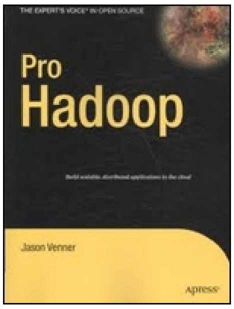

THE EXPERT'S VOICE® IN OPEN SOURCE
Pro
Hadoop
Build scalable, distributed applications in the cloud
Jason Venner
Apress®

Pro Hadoop

by Jason Venner

Apress. (c) 2009. Copying Prohibited.

Chapter 6: Tuning Your MapReduce Jobs

Overview

Once you have developed your MapReduce job, you need to be able to run it at scale on your cluster. A number of factors influence how your job scales. This chapter will cover how to recognize that your job is having a problem and how to tune the scaling parameters so that your job performs optimally.

First, we'll look at tunable items. The framework provides several parameters that let you tune how your job will run on the cluster. Most of these take effect at the job level, but a few work at the cluster level.

With large clusters of machines, it becomes important to have a simple monitoring framework that provides a visual indication of how the cluster is and has been performing. Having alerts delivered when a problem is developing or occurs is also essential. This chapter introduces several tools for monitoring Hadoop services.

Finally, you'll get some tips on what to do when your job isn't performing as it should. Your jobs may be failing or running slowly.

This chapter is focused on tuning jobs running on the cluster, rather than debugging the jobs themselves. Debugging is covered in the next chapter.

Tunable Items for Cluster and Jobs

Hadoop Core is designed for running jobs that have large input data sets and medium to large outputs, running on large sets of dissimilar machines. The framework has been heavily optimized for this use case.

Hadoop Core is optimized for clusters of heterogeneous machines that are not highly reliable. The HDFS file system is optimized for small numbers of very large files that are accessed sequentially. The optimal job is one that uses as input a dataset composed of a number of large input files, where each input file is at least 64MB in size and transforms this data via a MapReduce job into a small number of large files, again where each file is at least 64MB. The data stored in HDFS is generally not considered valuable or irreplaceable. The service level agreement (SLA) for jobs is long and can sustain recovery from machine failure.

Users commonly get into trouble when their jobs input large numbers of small files, output large numbers of small files, or require random access to files. Another problem is a need for rapid access to data or for rapid turnover of jobs.

HDFS installations get into trouble when large numbers of files are being created or exist on the DataNodes.

Hadoop Core does not provide high availability for HDFS or for job submission, and special care must be taken to ensure that required HDFS data can be recovered in the event of a critical failure of the NameNode.

Behind the Scenes: What the Framework Does

Each job has a number of steps in its execution: the setup, the map, the shuffle/sort, and the reduce. The framework sets up, manages, and tears down each step.

> **Note** The following discussion assumes that no other job is running on the cluster and that on submission, the job is immediately started.

On Job Submission

The framework will first store any resources that must be distributed in HDFS. These are the resources provided via the `-files`, `-archives`, and `-libjars` command-line arguments, as well as the JAR file indicated as the job JAR file. This step is executed on the local machine sequentially. If there are a large number of resources, this may take some wall clock time. The XML version of the `JobConf` data is also stored in HDFS.

The replication factor on these resource items is set to the value stored in the configuration under the key `mapred.submit.replication`, with a default value of `10`. The framework will then examine the input data set, using the `InputFormat` class to determine which input files must be passed whole to a task and which input files may be split across multiple tasks.

The framework will use the parameters listed in Table 6-1 to determine how many map tasks must be executed. Input formats may override this; for instance, `NLineInputFormat` forces the splits to be made by line count.

Table 6-1: Parameters Controlling the Number of Map Tasks for a Job

Getter	Parameter	Description	Default
`JobConf.getNumMapTasks()`	`mapred.map.tasks`	The suggested number of map tasks for the job	`1`
No getter	`mapred.min.split.size`	The minimum size of a split	`1`
`FileInputFormat.getMinSplitSize()`		The minimum size to use for this input format (a protected method, currently used only by `SequenceFileInputFormat`)	`Sequence FileInput.SYNC_I` or `1`
`Path.getBlockSize()`	`dfs.block.size`	The file system block size, in bytes of the input file	`67108864`
`InputFormat.isSplitable()`	Not configurable	Whether this file may be split	Varies

The parameters in Table 6-1 are used to compute the actual split size for each input file. The input format for the input file is responsible for indicating if the underlying file may be split. The public method `FileInputFormat.getSplits()` returns the list of splits for the input files. For inputs that can be split, three things are computed before the actual split size is determined: the goal size, which is the total input size divided by `JobConf.getNumMapTask()`; the minimum split size, `Math.max(JobConf.getInt("mapred.min.split.size",1), FileInputFormat.getMinSplitSize());` and the block size for the input file, `Path.getBlockSize()`. The protected method `FileInputFormat.computeSplitSize(goalSize,minSize,blockSize)` is called to produce the actual split size, and the calculation is `Math.max(minSize, Math.min(goalSize, blockSize))`. In summary, splits are determined as follows::

- If a file may not be split, `InputFormat.isSplitable()`, it will be queued as input to one map task.
- A split will be no smaller than the remaining data in the file or `minSize`.
- A split will be no larger than the lesser of the `goalSize` and the `blockSize`.

Tip Through at least Hadoop 0.19.1, compressed files may not be split. A number of patches enable splitting for various compression formats: bzip2 (`http://issues.apache.org/jira/browse/HADOOP-4012`), LZO (`http://issues.apache.org/jira/browse/HADOOP-4640`), and gzip (`http://issues.apache.org/jira/browse/HADOOP-4652`).

In general, a cluster will have the `mapred.map.tasks` parameter set to a value that approximates the number of map task slots available in the cluster or some multiple of that value. The ideal split size is one file system block size, as this allows the framework to attempt to provide data locally for the task that processes the split.

The end result of this process is a set of input splits that are each tagged with information about which machines have local copies of the split data. The splits are sorted in size order so that the largest splits are executed first. The split information and the job configuration information are passed to the JobTracker for execution via a job information file that is written to HDFS.

Some jobs require that the input files not be split. The simplest way to achieve this is to set the value of the configuration parameter `mapred.min.split.size` to `Long.MAX_VALUE:.JobConf.setInt("mapred.min.split.size", Long.MAX_VALUE);`.

Map Task Submission and Execution

The JobTracker has a set of map task execution slots, *N* per machine. Each input split is sent to a task execution slot for execution. Sending tasks to a slot that is hosted on the machine that has a local copy of the input split data minimizes network I/O.

If there are spare execution slots, and map speculative execution is enabled, multiple instances of a map task may be scheduled. In this case, the results of the first map task to complete will be used, the other instances killed, and the output, including the counter values, removed.

When map speculative execution is not enabled, only one instance of a map task will be run at a time. The TaskTracker on the machine will receive the task information, and if necessary, unpack all of the `DistributedCache` data into the task local directory and localize the paths to that data in the `JobConf` object that is being constructed for the task. With speculative execution for map tasks disabled, the only time more than one instance of a map task will occur in the job will be if the task is retried after failing.

> **Caution** The framework is able to back out only counter values and output files written to the task output directory. Any other side effects of killed speculative execution tasks or failed tasks must be handled by the application.

The TaskTracker picks a map runner class based on the content of the key `mapred.map.runner.class`. Its choices are the standard `MapRunner`, which runs a single thread; the `MultithreadedMapRunner`, which runs `mapred.map.multithreadedrunner.threads` (the default is ten threads); or the chain mapper.

A child JVM is allocated to run the mapper class, and the map task is started. The output data of the map task is partitioned and sorted by the output partitioner class and the output comparator class, and aggregated by the combiner class, if one is present. The result of this will be *N* sequence files on disk: one for each reduce task, or one file if there is no reduce task.

Each time the map method is called, an output record is emitted, or the reporter object is interacted with, a heartbeat timer is reset. The heartbeat timeout is stored in the configuration under the key `mapred.tasktracker.expiry.interval`, and has a default value of 600,000 milliseconds (msec), or 10 minutes. If this timeout expires, the map task is considered hung and terminated.

If a terminated task has not failed more than the allowed number of times, it is rescheduled to a different task execution slot. A failing task may have a debugging script invoked on it if the value of the configuration key `mapred.map.task.debug.script` is the path to an executable program. The script is invoked with the additional arguments of the paths to the stdout, stderr, and syslog output files for the task. See this book's appendix, which covers the `JobConf` object, for details on how to configure a debugging script for failing tasks.

When a task finishes, the output commit class is launched on the task output directory, to decide which files are to be discarded and which files are to be committed for the next step. The class name is stored in the configuration under the key `mapred.output.committer.class` and has the default class `FileOutputCommitter`.

If less than the required number of tasks succeed, the job is failed and the intermediate output is deleted. The TaskTracker will inform the JobTracker of the task's success and output locations.

Merge-Sorting

The JobTracker will queue the number of reduce tasks as specified by the `JobConf.setNumReduceTasks()` method and stored in the configuration under the key `mapred.reduce.tasks`. The JobTracker will queue these reduce tasks for execution among the available reduce slots.

