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

Ensure that Hadoop is installed, configured and is running. More details:

- 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 (see HDFS Architecture Guide) 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 and hence need to implement the Writable interface. Additionally, the `key` classes have to implement the WritableComparable interface to facilitate sorting by the framework.

Input and Output types of a MapReduce job:

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

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 works with a local-standalone, pseudo-distributed or fully-distributed Hadoop installation (Single Node Setup).

5.1. Source Code

WordCount.java

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {

  public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
      }
    }
```

```
 }

 public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
   int sum = 0;
   while (values.hasNext()) {
    sum += values.next().get();
   }
   output.collect(key, new IntWritable(sum));
  }
 }

 public static void main(String[] args) throws Exception {
  JobConf conf = new JobConf(WordCount.class);
  conf.setJobName("wordcount");

  conf.setOutputKeyClass(Text.class);
  conf.setOutputValueClass(IntWritable.class);

  conf.setMapperClass(Map.class);
```

```
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }
}

```

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
${HADOOP_HOME}/hadoop-${HADOOP_VERSION}-core.jar -d
wordcount_classes WordCount.java
$ jar -cvf /usr/joe/wordcount.jar -C wordcount_classes/ .
```

Assuming that:

- `/usr/joe/wordcount/input` - input directory in HDFS

- `/usr/joe/wordcount/output` - output directory in HDFS

Sample text-files as input:

```
$ bin/hadoop dfs -ls /usr/joe/wordcount/input/
/usr/joe/wordcount/input/file01
/usr/joe/wordcount/input/file02
$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file01
Hello World Bye World
$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

Run the application:

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

Output:

```
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-00000
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```

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. More details about the command line options are available at Commands Guide.

Running `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. The archive mytar.tgz will be placed and unarchived into a directory by the name "tgzdir".

5.3. Walk-through

The WordCount application is quite straight-forward.

The Mapper implementation (lines 14-26), via the map method (lines 18-25), processes one line at a time, as provided by the specified TextInputFormat (line 49). 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 46). 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 28-36), via the reduce method (lines 29-35) 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 `JobConf`. It then calls the `JobClient.runJob` (line 55) to submit the and monitor its progress.

We'll learn more about `JobConf`, `JobClient`, `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 framework. 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` interfaces. Applications typically implement them to provide the `map` and `reduce` methods.

We will then discuss other core interfaces including `JobConf`, `JobClient`, `Partitioner`, `OutputCollector`, `Reporter`, `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 implement the `Mapper` and `Reducer` interfaces 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.

Overall, `Mapper` implementations are passed the `JobConf` for the job via the JobConfigurable.configure(JobConf) method and override it to initialize themselves. The framework then calls map(WritableComparable, Writable, OutputCollector, Reporter) for each key/value pair in the `InputSplit` for that task. Applications can then override the Closeable.close() method to perform any required cleanup.

Output pairs do not need to be of the same types as input pairs. A given input pair may map to zero or many output pairs. Output pairs are collected with calls to OutputCollector.collect(WritableComparable,Writable).

Applications can use the `Reporter` 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 JobConf.setOutputKeyComparatorClass(Class).

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

Users can optionally specify a `combiner`, via JobConf.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 `JobConf`.

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 setNumMapTasks(int) (which only provides a hint to the framework) is used to set it even higher.

6.1.2. Reducer

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

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

Overall, `Reducer` implementations are passed the `JobConf` for the job via the JobConfigurable.configure(JobConf) method and can override it to initialize themselves. The framework then calls reduce(WritableComparable, Iterator, OutputCollector, Reporter) method for each `<key, (list of values)>` pair in the grouped inputs. Applications can then override the Closeable.close() method to perform any required cleanup.

`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 JobConf.setOutputValueGroupingComparator(Class). Since JobConf.setOutputKeyComparatorClass(Class) 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(WritableComparable, Iterator, OutputCollector, Reporter) 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 OutputCollector.collect(WritableComparable, Writable).

Applications can use the `Reporter` 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*> * `mapred.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`.

6.1.3. 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. Reporter

Reporter is a facility for MapReduce applications to report progress, set application-level status messages and update `Counters`.

Mapper and Reducer implementations can use the Reporter 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 mapred.task.timeout to a high-enough value (or even set it to *zero* for no time-outs).

Applications can also update Counters using the Reporter.

