolationRunner`, first set `keep.failed.tasks.files` to `true` (also see `keep.tasks.files.pattern`).

Next, go to the node on which the failed task ran and go to the `TaskTracker`'s local directory and run the `IsolationRunner`:

```
$ cd <local path>
/taskTracker/$user/jobcache/$jobid/${taskid}/work
$ bin/hadoop org.apache.hadoop.mapred.IsolationRunner
../job.xml
```

`IsolationRunner` will run the failed task in a single jvm, which can be in the debugger, over precisely the same input.

6.7.6. Profiling

Profiling is a utility to get a representative (2 or 3) sample of built-in java profiler for a sample of maps and reduces.

User can specify whether the system should collect profiler information for some of the tasks in the job by setting the configuration property `mapreduce.task.profile`. The value can be set using the api Job.setProfileEnabled(boolean). If the value is set `true`, the task profiling is enabled. The profiler information is stored in the user log directory. By default, profiling is not enabled for the job.

Once user configures that profiling is needed, she/he can use the configuration property `mapreduce.task.profile.{maps|reduces}` to set the ranges of MapReduce tasks to profile. The value can be set using the api Job.setProfileTaskRange(boolean,String). By default, the specified range is `0-2`.

User can also specify the profiler configuration arguments by setting the configuration property `mapreduce.task.profile.params`. The value can be specified using the api Job.setProfileParams(String). If the string contains a `%s`, it will be replaced with the name of the profiling output file when the task runs. These parameters are passed to the task child JVM on the command line. The default value for the profiling parameters is `-agentlib:hprof=cpu=samples,heap=sites,force=n,thread=y,verbose=n,file=%s`

6.7.7. Debugging

The MapReduce framework provides a facility to run user-provided scripts for debugging. When a MapReduce task fails, a user can run a debug script, to process task logs for example. The script is given access to the task's stdout and stderr outputs, syslog and jobconf. The output from the debug script's stdout and stderr is displayed on the console diagnostics and also as part of the job UI.

In the following sections we discuss how to submit a debug script with a job. The script file needs to be distributed and submitted to the framework.

6.7.7.1. How to distribute the script file:

The user needs to use DistributedCache to *distribute* and *symlink* the script file.

6.7.7.2. How to submit the script:

A quick way to submit the debug script is to set values for the properties `mapreduce.map.debug.script` and `mapreduce.reduce.debug.script`, for

debugging map and reduce tasks respectively. These properties can also be set by using APIs `Job.getConfiguration().set(Job.MAP_DEBUG_SCRIPT, String)` and `Job.getConfiguration().set(Job.REDUCE_DEBUG_SCRIPT, String)`. In streaming mode, a debug script can be submitted with the command-line options `-mapdebug` and `-reducedebug`, for debugging map and reduce tasks respectively.

The arguments to the script are the task's stdout, stderr, syslog and jobconf files. The debug command, run on the node where the MapReduce task failed, is:

```
$script $stdout $stderr $syslog $jobconf
```

Pipes programs have the c++ program name as a fifth argument for the command. Thus for the pipes programs the command is

```
$script $stdout $stderr $syslog $jobconf $program
```

6.7.7.3. Default Behavior:

For pipes, a default script is run to process core dumps under gdb, prints stack trace and gives info about running threads.

6.7.8. JobControl

JobControl is a utility which encapsulates a set of MapReduce jobs and their dependencies.

6.7.9. Data Compression

Hadoop MapReduce provides facilities for the application-writer to specify compression for both intermediate map-outputs and the job-outputs i.e. output of the reduces. It also comes bundled with CompressionCodec implementation for the zlib compression algorithm. The gzip file format is also supported.

Hadoop also provides native implementations of the above compression codecs for reasons of both performance (zlib) and non-availability of Java libraries. For more information see the Native Libraries Guide.

6.7.9.1. Intermediate Outputs

Applications can control compression of intermediate map-outputs via the `Job.getConfiguration().setBoolean(Job.MAP_OUTPUT_COMPRESS, bool)` api and the `CompressionCodec` to be used via the `Job.getConfiguration().setClass(Job.MAP_OUTPUT_COMPRESS_CODEC, Class, CompressionCodec.class)` api.

6.7.9.2. Job Outputs

Applications can control compression of job-outputs via the FileOutputFormat.setCompressOutput(Job, boolean) api and the `CompressionCodec` to be used can be specified via the FileOutputFormat.setOutputCompressorClass(Job, Class) api.

If the job outputs are to be stored in the SequenceFileOutputFormat, the required `SequenceFile.CompressionType` (i.e. `RECORD` / `BLOCK` - defaults to `RECORD`) can be specified via the SequenceFileOutputFormat.setOutputCompressionType(Job, SequenceFile.CompressionType) api.

6.7.10. Skipping Bad Records

Hadoop provides an option where a certain set of bad input records can be skipped when processing map inputs. Applications can control this feature through the SkipBadRecords class.