The TaskTracker that receives a reduce task will set up the local task execution environment if needed, and then fetch each of the map outputs that are destined for this reduce task. HTTP is the protocol used to transfer the map outputs. These map outputs are merge-sorted. The number of pieces that are fetched at one time is configurable. The value stored in the configuration under the key `mapred.reduce.parallel.copies` determines how many fetches are done in parallel. The default is five fetches.

A number of parameters control how the merge-sorting is done, as shown in Table 6-2.

Table 6-2: Merge-Sort Parameters

Parameter	Description	Default
`io.sort.factor`	The number of map output partitions to merge at a time.	`10`
`io.sort.mb`	The amount of buffer space in megabytes to use when sorting streams. This parameter often causes jobs to run out of memory on small memory machines.	`100`
`io.sort.record.percent`	The amount of the sort buffer dedicated for collecting records. Actual buffer space is this value * `io.sort.mb` / 4.	`0.05`
`io.sort.spill.percent`	The amount of the sort buffer or collection buffer that may be used before the data is spilled to disk.	`0.80`
`io.file.buffer.size`	The buffer size for I/O operations on the disk files.	`4096`
`io.bytes.per.checksum`	The amount of data per checksum.	`512`

`io.skip.checksum.errors`	If `true`, a block with a checksum failure may be skipped.	`false`

The Reduce Phase

Once the data is sorted, the reduce method may be called with the key/value groups. The reduce output is written to the local file system. On successful completion, the output commit class is called to select which output files are staged to the output area in HDFS.

If more than the allowed number of reduce tasks fail, the job is failed. Once the reduce tasks have finished, the job is done.

Writing to HDFS

There are two cases for an HDFS write: the write originates on a machine that hosts a DataNode of the HDFS cluster for which the write is destined, or the write originates on a machine that does not host a DataNode of the cluster. In both cases, the framework buffers a file system block-size worth of data in memory, and when the file is closed or the block fills, an HDFS write is issued.

The write process requests a set of DataNodes that will be used to store the block. If the local host is a DataNode in the file system, the local host will be the first DataNode in the returned set. The set will contain as many DataNodes as the replication factor requires, up to the number of DataNodes in the cluster. The replication factor may be set via the configuration key `dfs.replication`, which defaults to a factor of three, and should never be less than three. The replication for a particular file may be set by the following:

```
FileSystem.setReplication(Path path, int replication);
```

The block is written to the first DataNode in the list, the local host if possible, with the list of DataNodes that are to be used. On receipt of a block, each DataNode is responsible for initiating the transfer of the block to the next DataNode in the list. This allows writes to HDFS on a machine that hosts a DataNode to be very fast for the application, as they do not require bulk network traffic.

Cluster-Level Tunable Parameters

The cluster-level tunable parameters require a cluster restart to take effect. Some of them may require a restart of the HDFS portion of the cluster; others may require a restart of the MapReduce portion of the cluster. These parameters take effect only when the relevant server starts.

Server-Level Parameters

The server-level parameters, shown in Table 6-3, affect basic behavior of the servers. In general, these affect the number of worker threads, which may improve general responsiveness of the servers with an increase in CPU and memory use.

Table 6-3: Server-Level Tunable Parameters

Parameter	Description	Default
`dfs.datanode.handler.count`	The number of threads servicing DataNode block requests	`3`
`dfs.namenode.handler.count`	The number of threads servicing NameNode requests	`10`
`tasktracker.http.threads`	The number of threads for servicing map output files to reduce tasks	`40`
`ipc.server.listen.queue.size`	The number of network incoming connections that may queue for a server	`128`
`nofile`	The limit on the number of file descriptors a process can open (alter `/etc/security/limits.con` for Linux machines)	`1024`

The variables are generally configured by setting the values in the `conf/hadoop-site.xml` file. It is possible to set them via command-line options for the servers, either in the `conf/hadoop-env.sh` file or by setting environment variables (as is done in `conf/hadoop-env.sh`).

The `nofile` parameter is not a Hadoop configuration parameter. It is an operating system parameter. For users of the bash shell, it may be set or examined via the command `ulimit -n [value to set]`. Quite often, the operating system-imposed limit is too low, and the administrator must increase that value. The value `64000` is considered a safe minimum for medium-size busy clusters.

Caution A number of difficult-to-diagnose failures happen when an application or server is unable to allocate additional file descriptors. Java application writers are notorious for not closing I/O channels, resulting in massive consumption of file descriptors by the map and reduce tasks..

Caution Hadoop Core uses large numbers of file descriptors in each server. Rarely is the system default of 1,024 sufficient for the Hadoop servers or Hadoop jobs. Most installations find that a minimum limit of 64,000 is required. If you see errors in your log files that say `Bad connect ack with firstBadLink`, `Could not obtain block`, or `No live nodes contain current block`, you must increase the file descriptor limit for your Hadoop servers and jobs. How to change the limit is covered in Chapter 4, in the "File Descriptors" section.

HDFS Tunable Parameters

The most commonly tuned parameter for HDFS is the file system block size. The default block size is 64MB, specified as `67108864` bytes in `dfs.block.size`. The larger this value, the fewer individual blocks will be stored on the DataNodes, and the larger the input splits will be.

The DataNodes through at least Hadoop 0.19.0 have a limit to the number of blocks that can be stored. This limit appears to be roughly 500,000 blocks. After this size, the DataNode will start to drop in and out of the cluster. If enough DataNodes are having this problem, the HDFS performance will tend toward full stop.

When computing the number of tasks for a job, a task is created per input split, and input splits are created one per block of each input file by default. There is a maximum rate at which the JobTracker can start tasks, at least through Hadoop 0.19.0. The more tasks to execute, the longer it will take the JobTracker to schedule them, and the longer it will take the TaskTrackers to set up and tear down the tasks.

The other reason for increasing the block size is that on modern machines, an I/O-bound task will read 64MB of data in a small number of seconds, resulting in the ratio of task overhead to task runtime being very large. A downside to increasing this value is that it sets the minimum amount of I/O that must be done to access a single record. If your access patterns are not linearly reading large chunks of data from the file, having a large block size will greatly increase the disk and network loading required to service your I/O.

The DataNode and NameNode parameters are presented in Table 6-4.

Table 6-4: HDFS Tunable Parameters

Parameter	Description	Default
`fs.default.name`	The URI of the shared file system. This should be `hdfs://NameNodeHostName:PORT`.	`file:///`
`fs.trash.interval`	The interval between trash checkpoints. If `0`, the trash feature is disabled. The trash is used only for deletions done via the `hadoop dfs -rm` series of commands.	`0`
`dfs.hosts`	The full path to a file containing the list of hostnames that are allowed to connect to the NameNode. If specified, only the hosts in this file are permitted to connect to the NameNode.	
`dfs.hosts.exclude`	A path to a file containing a list of hosts to blacklist from the NameNode. If the file does not exist, no hosts are blacklisted. If a set of DataNode hostnames are added to this file while the NameNode is running, and the command `hadoop dfsadmin -refreshNodes` is executed,the DataNodes listed will be decommissioned. Any blocks stored on them will be redistributed to other nodes on the cluster such that the default replication for the blocks is satisfied. It is best to have this point to an empty file that exists, so that DataNodes may be	

	decommissioned as needed.	
dfs.namenode.decommission.interval	The interval in seconds that the NameNode checks to see if a DataNode decommission has finished.	300
dfs.replication.interval	The period in seconds that the NameNode computes the list of blocks needing replication.	3
dfs.access.time.precision	The precision in msec that access times are maintained. If this value is 0, no access times are maintained. Setting this to 0 may increase performance on busy clusters where the bottleneck is the NameNode edit log write speed.	3600000
dfs.max.objects	The maximum number of files, directories, and blocks permitted.	0
dfs.replication	The number of replicas of each block stored in the cluster. Larger values allow more DataNodes to fail before blocks are unavailable but increase the amount of network I/O required to store data and the disk space requirements. Large values also increase the likelihood that a map task will have a local replica of the input split.	3
dfs.block.size	The basic block size for the file system. This may be too small or too large for your cluster, depending on your job data access patterns.	67108864
dfs.datanode.handler.count	The number of threads handling block requests. Increasing this may increase DataNode throughput, particularly if the DataNode uses multiple separate physical devices for block storage.	3
dfs.replication.considerLoad	Consider the DataNode loading when picking replication locations.	true
dfs.datanode.du.reserved	The amount of space that must be kept free in each location used for block storage.	0.0
dfs.permissions	Permission checking is enabled for file access.	true
dfs.df.interval	The interval between disk usage statistic collection in msec.	60000
dfs.blockreport.intervalMsec	The amount of time between block reports. The block report does a scan of every block that is stored on the DataNode and reports this information to the NameNode. This report as of Hadoop 0.19.0 blocks the DataNode from servicing block reports and is the cause of the congestion collapse of HDFS when more than 500,000 blocks are stored on a DataNode.	3600000
dfs.heartbeat.interval	The heartbeat interval with the NameNode.	3
dfs.namenode.handler.count	The number of server threads for the NameNode. This is commonly greatly increased in busy and large clusters.	10
dfs.name.dir	The location where the NameNode metadata storage is kept. This may be a comma-separated list of directories. A copy will be kept in each location. Writes	${hadoop.tmp.dir}/dfs/name,in /tmp by default