6.1.5. OutputCollector

OutputCollector is a generalization of the facility provided by the MapReduce framework to collect data output by the Mapper or the Reducer (either the intermediate outputs or the output of the job).

Hadoop MapReduce comes bundled with a library of generally useful mappers, reducers, and partitioners.

6.2. Job Configuration

JobConf represents a MapReduce job configuration.

JobConf 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 JobConf, however:

- f 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. setNumMapTasks(int)).

JobConf is typically used to specify the Mapper, combiner (if any), Partitioner, Reducer, InputFormat, OutputFormat and OutputCommitter implementations. JobConf also indicates the set of input files (setInputPaths(JobConf, Path...) /addInputPath(JobConf, Path)) and (setInputPaths(JobConf, String) /addInputPaths(JobConf, String)) and where the output files should be written (setOutputPath(Path)).

Optionally, JobConf 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 (setMapDebugScript(String)/setReduceDebugScript(String)) , whether job tasks can be executed in a *speculative* manner setReduceSpeculativeExecution(boolean)) , maximum number of attempts per task (setMaxMapAttempts(int)/setMaxReduceAttempts(int)) , percentage of tasks failure which can be tolerated by the job (setMaxMapTaskFailuresPercent(int)/setMaxReduceTaskFailuresPercent(int)) etc.

Of course, users can 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 `JobConf` 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>mapred.map.child.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>mapred.reduce.child.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. Memory Management

Users/admins can also specify the maximum virtual memory of the launched child-task, and any sub-process it launches recursively, using `mapred.{map|reduce}.child.ulimit`. Note that the value set here is a per process limit. The value for `mapred.{map|reduce}.child.ulimit` should be specified in kilo bytes (KB). And also the value must be greater than or equal to the -Xmx passed to JavaVM, else the VM might not start.

Note: `mapred.{map|reduce}.child.java.opts` are used only for configuring the launched child tasks from task tracker. Configuring the memory options for daemons is documented in Configuring the Environment of the Hadoop Daemons.

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.

Users can choose to override default limits of Virtual Memory and RAM enforced by the task tracker, if memory management is enabled. Users can set the following parameter per job:

Name	Type	Description
`mapred.task.maxvmem`	int	A number, in bytes, that represents the maximum Virtual Memory task-limit for each task of the job. A task will be killed if it consumes more Virtual Memory than this number.
mapred.task.maxpmem	int	A number, in bytes, that represents the maximum RAM task-limit for each task of the job. This number can be optionally used by Schedulers to prevent over-scheduling of

		tasks on a node based on RAM needs.

6.3.2. Map Parameters

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

Name	Type	Description
io.sort.mb	int	The cumulative size of the serialization and accounting buffers storing records emitted from the map, in megabytes.
io.sort.record.percent	float	The ratio of serialization to accounting space can be adjusted. Each serialized record requires 16 bytes of accounting information in addition to its serialized size to effect the sort. This percentage of space allocated from `io.sort.mb` affects the probability of a spill to disk being caused by either exhaustion of the serialization buffer or the accounting space. Clearly, for a map outputting small records, a higher value than the default will likely decrease the number of spills to disk.
io.sort.spill.percent	float	This is the threshold for the accounting and serialization buffers. When this percentage of either buffer has filled, their contents will be spilled to disk in the background. Let `io.sort.record.percent`

		be *r*, `io.sort.mb` be *x*, and this value be *q*. The maximum number of records collected before the collection thread will spill is `r * x * q * 2^16`. 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 either spill threshold is exceeded while a spill is in progress, collection will continue until the spill is finished. For example, if `io.sort.buffer.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
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.
mapred.inmem.merge.threshold	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.
mapred.job.shuffle.merge.percer	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.
mapred.job.shuffle.input.buffer.pe	float	The percentage of memory- relative to the maximum heapsize as typically specified in `mapred.reduce.child.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.
mapred.job.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 `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, `${mapred.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:

- `${mapred.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.
- `${mapred.local.dir}/taskTracker/$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.
- `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/` : The localized job directory
    - `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/work/` : 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 `job.local.dir`. The directory can accessed through the API JobConf.getJobLocalDir(). It is available as System property also. So, users (streaming etc.) can call `System.getProperty("job.local.dir")` to access the directory.
    - `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/jars/` : The jars directory, which has the job jar file and expanded jar. The `job.jar` is the application's jar file that is automatically distributed to each machine. It is expanded in jars directory before the tasks for the job start. The job.jar location is accessible to the application through the api JobConf.getJar() . To access the unjarred directory, JobConf.getJar().getParent() can be called.
    - `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/job.xml` : The job.xml file, the generic job configuration, localized for the job.
    - `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/$taskid` : The task directory for each task attempt. Each task directory again has the following structure :
        - `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/$taskid/job` : 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.
        - `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/$taskid/out` : A directory for intermediate output files. This contains the temporary map reduce data generated by the framework such as map output files etc.
        - `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/$taskid/wor` : The current working directory of the task. With jvm reuse enabled for tasks, this directory will be the directory on which the jvm has started
        - `${mapred.local.dir}/taskTracker/$user/jobcache/$jobid/$taskid/wor` : The temporary directory for the task. (User can specify the property `mapred.child.tmp` 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 `mapred.child.tmp` has the value `./tmp`

6.3.5. Task JVM Reuse

Jobs can enable task JVMs to be reused by specifying the job configuration `mapred.job.reuse.jvm.num.tasks`. 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 JobConf.setNumTasksToExecutePerJvm(int)

6.3.6. Configured Parameters

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

Name	Type	Description
mapred.job.id	String	The job id
mapred.jar	String	job.jar location in job directory
job.local.dir	String	The job specific shared scratch space
mapred.tip.id	String	The task id
mapred.task.id	String	The task attempt id
mapred.task.is.map	boolean	Is this a map task
mapred.task.partition	int	The id of the task within the job
map.input.file	String	The filename that the map is reading from
map.input.start	long	The offset of the start of the map input split
map.input.length	long	The number of bytes in the map input split
mapred.work.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, mapred.job.id becomes mapred_job_id and mapred.jar becomes mapred_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 at native_libraries.html

6.4. Job Submission and Monitoring

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

`JobClient` 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.

Job history files are also logged to user specified directory `hadoop.job.history.user.location` which defaults to job output directory. The files are stored in "_logs/history/" in the specified directory. Hence, by default they will be in mapred.output.dir/_logs/history. User can stop logging by giving the value `none` for `hadoop.job.history.user.location`

User can view the history logs summary in specified directory using the following command

```
$ bin/hadoop job -history output-dir
```