This feature can be used when map tasks crash deterministically on certain input. This usually happens due to bugs in the map function. Usually, the user would have to fix these bugs. This is, however, not possible sometimes. The bug may be in third party libraries, for example, for which the source code is not available. In such cases, the task never completes successfully even after multiple attempts, and the job fails. With this feature, only a small portion of data surrounding the bad records is lost, which may be acceptable for some applications (those performing statistical analysis on very large data, for example).

By default this feature is disabled. For enabling it, refer to SkipBadRecords.setMapperMaxSkipRecords(Configuration, long) and SkipBadRecords.setReducerMaxSkipGroups(Configuration, long).

With this feature enabled, the framework gets into 'skipping mode' after a certain number of map failures. For more details, see SkipBadRecords.setAttemptsToStartSkipping(Configuration, int). In 'skipping mode', map tasks maintain the range of records being processed. To do this, the framework relies on the processed record counter. See SkipBadRecords.COUNTER_MAP_PROCESSED_RECORDS and SkipBadRecords.COUNTER_REDUCE_PROCESSED_GROUPS. This counter enables the framework to know how many records have been processed successfully, and hence, what record range caused a task to crash. On further attempts, this range of records is skipped.

The number of records skipped depends on how frequently the processed record counter is incremented by the application. It is recommended that this counter be incremented after

every record is processed. This may not be possible in some applications that typically batch their processing. In such cases, the framework may skip additional records surrounding the bad record. Users can control the number of skipped records through SkipBadRecords.setMapperMaxSkipRecords(Configuration, long) and SkipBadRecords.setReducerMaxSkipGroups(Configuration, long). The framework tries to narrow the range of skipped records using a binary search-like approach. The skipped range is divided into two halves and only one half gets executed. On subsequent failures, the framework figures out which half contains bad records. A task will be re-executed till the acceptable skipped value is met or all task attempts are exhausted. To increase the number of task attempts, use Job.setMaxMapAttempts(int) and Job.setMaxReduceAttempts(int).

Skipped records are written to HDFS in the sequence file format, for later analysis. The location can be changed through SkipBadRecords.setSkipOutputPath(conf, Path).

7. Example: WordCount v2.0

Here is a more complete `WordCount` which uses many of the features provided by the MapReduce framework we discussed so far.

This example needs the HDFS to be up and running, especially for the `DistributedCache`-related features. Hence it only works with a pseudo-distributed (Single Node Setup) or fully-distributed (Cluster Setup) Hadoop installation.