	to the locations are synchronous. If this data is lost, your entire HDFS data set is lost. Keep multiple copies on multiple machines.	
`dfs.name.edits.dir`	The location where metadata edits are synchronously written. This may be a comma-separated list of directories. Ideally, this should hold multiple locations on separate physical devices. If this is lost, your last few minutes of changes will be lost.	`${dfs.Name.dir}`
`dfs.data.dir`	The comma-separated list of directories to use for block storage. This list will be used in a round-robin fashion for storing new data blocks. The locations should be on separate physical devices. Using multiple physical devices yields roughly 50% better performance than RAID 0 striping.	`${hadoop.tmp.dir}/dfs/data`
`dfs.safemode.threshold.pct`	The percentage of blocks that must be minimally replicated before the HDFS will start accepting write requests. This condition is examined only on HDFS startup.	`0.999f`
`dfs.balance.bandwidthPerSec`	The amount of bandwidth that may be used to rebalance block storage among DataNodes.This value is in bytes per second.	`1048576`

JobTracker and TaskTracker Tunable Parameters

The JobTracker is the server that handles the management of the queued and executing jobs. The TaskTrackers are the servers that actually execute the individual map and reduce tasks. Table 6-5 shows the tunable parameters for the JobTracker, and Table 6-6 shows those for TaskTrackers. The JobTracker parameters are global to the cluster. The TaskTracker parameters are for the individual TaskTrackers.

Table 6-5: JobTracker Tunable Parameters

Parameter	Description	Default
`mapred.job.tracker`	The host and port of the JobTracker server. A value of `local` means to run the job in the current JVM with no more than 1 reduce. If the configuration specifies local, no JobTracker server will be started. Per-job configurable.	`local`
`mapred.max.tracker.failures`	The number of task failures allowed on a TaskTracker before the TaskTracker is considered failed for the job with the failing tasks. Per-job configurable.	`4`
`mapred.system.dir`	An HDFS path used for storing job data. If multiple JobTracker servers will share an HDFS cluster, each must have a different `mapred.system.dir`, or the JobTrackers will delete each other's job	`${hadoop.tmp.dir}/mapred`

	files.	
`mapred.temp.dir`	An HDFS path used for storing shared temporary data such as `DistributedCache` data. Per-cluster configurable.	`${hadoop.tmp.dir}/mapred`
`mapred.job.tracker.handler.count`	The number of server threads for handling TaskTracker requests. The recommended value is 4% of the TaskTracker nodes. Per-cluster configurable.	`10`
`mapred.jobtracker.restart.recover`	If this value is `true`, a JobTracker will attempt to restart any queued or running jobs that were running before a crash/shutdown. Per-cluster configurable.	`False`
`mapred.jobtracker.job.history.block.size`	The basic block size used for writes to the history file. Keeping this relatively small ensures that the most data is persisted in the event of a crash. Per-cluster configurable.	`3145728`
`mapred.jobtracker.completeuserjobs.maximum`	The number of jobs to be kept in the JobTracker history. Per-cluster configurable.	`100`
`mapred.jobtracker.maxtasks.per.job`	The maximum number of tasks allowed for a single job. A value of `-1` means no limit. Per-cluster configurable.	`-1`
`mapred.jobtracker.taskScheduler.maxRunningTasksPerJob`	The maximum number of tasks a job can run before it may be preempted. Per-cluster configurable when the Capacity Scheduler services (discussed in Chapter 8) are enabled.	Unlimited
`mapred.job.tracker.persist.jobstatus.active`	Determines whether job status results are persisted to HDFS. Per-cluster configurable.	`false`
`mapred.job.tracker.persist.jobstatus.hours`	The number of hours that job status information is kept. Per cluster.	`0`
`mapred.job.tracker.persist.jobstatus.dir`	The directory where status information is kept. Per-cluster configurable.	`/jobtracker/jobsInfo`
`mapred.hosts`	The full path to a file of hostnames that are permitted to talk to the JobTracker. If specified, only the hosts in this file	

	are permitted.	
mapred.hosts.exclude	The full path to a file of hostnames that are blacklisted from talking to the JobTracker.	

Table 6-6: TaskTracker Tunable Parameters

Parameter	Description	Default
mapred.local.dir	The set of directories to use for task local storage. If multiple directories are provided, the usage is spread over the multiple directories. The directories should be on separate physical devices. Per-TaskTracker configurable.	${hadoop.t
local.cache.size	The local cache directory limit. If more than this many bytes of data are in the task local DistributedCache directory, there will be an attempt to remove unreferenced files. Per-TaskTracker configurable.	1073741824
mapred.local.dir.minspacestart	If the space available in the directories specified by mapred.local.dir falls below this value, do not accept more tasks. This prevents tasks from failing due to lack of temp space. The 0 value should be changed to something reasonable for your jobs. Per-TaskTracker configurable.	0
mapred.local.dir.minspacekill	If the available space in the mapred.local.dir set of directories is below this, accept no more tasks (as if mapred.local.dir.minspace were set to this value) and start killing tasks, starting with reduce tasks, until there is this much space free. Per-TaskTracker configurable.	0
mapred.tasktracker.expiry.interval	The number of msec without a heartbeat that a TaskTracker may go without reporting, before being considered hung and being killed. Per-TaskTracker configurable.	600000
mapred.child.ulimit	Only valid on Unix machines. This is used for processes started by the org.apache.util.hadoop.Shell class. The framework uses this to launch external subprocesses, such as the pipes jobs and the external programs of streaming jobs. Per-TaskTracker configurable.	Unlimited
mapred.tasktracker.taskmemorymanager.monitoring-interval	The rate in msec that virtual memory use by tasks is monitored.	5000
mapred.tasktracker.tasks.maxmemory	The maximum amount of virtual memory a task and its children may use before the TaskTracker will kill the task. A value of t indicates no limit. Per-TaskTracker configurable.	-1
mapred.tasktracker.procfsbasedprocesstree.sleeptime-before-sigkill	A task over its memory limit is sent a SIGTERM. If the task has not exited within this time in msec, a SIGKILL is sent.	5000
mapred.map.tasks.maximum	The number of map tasks to run simultaneously on a TaskTracker. This should Either be 1 (if there is only one CPU) or roughly one less than the number of CPUs on the machine. This parameter needs to be tuned for a particular job mix. Per-TaskTracker configurable.	2
mapred.reduce.tasks.maximum	The number of simultaneous reduce tasks to run. This value is really a function of the CPU	2

	and I/O bandwidth available to the machine. It needs to be tuned for the machines and job mix. Per-TaskTracker configurable.	
`mapred.tasktracker.dns.interface`	For multihomed TaskTracker nodes, report this interface's IP address to the JobTracker. If not `default`, this value is the name of a network interface, such as `eth0`. Per-TaskTracker configurable.	`default`
`mapred.tasktracker.dns.nameserver`	For multihomed TaskTracker nodes, use this address for DNS hostname resolution when resolving the IP address of the network interface specified by `mapred.tasktracker.dns.interface`. The value `default` means use the system default.	`default`
`tasktracker.http.threads`	The number of threads serving HTTP requests for reduce tasks requesting map output. If your system has many reduce execution slots, the default may be too small.	`40`
`mapred.userlog.limit.kb`	The maximum amount of data that may be written to a task user log.	`0`
`mapred.userlog.retain.hours`	The number of hours that user logs are retained.	`24`

Per-Job Tunable Parameters

The framework provides rich control over the way individual jobs are executed on the cluster. You can tune file system and task-related parameters. Table 6-7 shows the tunable parameters for the file system.

Table 6-7: File System Tunable Parameters

Parameter	Description	Default
`fs.default.name`	This is the URI for the shared file system. Normally it will be set to `hdfs://NamenodeHostname:NameNodePort`.	`file:///`
`dfs.replication`	The job may configure this value.	`3`
`dfs.block.size`	The client may also configure this value.	`67108864`
`dfs.client.block.write.retries`	The number of write attempts before a write is considered failed. In general, if writes are being retried, there is a problem with the HDFS or machine configuration.	`3`

The task-tunable parameters directly control the behavior of tasks in the cluster. These are the heart of the MapReduce framework. A large number of parameters affect the job. Only those parameters that directly control core functions are listed in Table 6-8. Many of the parameters are detailed in this book's appendix, which discusses the `JobConf` object.