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 output-dir
```

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 `JobConf`, and then uses the `JobClient` to submit the job and monitor its progress.

6.4.1. Job Authorization

Job level authorization and queue level authorization are enabled on the cluster, if the configuration `mapred.acls.enabled` is set to true. When enabled, access control checks are done by (a) the JobTracker before allowing users to submit jobs to queues and administering these jobs and (b) by the JobTracker and the TaskTracker before allowing users to view job details or to modify a job using MapReduce 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 job ACLs configured, a job's owner, the superuser and cluster administrators (`mapreduce.cluster.administrators`) and queue administrators of the queue to which the job was submitted to (`mapred.queue.queue-name.acl-administer-jobs`) 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 permitted by the queue level ACL, "mapred.queue.queue-name.acl-administer-jobs", configured via mapred-queue-acls.xml. The caller will be able to do the operation if he/she is part of either queue admins ACL or job modification ACL.

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.

6.4.2. 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:

- runJob(JobConf) : Submits the job and returns only after the job has completed.
- submitJob(JobConf) : Only submits the job, then poll the returned handle to the RunningJob to query status and make scheduling decisions.
- JobConf.setJobEndNotificationURI(String) : Sets up a notification upon job-completion, thus avoiding polling.

6.4.3. Job Credentials

In a secure cluster, the user is authenticated via Kerberos' kinit command. Because of scalability concerns, we don't push the client's Kerberos' tickets in MapReduce jobs. Instead, we acquire delegation tokens from each HDFS NameNode that the job will use and store them in the job as part of job submission. The delegation tokens are automatically obtained for the HDFS that holds the staging directories, where the job job files are written, and any HDFS systems referenced by FileInputFormats, FileOutputFormats, DistCp, and the distributed cache. Other applications require to set the configuration "mapreduce.job.hdfs-servers" for all NameNodes that tasks might need to talk during the job execution. This is a comma separated list of file system names, such as "hdfs://nn1/,hdfs://nn2/". These tokens are passed to the JobTracker as part of the job submission as Credentials.

Similar to HDFS delegation tokens, we also have MapReduce delegation tokens. The MapReduce tokens are provided so that tasks can spawn jobs if they wish to. The tasks authenticate to the JobTracker via the MapReduce delegation tokens. The delegation token can be obtained via the API in JobClient.getDelegationToken. The obtained token must then be pushed onto the credentials that is there in the JobConf used for job submission. The API Credentials.addToken can be used for this.

The credentials are sent to the JobTracker as part of the job submission process. The JobTracker persists the tokens and secrets in its filesystem (typically HDFS) in a file within mapred.system.dir/JOBID. The TaskTracker localizes the file as part job localization. Tasks see an environment variable called HADOOP_TOKEN_FILE_LOCATION and the framework sets this to point to the localized file. In order to launch jobs from tasks or for doing any HDFS operation, tasks must set the configuration "mapreduce.job.credentials.binary" to point to this token file.

The HDFS delegation tokens passed to the JobTracker during job submission are are cancelled by the JobTracker when the job completes. This is the default behavior unless mapreduce.job.complete.cancel.delegation.tokens is set to false in the JobConf. For jobs whose tasks in turn spawns jobs, this should be set to false. Applications sharing JobConf objects between multiple jobs on the JobClient side should look at setting mapreduce.job.complete.cancel.delegation.tokens to false. This is because the Credentials object within the JobConf will then be shared. All jobs will end up sharing the same tokens, and hence the tokens should not be canceled when the jobs in the sequence finish.

Apart from the HDFS delegation tokens, arbitrary secrets can also be passed during the job submission for tasks to access other third party services. The APIs JobConf.getCredentials or JobContext.getCredentials() should be used to get the credentials object and then Credentials.addSecretKey should be used to add secrets.

For applications written using the old MapReduce API, the Mapper/Reducer classes need to implement JobConfigurable in order to get access to the credentials in the tasks. A reference to the JobConf passed in the JobConfigurable.configure should be stored. In the new MapReduce API, a similar thing can be done in the Mapper.setup method. The api JobConf.getCredentials() or the api JobContext.getCredentials() should be used to get the credentials reference (depending on whether the new MapReduce API or the old MapReduce API is used). Tasks can access the secrets using the APIs in Credentials

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 `mapred.min.split.size`.

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 `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. 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.

6.6.2. 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 `${mapred.output.dir}/_temporary/_${taskid}` sub-directory accessible via `${mapred.work.output.dir}` for each task-attempt on the `FileSystem` where the output of the task-attempt is stored. On successful completion of the task-attempt, the files in the `${mapred.output.dir}/_temporary/_${taskid}` (only) are *promoted* to `${mapred.output.dir}`. 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 `${mapred.work.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 `${mapred.work.output.dir}` during execution of a particular task-attempt is actually `${mapred.output.dir}/_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.

6.6.3. 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 `mapred.job.queue.name` property, or through the setQueueName(String) API. 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`) and update them via Reporter.incrCounter(Enum, long) or Reporter.incrCounter(String, String, long) in the `map` and/or `reduce` methods. 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 `JobConf`. 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 `mapred.create.symlink` 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 `mapred.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.

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.task.files` to `true` (also see `keep.task.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/${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.

Note that currently IsolationRunner will only re-run map tasks.

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 `mapred.task.profile`. The value can be set using the api JobConf.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 `mapred.task.profile.{maps|reduces}` to set the ranges of MapReduce tasks to profile. The value can be set using the api JobConf.setProfileTaskRange(boolean,String). By default, the specified range is `0-2`.

User can also specify the profiler configuration arguments by setting the configuration property `mapred.task.profile.params`. The value can be specified using the api JobConf.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 `mapred.map.task.debug.script` and `mapred.reduce.task.debug.script`, for debugging map and reduce tasks respectively. These properties can also be set by using APIs JobConf.setMapDebugScript(String) and JobConf.setReduceDebugScript(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. More details on their usage and availability are available here.

6.7.9.1. Intermediate Outputs

Applications can control compression of intermediate map-outputs via the JobConf.setCompressMapOutput(boolean) api and the `CompressionCodec` to be used via the JobConf.setMapOutputCompressorClass(Class) api.

6.7.9.2. Job Outputs

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

Skipped records are written to HDFS in the sequence file format, for later analysis. The location can be changed through SkipBadRecords.setSkipOutputPath(JobConf, 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 needs the HDFS to be up and running, especially for the `DistributedCache`-related features. Hence it only works with a pseudo-distributed or fully-distributed Hadoop installation.

7.1. Source Code

WordCount.java