7.1. Source Code

	WordCount2.java
1.	`package org.myorg;`
2.	
3.	`import java.io.*;`
4.	`import java.util.*;`
5.	
6.	`import org.apache.hadoop.fs.Path;`
7.	`import org.apache.hadoop.filecache.DistributedCac`
8.	`import org.apache.hadoop.conf.*;`
9.	`import org.apache.hadoop.io.*;`
10.	`import org.apache.hadoop.mapreduce.*;`

```
11. import org.apache.hadoop.mapreduce.lib.input.*;
12. import org.apache.hadoop.mapreduce.lib.output.*;
13. import org.apache.hadoop.util.*;
14.
15. public class WordCount2 extends Configured implem
16.
17.    public static class Map
18.        extends Mapper<LongWritable, Text, Text, I
19.
20.      static enum Counters { INPUT_WORDS }
21.
22.      private final static IntWritable one = new I
23.      private Text word = new Text();
24.
25.      private boolean caseSensitive = true;
26.      private Set<String> patternsToSkip = new Has
27.
28.      private long numRecords = 0;
29.      private String inputFile;
30.
31.      public void setup(Context context) {
32.        Configuration conf = context.getConfigurat
33.        caseSensitive = conf.getBoolean("wordcount
34.        inputFile = conf.get("mapreduce.map.input.
35.
36.        if (conf.getBoolean("wordcount.skip.patter
37.          Path[] patternsFiles = new Path[0];
```

```
38.         try {
39.           patternsFiles = DistributedCache.getLo
40.         } catch (IOException ioe) {
41.           System.err.println("Caught exception w
42.               + StringUtils.stringifyException(i
43.         }
44.         for (Path patternsFile : patternsFiles)
45.           parseSkipFile(patternsFile);
46.         }
47.       }
48.     }
49.
50.     private void parseSkipFile(Path patternsFile
51.       try {
52.         BufferedReader fis = new BufferedReader(
53.             patternsFile.toString()));
54.         String pattern = null;
55.         while ((pattern = fis.readLine()) != nul
56.           patternsToSkip.add(pattern);
57.         }
58.       } catch (IOException ioe) {
59.         System.err.println("Caught exception whi
60.             + patternsFile + "' : " + StringUtil
61.       }
62.     }
63.
64.     public void map(LongWritable key, Text value
```

```
65.           throws IOException, InterruptedException
66.       String line = (caseSensitive) ?
67.           value.toString() : value.toString().to
68.
69.       for (String pattern : patternsToSkip) {
70.         line = line.replaceAll(pattern, "");
71.       }
72.
73.       StringTokenizer tokenizer = new StringToke
74.       while (tokenizer.hasMoreTokens()) {
75.         word.set(tokenizer.nextToken());
76.         context.write(word, one);
77.         context.getCounter(Counters.INPUT_WORDS)
78.       }
79.
80.       if ((++numRecords % 100) == 0) {
81.         context.setStatus("Finished processing "
82.             + " records " + "from the input file
83.       }
84.     }
85.   }
86.
87.   public static class Reduce
88.       extends Reducer<Text, IntWritable, Text, I
89.     public void reduce(Text key, Iterable<IntWri
90.         Context context) throws IOException, Int
91.
```

```
92.       int sum = 0;
93.       for (IntWritable val : values) {
94.         sum += val.get();
95.       }
96.       context.write(key, new IntWritable(sum));
97.     }
98.   }
99.
100.  public int run(String[] args) throws Exception
101.    Job job = new Job(getConf());
102.    job.setJarByClass(WordCount2.class);
103.    job.setJobName("wordcount2.0");
104.
105.    job.setOutputKeyClass(Text.class);
106.    job.setOutputValueClass(IntWritable.class);
107.
108.    job.setMapperClass(Map.class);
109.    job.setCombinerClass(Reduce.class);
110.    job.setReducerClass(Reduce.class);
111.
112.    // Note that these are the default.
113.    job.setInputFormatClass(TextInputFormat.clas
114.    job.setOutputFormatClass(TextOutputFormat.cl
115.
116.    List<String> other_args = new ArrayList<Stri
117.    for (int i=0; i < args.length; ++i) {
118.      if ("-skip".equals(args[i])) {
```

```
119.            DistributedCache.addCacheFile(new Path(a
120.                job.getConfiguration());
121.            job.getConfiguration().setBoolean("wordc
122.          } else {
123.            other_args.add(args[i]);
124.          }
125.        }
126.
127.        FileInputFormat.setInputPaths(job, new Path(
128.        FileOutputFormat.setOutputPath(job, new Path
129.
130.        boolean success = job.waitForCompletion(true
131.        return success ? 0 : 1;
132.      }
133.
134.      public static void main(String[] args) throws
135.        int res = ToolRunner.run(new Configuration()
136.        System.exit(res);
137.      }
138.    }
```

7.2. Sample Runs

Sample text-files as input:

```
$ bin/hadoop fs -ls /user/joe/wordcount/input/
/user/joe/wordcount/input/file01
/user/joe/wordcount/input/file02
$ bin/hadoop fs -cat /user/joe/wordcount/input/file01
Hello World, Bye World!
```

```
$ bin/hadoop fs -cat /user/joe/wordcount/input/file02
Hello Hadoop, Goodbye to hadoop.
```

Run the application:

```
$ bin/hadoop jar /user/joe/wordcount.jar org.myorg.WordCount2
/user/joe/wordcount/input /user/joe/wordcount/output
```

Output:

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

Notice that the inputs differ from the first version we looked at, and how they affect the outputs.

Now, lets plug-in a pattern-file which lists the word-patterns to be ignored, via the `DistributedCache`.

```
$ hadoop fs -cat /user/joe/wordcount/patterns.txt
\.
\,
\!
to
```

Run it again, this time with more options:

```
$ bin/hadoop jar /user/joe/wordcount.jar org.myorg.WordCount2
-Dwordcount.case.sensitive=true /user/joe/wordcount/input
/user/joe/wordcount/output -skip
/user/joe/wordcount/patterns.txt
```

As expected, the output:

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

```
Hello 2
World 2
hadoop 1
```

Run it once more, this time switch-off case-sensitivity:

```
$ bin/hadoop jar /user/joe/wordcount.jar org.myorg.WordCount2
-Dwordcount.case.sensitive=false /user/joe/wordcount/input
/user/joe/wordcount/output -skip
/user/joe/wordcount/patterns.txt
```

Sure enough, the output:

```
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000
bye 1
goodbye 1
hadoop 2
hello 2
world 2
```

7.3. Highlights

The second version of `WordCount` improves upon the previous one by using some features offered by the MapReduce framework:

- Demonstrates how applications can access configuration parameters in the `setup` method of the `Mapper` (and `Reducer`) implementations (lines 31-48).
- Demonstrates how the `DistributedCache` can be used to distribute read-only data needed by the jobs. Here it allows the user to specify word-patterns to skip while counting (line 119).
- Demonstrates the utility of the `Tool` interface and the `GenericOptionsParser` to handle generic Hadoop command-line options (line 135).
- Demonstrates how applications can use `Counters` (line 77) and how they can set application-specific status information via the `Context` instance passed to the `map` (and `reduce`) method (line 81).

Java and JNI are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.