Table 6-8: Core Job-Level Task Parameters

Parameter	Description	Default
`mapred.map.tasks`	The suggested number of map tasks for a job.	2
`mapred.reduce.tasks`	The number of reduce tasks for the job.	1
`mapred.map.max.attempts`	The maximum number times a map task will be retried after an error, before it is considered failed.	4
`mapred.reduce.max.attempts`	The maximum number of times a reduce task will be retried after an error, before it is considered failed.	4

`mapred.reduce.parallel.copies`	The number of parallel fetches of map output data made via HTTP at a time.	`5`
`mapred.reduce.copy.backoff`	The maximum amount of time to try to fetch a map output partition, before abandoning that partition.	`300`
`mapred.task.timeout`	The amount of time in msec that a task may go without the map or reduce method finishing, or making a call on the reporter or output collector.	`600000` (10 min)
`mapred.child.java.opts`	The options to use for initializing the task JVM. `@taskid@` is replaced with the current task ID.	`-Xmx200m`
`mapred.child.tmp`	The value passed to the JVM for `java.io. tmpdir`. If it is a relative path, it will be relative to the task's local working directory.	`/tmp`
`mapred.map.tasks.speculative.execution`	Whether idle map task slots will be used to set up execution races for executing identical map tasks. This will consume more cluster resources and may offer faster job throughput. This must be `false` if your map tasks have side effects that the framework cannot undo or have real costs.	`true`
`mapred.reduce.tasks.speculative.execution`	Enable the use of unused reduce task execution slots to try a task in multiple slots, to see if one slot may complete the task faster. This will consume more cluster resources and may offer faster job throughput. This *must* be `false` if your reduce tasks have side effects the framework cannot undo or have real costs.	`true`
`mapred.job.reuse.jvm.num.tasks`	The number of times a task JVM may be reused for additional tasks of the same type for the same job. A value of `-1` indicates no limit.	`1`
`mapred.submit.replication`	The replication factor for per-job data. This needs to be tuned on a per-job basis.	`10`
`keep.failed.task.files`	Whether the local directories for failed tasks should be kept. This is for debugging. There is no automatic mechanism in the framework to clean these directories if this is set to `false`.	`false`
`keep.task.files.pattern`	If set, a `java.util.Pattern` will be applied to task names to determine if their local directories will be kept. This is	Unset

	normally not present.	
`mapred.output.compress`	Use compression on the final output data files for the job. This is usually a significant win for jobs with large output.	`false`
`mapred.output.compression.type`	The type of compression to do for the job output files if they are `SequenceFiles`. `BLOCK` is generally considered better if random access to the output is not desired.	`RECORD`
`mapred.output.compression.codec`	The codec to use for compression.	`org.apache.hadoop.io.compress.De`
`mapred.compress.map.output`	If `true`, use compression on the map output that is destined for a reduce task. This is usually a significant win.	`false`
`mapred.map.output.compression.codec`	The codec to use for intermediate map output files. The `LzoCodec` appears to be the current best choice if it is available.	`org.apache.hadoop.io.compress.De` `Codec`
`io.seqfile.compress.blocksize`	The minimum block size to use for block-level compression of `SequenceFiles`.	`1000000`
`io.seqfile.lazydecompress`	Only decompress `SequenceFile` data when it is needed.	`true`
`io.seqfile.sorter.recordlimit`	The maximum number of records to attempt to keep in memory when sorting the records of a `SequenceFile`.	`1000000`
`map.sort.class`	The sort implementation to use when sorting keys using the `OutputComparator`.	`org.apache.hadoop.util.Quick Sor`
`jobclient.output.filter`	The status of the tasks whose user log data is reported to the console of the `JobClient` that submitted the job. The values allowed are `NONE`, `KILLED`, `FAILED`, `SUCCEEDED`, and `ALL`.	`FAILED`
`mapred.task.profile`	If `true`, some tasks may be profiled.	`false`
`mapred.task.profile.maps`	The set of map tasks to profile. See this book's appendix for how this may be set.	`0-2`
`mapred.task.profile.reduces`	The set of reduce tasks to profile. See this book's appendix for how this may be set.	`0-2`
`mapred.skip.attempts.to.start.skipping`	The number of failures of a task before skip mode is engaged. This is covered in Chapter 8.	`2`
`mapred.skip.map.auto.incr.proc.count`	Automatically increment the counter `ReduceProcessedGroups`. This must be `false` for streaming jobs or jobs that	`true`

	buffer records before reducing.	
mapred.skip.out.dir	If unset, skipped records are written to `file_logs/skip` in the output directory. If the value is exactly none, no records will be written. If set to anything else, it becomes the directory where skipped records are written.	Unset
mapred.skip.map.max.skip.records	The number of contiguous records, including the bad record that may be skipped. The framework will attempt to narrow down the region to skip to this size. If the value is `0`, no skipping is allowed. If the value is `Long.MAX_VALUE`, the entire split will be skipped.	0
mapred.skip.reduce.max.skip.groups	The number of key/value set groups surrounding a bad record group that may be skipped by the reduce task. See Chapter 8 for details.	0

Monitoring Hadoop Core Services

To be able to detect incipient failures, or otherwise recognize that a problem is developing or has occurred, some mechanism must be available to monitor current status, and if possible provide historical status. The Hadoop framework provides several APIs for allowing external agents to provide monitoring services to the Hadoop Core services. Here, we will look at Java Management Extensions (JMX), Nagios, Ganglia, Chukwa, and FailMon.

JMX: Hadoop Core Server and Task State Monitor

Hadoop provides local JMX bean services for all services. This allows for the use of JMX-aware applications to collect information about the state of the servers. The default configuration provides for only local access to the managed beans (MBeans). To enable remote access, after determining a port for JMX use, alter the `conf/hadoop-env.sh` file (shown in Listing 6-1) and change the JMX properties being set on the servers.

Listing 6-1: The Default hadoop-env.sh Settings for Hadoop Servers to Enable JMX

```
export HADOOP_NAMENODE_OPTS="-Dcom.sun.management.jmxremote $HADOOP_NAMENODE_OPTS"
export HADOOP_SECONDARYNAMENODE_OPTS="-Dcom.sun.management.jmxremote ➥
$HADOOP_SECONDARYNAMENODE_OPTS"
export HADOOP_DATANODE_OPTS="-Dcom.sun.management.jmxremote $HADOOP_DATANODE_OPTS"
export HADOOP_BALANCER_OPTS="-Dcom.sun.management.jmxremote $HADOOP_BALANCER_OPTS"
export HADOOP_JOBTRACKER_OPTS="-Dcom.sun.management.jmxremote ➥
$HADOOP_JOBTRACKER_OPTS"
# export HADOOP_TASKTRACKER_OPTS=
```

The string `-Dcom.sun.management.jmxremote` enables the JMX management bean services in the servers. The string is a JVM argument and passed to the JVM at start time on the command line.

JMX supports several connection options. See the Sun-supplied documentation for configuring access control and remote access, at `http://java.sun.com/javase/6/docs/technotes/guides/jmx/ index.html`.

Nagios: A Monitoring and Alert Generation Framework

Nagios (`http://www.nagios.org`) provides a flexible customizable framework for collecting data about the state of a complex system and triggering various levels of alerts based on the collected data. A service of this type is essential for your cluster administration and operations team.

The University of Nebraska has a web page (`http://t2.unl.edu/documentation/hadoop/monitoring-`

guide/) that details how to use the Nagios check_jmx plug-in to monitor Hadoop servers. The information is reproduced here. This example assumes that you understand how to construct the JMX password file and access control file.

To enable JMX monitoring on Hadoop, add the following lines to hadoop-env.sh:

```
export HADOOP_NAMENODE_OPTS=" -Dcom.sun.management.jmxremote.authenticate=false ➥
    -Dcom.sun.management.jmxremote.ssl=false ➥
    -Dcom.sun.management.jmxremote.port=8004 ➥
    -Dcom.sun.management.jmxremote.password.file= ➥
    $HADOOP_HOME/conf/jmxremote.password ➥
    -Dcom.sun.management.jmxremote.access.file=$HADOOP_HOME/conf/jmxremote.access"
export HADOOP_DATANODE_OPTS=" -Dcom.sun.management.jmxremote.authenticate=false ➥
    -Dcom.sun.management.jmxremote.ssl=false ➥
    -Dcom.sun.management.jmxremote.port=8004 ➥
    -Dcom.sun.management.jmxremote.password.file= ➥
    $HADOOP_HOME/conf/jmxremote.password ➥
    -Dcom.sun.management.jmxremote.access.file=$HADOOP_HOME/conf/jmxremote.access"
```

The following lines add check_jmx to the Nagios deployment:

```
./check_jmx -U service:jmx:rmi:///jndi/rmi://node182:8004/jmxrmi ➥
     -O hadoop.dfs:service=DataNode,name=DataNodeStatistics ➥
     -A BlockReportsMaxTime -w 10 -c 150
./check_jmx -U service:jmx:rmi:///jndi/rmi://node182:8004/jmxrmi ➥
    -O java.lang:type=Memory -A HeapMemoryUsage -K used -C 10000000
```

Ganglia: A Visual Monitoring Tool with History

Hadoop has built-in support for Ganglia version 3.0 through Hadoop 0.19.0. Support for Ganglia 3.1 is expected for Hadoop 0.20. The Ganglia framework is available from http://ganglia.sourceforge.net.