```
package org.myorg;

import java.io.*;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount extends Configured implements Tool {

  public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text,
IntWritable> {

  static enum Counters { INPUT_WORDS }

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  private boolean caseSensitive = true;
  private Set<String> patternsToSkip = new HashSet<String>();

  private long numRecords = 0;
  private String inputFile;

  public void configure(JobConf job) {
    caseSensitive = job.getBoolean("wordcount.case.sensitive", true);
    inputFile = job.get("map.input.file");

    if (job.getBoolean("wordcount.skip.patterns", false)) {
      Path[] patternsFiles = new Path[0];
      try {
```

```
     patternsFiles = DistributedCache.getLocalCacheFiles(job);
    } catch (IOException ioe) {
     System.err.println("Caught exception while getting cached files: " + StringUtils.stringifyException(ioe));
    }
    for (Path patternsFile : patternsFiles) {
     parseSkipFile(patternsFile);
    }
   }
  }

  private void parseSkipFile(Path patternsFile) {
   try {
    BufferedReader fis = new BufferedReader(new FileReader(patternsFile.toString()));
    String pattern = null;
    while ((pattern = fis.readLine()) != null) {
     patternsToSkip.add(pattern);
    }
   } catch (IOException ioe) {
    System.err.println("Caught exception while parsing the cached file '" + patternsFile + "' : " + StringUtils.stringifyException(ioe));
   }
  }

  public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
   String line = (caseSensitive) ? value.toString() : value.toString().toLowerCase();

   for (String pattern : patternsToSkip) {
    line = line.replaceAll(pattern, "");
   }

   StringTokenizer tokenizer = new StringTokenizer(line);
   while (tokenizer.hasMoreTokens()) {
    word.set(tokenizer.nextToken());
    output.collect(word, one);
    reporter.incrCounter(Counters.INPUT_WORDS, 1);
   }

   if ((++numRecords % 100) == 0) {
    reporter.setStatus("Finished processing " + numRecords + " records " + "from the input file: " + inputFile);
   }
  }
 }

 public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
   int sum = 0;
   while (values.hasNext()) {
    sum += values.next().get();
   }
   output.collect(key, new IntWritable(sum));
  }
 }

 public int run(String[] args) throws Exception {
  JobConf conf = new JobConf(getConf(), WordCount.class);
  conf.setJobName("wordcount");

  conf.setOutputKeyClass(Text.class);
  conf.setOutputValueClass(IntWritable.class);

  conf.setMapperClass(Map.class);
conf.setCombinerClass(Reduce.class);
conf.setReducerClass(Reduce.class);

conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);

  List<String> other_args = new ArrayList<String>();
  for (int i=0; i < args.length; ++i) {
   if ("-skip".equals(args[i])) {
DistributedCache.addCacheFile(new Path(args[++i]).toUri(), conf);
conf.setBoolean("wordcount.skip.patterns", true);
   } else {
    other_args.add(args[i]);
   }
  }

FileInputFormat.setInputPaths(conf, new Path(other_args.get(0)));

```

```
FileOutputFormat.setOutputPath(conf, new Path(other_args.get(1)));

    JobClient.runJob(conf);
    return 0;
  }

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

```

7.2. Sample Runs

Sample text-files as input:

```
$ bin/hadoop dfs -ls /usr/joe/wordcount/input/
/usr/joe/wordcount/input/file01
/usr/joe/wordcount/input/file02
$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file01
Hello World, Bye World!
$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file02
Hello Hadoop, Goodbye to hadoop.
```

Run the application:

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

Output:

```
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-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 dfs -cat /user/joe/wordcount/patterns.txt
\.
\,
\!
to
```

Run it again, this time with more options:

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

As expected, the output:

```
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-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 /usr/joe/wordcount.jar org.myorg.WordCount
-Dwordcount.case.sensitive=false /usr/joe/wordcount/input
/usr/joe/wordcount/output -skip
/user/joe/wordcount/patterns.txt
```

Sure enough, the output:

```
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-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 `configure` method of the `Mapper` (and `Reducer`) implementations (lines 28-43).
- 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 104).
- Demonstrates the utility of the `Tool` interface and the `GenericOptionsParser` to handle generic Hadoop command-line options (lines 87-116, 119).
- Demonstrates how applications can use `Counters` (line 68) and how they can set application-specific status information via the `Reporter` instance passed to the `map` (and `reduce`) method (line 72).

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