Ganglia by itself is a highly scalable cluster monitoring tool, and provides visual information on the state of individual machines in a cluster or summary information for a cluster or sets of clusters. Ganglia provides the ability to view different time windows into the past, normally one hour, one day, one week, one month, and so on.

> **Caution** Due to some limitations in the Ganglia support in Hadoop through at least Hadoop 0.19.1, the configuration requirements are not as simple as Ganglia configuration normally is.

Ganglia is composed of two servers: the gmetad server, which provides historical data and collects current data, and the gmond server, which collects and serves current statistics. The Ganglia web interface is generally installed on the host(s) running the gmetad servers, and in coordination with the host's httpd provides a graphical view of the cluster information. In general, each node will run gmond, but only one or a small number of nodes will also run gmetad.

For Hadoop reporting to work with Ganglia, the configuration changes shown in Table 6-9 must be made in the conf/hadoop-metrics.properties file. Each Hadoop cluster must be allocated a unique multicast address/port, and be considered a single reporting domain. Each cluster must also be allocated a unique cluster name. The cluster name is referred to as CLUSTER in this section. The UDP port for reporting is referred to as PORT, and for simplicity, the multicast port will be identical to PORT.

Table 6-9: Required Parameters for Hadoop Ganglia Reporting Configuration

Substitution String	Description
CLUSTER	The unique cluster name shared by all hosts within the cluster/reporting domain.
HOSTNAME	The hostname of the machine that will be the Ganglia reporting master for CLUSTER
PORT	The non-multicast UDP port that the gmond server on HOSTNAME will listen on. Also the multicast port unique to CLUSTER, which all gmond servers in CLUSTER will listen and transmit on.
MULTICAST	The multicast address that the gmond servers in the cluster will communicate over. The default of 239.2.11.71 is acceptable as long as each CLUSTER uses a unique PORT.

> **Note** MULTICAST:PORT must be unique per CLUSTER, but generally MULTICAST is left at the default value, so PORT becomes the unique value per cluster. The gmond on HOSTNAME will need to be configured to listen on the non-multicast UDP port of PORT. Many enterprise-grade switches will need to have multicast enabled for each

CLUSTER's MULTICAST:PORT.

All nodes in the cluster will have the gmond server configured with the cluster name parameter set with the cluster's unique name. One node in the cluster, traditionally the NameNode or a JobTracker node, is configured to also accept non-multicast reporting on a port, commonly the same port as the multicast reception port. This host will be considered the Ganglia cluster master, and its hostname is the value for HOSTNAME. This host is also the host used in the /etc/gmetad.conf file. The conf/hadoop-metrics file needs to be altered as shown in Listing 6-2. The HOSTNAME and PORT must be substituted for the actual values. This file must then be distributed to all of the Hadoop conf directories and all Hadoop servers restarted.

Listing 6-2: The conf/hadoop-metrics.properties File for Ganglia Reporting

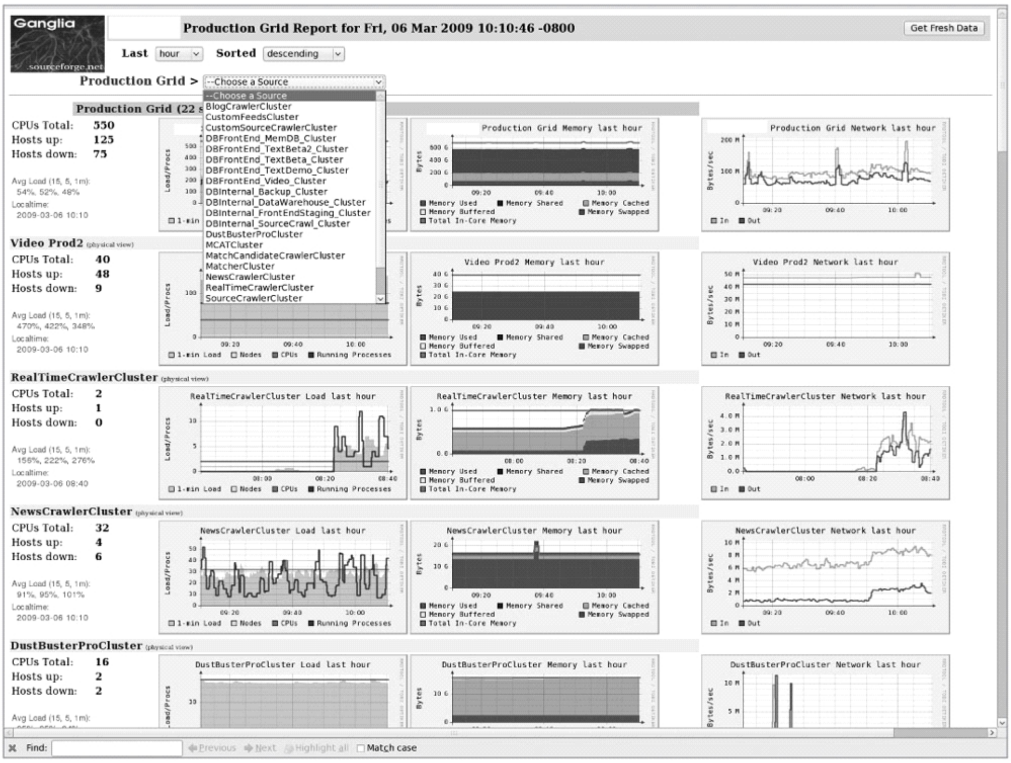
```
# Configuration of the "dfs" context for ganglia
dfs.class=org.apache.hadoop.metrics.ganglia.GangliaContext
dfs.period=10
dfs.servers=HOSTNAME:PORT

# Configuration of the "mapred" context for ganglia
mapred.class=org.apache.hadoop.metrics.ganglia.GangliaContext
mapred.period=10
mapred.servers=HOSTNAME:PORT


# Configuration of the "jvm" context for ganglia
jvm.class=org.apache.hadoop.metrics.ganglia.GangliaContext
jvm.period=10
jvm.servers=HOSTNAME:PORT
```

All of the Hadoop servers will now deliver metric data to HOSTNAME:PORT via UDP, once every 10 seconds.

The gmetad server that will collect metric information for the cluster will need to be instructed to collect metric information about CLUSTER from the master node via a TCP connection to HOSTNAME:PORT. The following is the configuration line in the gmetad.conf file for CLUSTER:

```
data_source "CLUSTER" HOSTNAME:PORT
```

The Ganglia web interface will provide a graphical view of the clusters, as shown in Figure 6-1.



Ganglia
sourceforge.net
Get Fresh Data
Last hour
Sorted descending
Production Grid > --Choose a Source
--Choose a Source
BlogCrawlerCluster
CustomFeedsCluster
CustomSourceCrawlerCluster
DBFrontEnd_MemDB_Cluster
DBFrontEnd_TextBeta2_Cluster
DBFrontEnd_TextBeta_Cluster
DBFrontEnd_TextDemo_Cluster
DBFrontEnd_Video_Cluster
DBInternal_Backup_Cluster
DBInternal_DataWarehouse_Cluster
DBInternal_FrontEndStaging_Cluster
DBInternal_SourceCrawl_Cluster
DustBusterProCluster
MCATCluster
MatchCandidateCrawlerCluster
MatcherCluster
NewsCrawlerCluster
RealTimeCrawlerCluster
SourceCrawlerCluster
Production Grid Memory last hour
Production Grid Network last hour
Video Prod2 Memory last hour
Video Prod2 Network last hour
RealTimeCrawlerCluster Load last hour
RealTimeCrawlerCluster Memory last hour
RealTimeCrawlerCluster Network last hour
NewsCrawlerCluster Load last hour
NewsCrawlerCluster Memory last hour
NewsCrawlerCluster Network last hour
DustBusterProCluster Load last hour
DustBusterProCluster Memory last hour
DustBusterProCluster Network last hour
Load/Procs
Bytes
Bytes/sec
1-min Load
Nodes
CPUs
Running Processes
Memory Used
Memory Shared
Memory Cached
Memory Buffered
Memory Swapped
Total In-Core Memory
In
Out
Find:
Previous
Next
Highlight all
Match case

Production Grid Report for Fri, 06 Mar 2009 10:10:46 -0800

Production Grid (22 s

CPUs Total: 550
Hosts up: 125
Hosts down: 75

Avg Load (15, 5, 1m):
54%, 52%, 48%
Localtime:
2009-03-06 10:10

Video Prod2 (physical view)

CPUs Total: 40
Hosts up: 48
Hosts down: 9

Avg Load (15, 5, 1m):
470%, 422%, 348%
Localtime:
2009-03-06 10:10

RealTimeCrawlerCluster (physical view)

CPUs Total: 2
Hosts up: 1
Hosts down: 0

Avg Load (15, 5, 1m):
156%, 222%, 276%
Localtime:
2009-03-06 08:40

NewsCrawlerCluster (physical view)

CPUs Total: 32
Hosts up: 4
Hosts down: 6

Avg Load (15, 5, 1m):
91%, 95%, 101%
Localtime:
2009-03-06 10:10

DustBusterProCluster (physical view)

CPUs Total: 16
Hosts up: 2
Hosts down: 2

Avg Load (15, 5, 1m):

Figure 6-1: The Ganglia web view of a running set of clusters

When tuning jobs, Ganglia provides a wonderful interface to determine when your job is fully utilizing a cluster resource. Determining which resource is fully utilized, tuning the appropriate configuration parameters for that resource, and then rerunning the job will allow you to optimize your job's runtime on your cluster.

Chukwa: A Monitoring Service

Chukwa's goal is to provide extract, transform, and load (ETL) services for cluster logging data, thereby providing end users with a simple and efficient way to find the logging events that are actually important. Chukwa is new in Hadoop 0.19.0 and evolving rapidly.

Chukwa uses HDFS to collect data from various data providers, and MapReduce to analyze the collected data. The instance in Hadoop 0.19.0 appears to be currently optimized for the collection of data from log files, and then run a scheduled MapReduce job over the collected data. The Chukwa Quick Start is hosted on the Hadoop wiki, at `http://wiki.apache.org/hadoop/Chukwa_Quick_Start`.

FailMon: A Hardware Diagnostic Tool

The FailMon framework attempts to identify failures on large clusters by analyzing data collected from the Hadoop logs, the system logs, and other sources. The FailMon tools stem from a larger IBM effort to improve the operational reliability of large installations by predicting failures and taking corrective action before the failure occurs (see (https://issues.apache.org/jira/secure/attachment`/12386597/failmon.pdf`). This is a very early technology and is expected to evolve rapidly.

The FailMon package consists primarily of data collection tools with MapReduce jobs to perform analysis of the collected data.

Tuning to Improve Job Performance

The general goal for tuning is for your jobs to finish as rapidly as possible using no more resources than necessary. This section covers best practices for achieving optimum performance of jobs.

Speeding Up the Job and Task Start

If the job requires many resources to be copied into HDFS for distribution via the distributed cache, or has large datasets that need to be written to HDFS prior to job start, substantial wall clock time can be spent copying in the files. For constant resources, it is simplest and quickest to make them available on all of the cluster machines and adjust the TaskTracker classpaths to reflect these resource locations.

The disadvantage of installing the resources on all of the machines is that it increases administrative complexity, as well as the possibility that the required resources are unavailable or an incorrect version. The advantage of this approach is that it reduces the amount of work the framework must do when setting up each task and may decrease the overall job runtime.

Table 6-10 provides a checklist of items to look for that affect write performance and what to do when the situations occur.

Table 6-10: What to Monitor for Initial Bulk Transfer of Input Data

Resource	What to Look For	What to Do
Source machine CPU utilization	The CPU is maxed out, the compression level is too high, or the compression algorithm is computationally too expensive	Change the compression or change the number of threads.
Source machine network	Saturation of the outbound network connection with traffic for HDFS	Increase the number of transfer threads or provide a higher-speed network connection.
Per DataNode network input	If it is not saturated, more writes could be delivered to this DataNode	Increase the number of simultaneous threads writing or reduce the number of files being created by increasing the individual file sizes.
DataNode I/O wait	I/O contention on a DataNode	Add more independent locations to `dfs.data.dir` or add more DataNodes.

If you have a large number of files that need to be stored in HDFS prior to the task start, such as might occur if your job needs to populate the job input directory, there are several things you may try, in varying combinations:

- It may be faster to copy the files from a machine that hosts a DataNode, as all of the writes will first go to the local DataNode, and the application will not have to wait for the data to traverse the network. The downside is that one replica of every block will end up on the local DataNode, greatly reducing the opportunity for data to be local to a map task. The DataNode may also get unbalanced with respect to storage, compared to other DataNodes. Ideally, bulk input of data to be used as input to a map task should be input from a host that does not also provide DataNode services, to ensure even distribution of the stored blocks across the DataNodes.
- It may be faster to run the copies in parallel. The limiting factor will be the network speed or the local DataNode disk speed in the event the copy host is also a DataNode.
- Use compression for data to be used once. LZO provides very good compression at little CPU overhead, provided that a native implementation is available.
- Create an archive of the input files, so that fewer files need to be created in HDFS. The downside is that zip and tar archives must be processed whole by a map task and may not be split into pieces (at least through Hadoop 0.19.0). Writing compressed sequence files, where the key/value pairs are of the type `BytesWritable`, will give you input that may be split and a reduction in file size.
- If you have large volumes of data, you may need to set up special machines with high-bandwidth network connections to the switching fabric that holds your DataNodes. Each block being written is sent directly to a DataNode. That DataNode will in turn send the block to the next DataNode in the chain and so on, until the required number of replicas are complete.
- If the origination machine has a higher bandwidth connection and is able to write multiple blocks in parallel (via multiple open files) while the bandwidth to each DataNode will be capped by the DataNode network speed, the origination machine will be able to write to HDFS at a higher rate.

There are very few tunable parameters at this point. You may change the `dfs.block.size` parameter to issue larger or smaller writes. You may decrease the `dfs.replication` parameter to reduce the overall HDFS write load, or increase it to increase the chance of local access by later MapReduce jobs. Compression generally helps but may cause issues later. Figure 6-2 illustrates how HDFS operations that your application issues are actually handled by the framework. Implicit in Figure 6-2 is that the replication count is three.

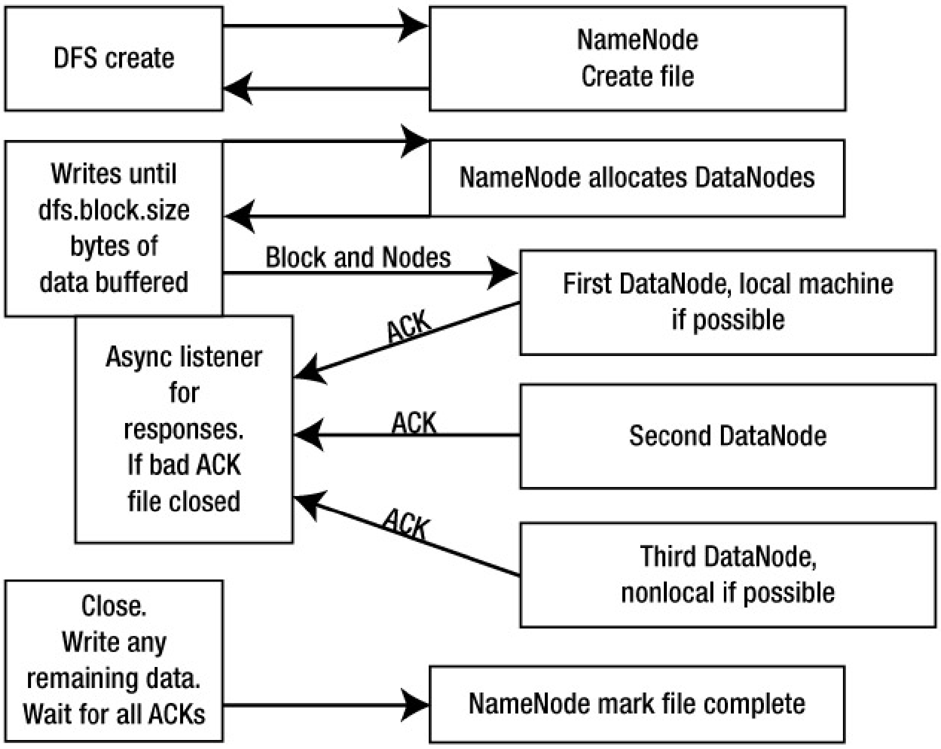

DFS create
NameNode Create file
Writes until dfs.block.size bytes of data buffered
NameNode allocates DataNodes
Block and Nodes
First DataNode, local machine if possible
ACK
Async listener for responses. If bad ACK file closed
ACK
Second DataNode
ACK
Third DataNode, nonlocal if possible
Close. Write any remaining data. Wait for all ACKs
NameNode mark file complete

Figure 6-2: Writing a block of data to HDFS

From a monitoring perspective, you will want to monitor the network utilization on the upload machine and to a lesser extent on the DataNodes. If you are using compression, you will want to monitor CPU utilization on the machines doing the compression.

You may also wish to monitor the disk-write rate on the DataNodes, to verify that you are getting a good write rate. Since the incoming data rate is capped by the network rate, generally this is not a significant factor. If you see pauses in the network traffic or disk I/O, it implies that a Hadoop server may be unresponsive, and the client is timing out and will retry.

In general, increasing the server threads (`dfs.datanode.handler.count`) and the TCP listen queue depth (`ipc.server.listen.queue.size`) may help. It may be that the NameNode is not keeping up with requests, and in that case, increasing `dfs.namenode.handler.count` may help.

Optimizing a Job's Map Phase

The map phase involves dispatching a map task to a TaskTracker. Once the TaskTracker has a task to execute, it will prepare the task execution environment by building or refreshing the `DistributedCache` data. The TaskTracker maintains information about the `DistributedCache` for a particular job, and multiple tasks from the same job will share the same local execution environment. If you don't have an existing child JVM that has been used for this job's task and is within its reuse limit, start a new child JVM. The TaskTracker will then trigger the start of the map task in the child JVM.

The child JVM will start reading key/value pairs from its input, executing the map method for each pair. The output key/value pairs will be partitioned as needed and collected in the proper output format. If there is a reduce phase, the output format will be on the local disk in a sequence file. If there is not a reduce phase, the output will be in the job-specified output format and stored in HDFS.

Figure 6-3 shows a diagram of the job setup and map task execution. The left side follows the actions of the JobTracker from job submission through executing the map tasks on the available TaskTrackers. The right side follows the loop that a TaskTracker executes for map tasks. The diagram is read from top to bottom. The `Tasktracker$Child` is the class providing a `main()` method for the actual map task, which will be executed in a JVM launched and managed by the TaskTracker.

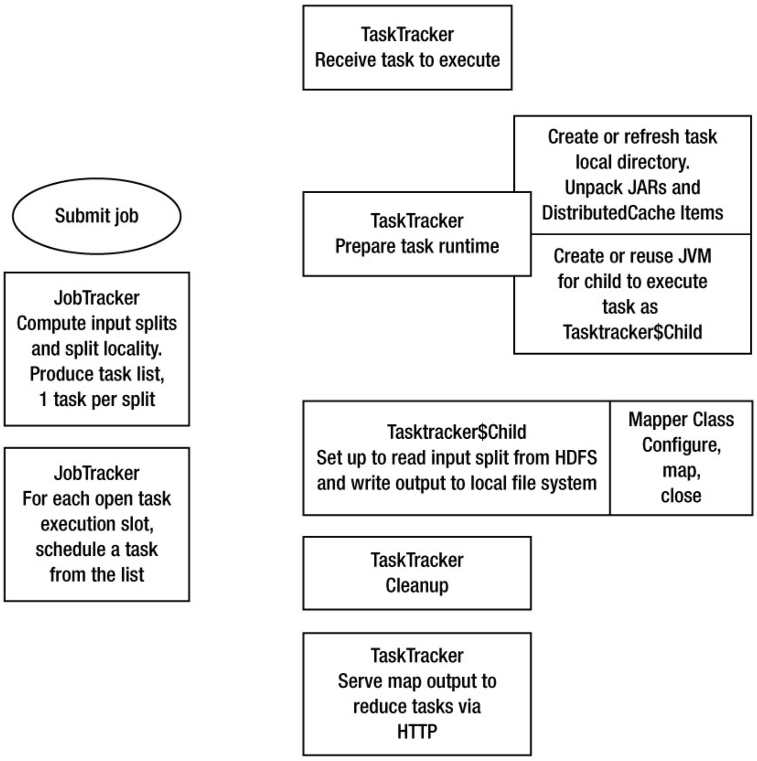

TaskTracker
Receive task to execute
Create or refresh task local directory. Unpack JARs and DistributedCache Items
Submit job
TaskTracker
Prepare task runtime
Create or reuse JVM for child to execute task as Tasktracker$Child
JobTracker
Compute input splits and split locality. Produce task list, 1 task per split
Tasktracker$Child
Set up to read input split from HDFS and write output to local file system
Mapper Class
Configure, map, close
JobTracker
For each open task execution slot, schedule a task from the list
TaskTracker
Cleanup
TaskTracker
Serve map output to reduce tasks via HTTP

Figure 6-3: Behind the scenes in a map task

The following are some items you can tune for the map phase:

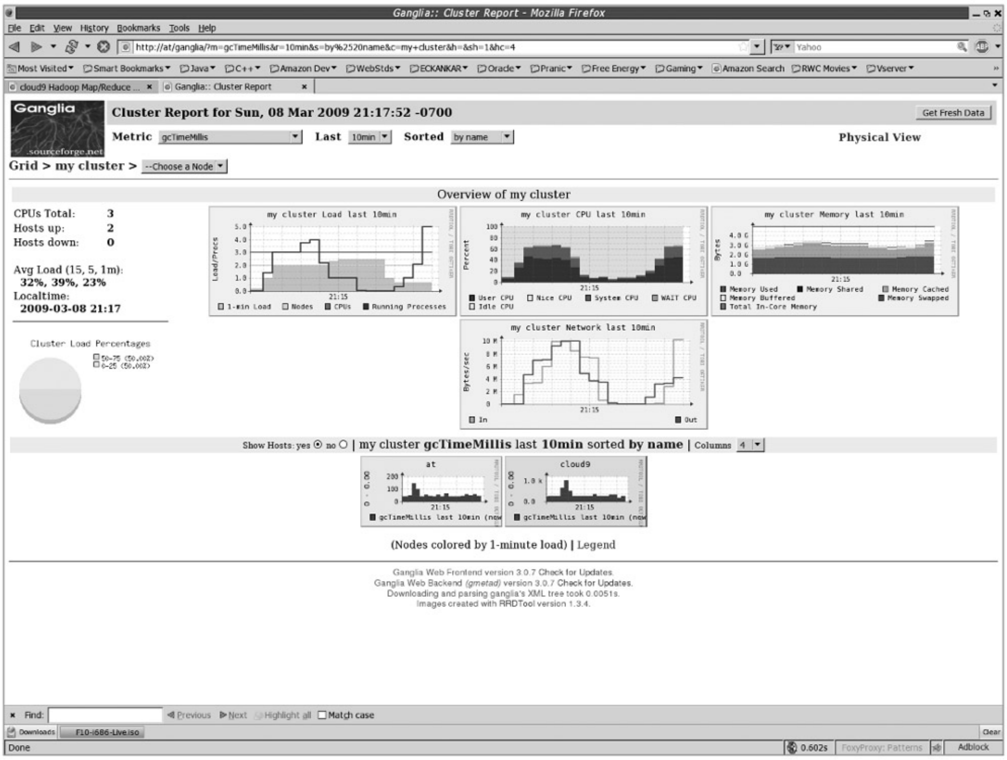
Map task run time: Each map task has some setup and teardown overhead. If the runtime of the map task is short, this overhead becomes the major time component. If the runtimes of tasks are too long, a single task may hold the cluster for a long period, or retrying a failed task becomes expensive. In general, less than a minute is usually too short, What is too long for a map task is job-specific. The primary tuning point for this is the `dfs.block.size` parameter. Increasing this parameter usually increases the split size and the task run time. On a per-job or per-cluster basis, you may also change `mapred.min.split.size`. It is better to use `dfs.block.size`, as the data is more likely to be local when the split size equals the HDFS file system block size.

TaskTracker node CPU utilization: If the map tasks are computationally intensive, a significant goal is to use all of the available CPU resources for that computation. There are two methods for controlling CPU utilization for map tasks:

- The job or cluster may configure the use of `MultithreadedMapRunner` for the `MapRunner` via `mapred.map.runner.class`, and specify the number of execution treads via `mapred.map.multithreadedrunner.threads`.
- The cluster may specify the number of map tasks to run simultaneously by a Task-Tracker via `mapred.tasktracker.map.tasks.maximum`. This may be done on the command line for any job that uses the `GenericOptionsParser`.

Data location: If the map tasks are not receiving their input split from a local DataNode, the I/O performance will be limited to the network speed. This value is visible in the job counters of running and completed jobs, under the section titled "Data Local map tasks," the Total column gives the number of map tasks that ran with the input split served from a local DataNode. Other than increasing the replication factor and trying to ensure that the input split size is the file system block size, there is little tuning to be done.

Child garbage collection: If there is significant object churn in the `Mapper.map` method, and there is insufficient heap space allocated, the JVM hosting the task may spend a significant amount of wall clock time doing garbage collection. This is readily visible only via the Ganglia reporting framework or through the JMX MBean interface. The Ganglia reporting variable is `gcTimeMillis()` and is visible in the main reporting page for Ganglia, as shown in Figure 6-4.



Ganglia:: Cluster Report - Mozilla Firefox

http://at/ganglia/?m=gcTimeMillis&r=10min&s=by%2520name&c=my+cluster&h=&sh=1&hc=4

Cluster Report for Sun, 08 Mar 2009 21:17:52 -0700

Metric gcTimeMillis Last 10min Sorted by name

Grid > my cluster > --Choose a Node

Overview of my cluster

CPUs Total: 3
Hosts up: 2
Hosts down: 0

Avg Load (15, 5, 1m):
32%, 39%, 23%
Localtime:
2009-03-08 21:17

Show Hosts: yes ● no ○ | my cluster **gcTimeMillis** last **10min** sorted **by name** | Columns 4

(Nodes colored by 1-minute load) | Legend

Ganglia Web Frontend version 3.0.7 Check for Updates.
Ganglia Web Backend *(gmetad)* version 3.0.7 Check for Updates.
Downloading and parsing ganglia's XML tree took 0.0051s.
Images created with RRDTool version 1.3.4.

Done

Figure 6-4: Ganglia report showing gcTime for a two-host cluster, where one host is in trouble

Figure 6-4 shows an example of a Ganglia report for a two-host cluster, where one host is having problems. Note in the bottom-right graph, showing `gcTimeMilis`, how the host `cloud9` is spending roughly 2 to 400 msec per sample period doing garbage collection. This would imply that the child JVM has been configured with insufficient memory. At the current time, it is not possible to differentiate the garbage collection timing for the different server processes.

In this case, it's possible that increasing the child JVM memory limit, via `mapred.child.java.opts`, would be helpful. In this 10-minute window, the same task was run twice. The second time, it was run with twice as much memory per child JVM via `mapred.child.java.opts`. Note how much less time was taken in garbage collection on the right side of the graph for `cloud9` versus the left half of the graph.

Here are the command-line options to enable multithreaded map running with ten threads:

```
-D mapred.map.runner.class=org.apache.hadoop.mapred.lib.MultithreadedMapRunner➥
    -D mapred.tasktracker.map.tasks.maximum=10
```

Tuning the Reduce Task Setup

The reduce task requires the same type of setup as the map task does with respect to the `DistributedCache` and the child JVM working environment. The two key differences relate to the input and the output. The reduce task input must be fetched from each of the TaskTrackers on which a map task has run, and these individual datasets need to be sorted. The reduce output is written to HDFS, unlike with the map task, which has output to the local file system.

As you can see from Figure 6-5, there are several steps for a reduce task, each of which has different constraints, as follows:

- The JobTracker can launch only so many tasks per second; this is something that will change after Hadoop 0.19.1.
- The tuning parameters for the map task with respect to job setup apply equally to the reduce task.
- The framework must fetch all of the map outputs for the reduce task, from the TaskTrackers that have them.
- The data to be fetched may be large enough that the network transfer speed becomes a bounding issue.

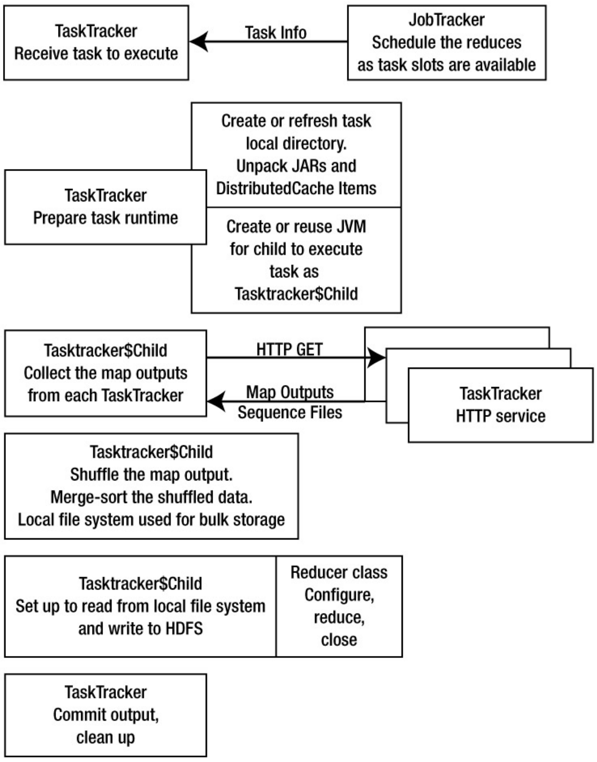


Figure 6-5: Behind the scenes in the reduce task

The following parameters affect the reduce task:

- `mapred.reduce.parallel.copies`: Controls how many fetches are run in parallel for each reduce task.
- `tasktracker.http.threads`: Controls the number of threads each TaskTracker runs to service these map output requests.
- `ipc.server.listen.queue.size`: At a lower layer, controls the number of requests that can queue before the client gets a connection-refused message.

There are several ways to reduce the size of the map output files, which can greatly speed up this phase. Some care needs to be used, as some of the compression options may slow down the shuffle and sort phase. The simplest thing is to specify a combiner class, which will act as a mini-reduce phase in each map task (as described in Chapter 5). This works very well for aggregation jobs, and not so well for jobs that need the full value space in the reduce task on which to operate. Many sites will enable map output file compression, via the Boolean value `mapred.compress.map.output`, in the `hadoop-site.mxl` file.

The choice of the compression algorithm is less clear. Native LZO is usually a good choice. The final trade-off is record-level versus block-level compression. The default is stored in `io.seqfile.compression.type`, and is `RECORD`. Conceptually, `RECORD` is better for the map output, as there will be a fair bit of reading through the files during the shuffle and sort phases. This is something that will have to be tried on a per-job basis. The other issue is that, at least through Hadoop 0.19.0, there is only one setting for this parameter, which affects all `SequenceFiles`.

> **Note** There are a number of parameters that control the shuffle and merge. Tuning these parameters is expert work. There is a short discussion of the parameters in the Hadoop documentation, in the "Shuffle/Reduce Parameters" section (`http://hadoop.apache.org/core/docs/current/mapred_tutorial.html#Shuffle/Reduce+Parameters`).

Choosing the number of reduce tasks to run per machine and per cluster is the final level of tuning. A major determinant here is how the output data will be used, and that is application-specific. With reduce tasks, I/O, rather than CPU usage, is usually the bottleneck. If the DataNodes are coresident with the TaskTrackers, the reduce tasks will always have a local DataNode for the output. This will allow the initial writes to go at local speed, but the file closes will block until all the replicas are complete.

It is not uncommon for jobs to open many files in the reduce phase, which generally causes a huge slowdown, if not failure, in the HDFS cluster, so the job will take a significant amount of time to finish.

The following are some tuning points for the reduce phase:

Shuffle/sort time: The shuffle and sort cannot complete until all of the map output data is available. If this is an issue, you can try the following:

- Use a combiner class.
- Increase the number of `tasktracker.http.threads`.
- Increase the `ipc.server.listen.queue.size`.
- Set `mapred.compress.map.output` to `true`.
- Vary the compression codec stored in `mapred.map.output.compression.codec`.
- Experiment with `io.seqfile.compression.type` as `RECORD` or `BLOCK`.
- Change your algorithm so that less data needs to pass to the reduce phase. Try more reduce tasks, to reduce the volume of data that each reduce phase must sort.

Network saturation: The pull of the map outputs should just saturate your network. If the reduce tasks are timing out while trying to fetch outputs, increase the `tasktracker.http.threads`. If the network is saturated, enable compression, reduce the number of map tasks, improve the combiner class, or restructure the job to reduce the data passed to the reduce phase.

> **Note** I once had a job where part of the value associated with each key was a large block of XML data that was unused by the reduce phase. Modifying the map to drop the XML data provided a tenfold improvement.

Actual reduce time: You may find that the time to actually reduce the data, after the shuffle and sort are done, is too long. If you are using `MultipleOutputFormat`, ensure that the number of files being created is small. If many small files must be created, write them as a zip archive. The Ganglia `gmetric` value `FilesCreated` will give you an idea of the rate of HDFS file creation.

Write time: The write time may be too long if the volume of data or the number of files are large. Enable output compression via setting `mapred.output.compress` to `true`. Experiment with codecs. Pack multiple files into zip files or other archive formats.

> **Note** I had one job that needed to create many tens of thousands of small files. Writing the files as a zip archive in HDFS resulted in a hundredfold speed increase.

Overall reduce phase time: If the reduce phase is very long, you may want to tailor the number of reduce tasks per job and per machine. The job may specify the number of reduce tasks to run, but at least through Hadoop 0.19.1, the number of reduce tasks per TaskTracker is fixed at start time. If your cluster will run a specific set of jobs, with experimentation, you may find a reasonable number for the cluster-level parameter, and given that, identify a specific value for the number of reduce tasks for each job.

Addressing Job-Level Issues

One of the more interesting things to see is a cluster going almost idle while a job is running. This usually happens because a small number of tasks have not finished. This situation is called the *job tail*. This can happen with the map tasks or the reduce tasks. With multithreaded map tasks, the key or keys passed to one thread can sometimes take much longer to finish, and the task needs to wait for one or a small number of threads to complete, leaving the machine mostly idle. This is called the *task tail*.

Dealing with the Task Tail

I've had substantial experience with clusters set up with a single map task per TaskTracker, and have set the number of threads used by the `MultithreadedMapRunner` class to tune the task for full CPU utilization (roughly 80% to 90%). In one particular job, there was a large variance in the time it took to process a key: some keys took three hours, and others three seconds. If the long-running keys came late in an input split, the task would end up running one thread and idle six of the processors on the machine. The only solution for this was to reorder the input keys so that the long-running keys came first in the splits, or to abandon the long-running keys after a set elapsed run time, and reprocess all of the long-running keys in an additional job later.

Dealing with the Job Tail

The Hadoop standard is for very large jobs, spread over many machines, such that the time of one or two tasks is small compared to the run time of the job. This, in part, is where the 10-minute timeouts for server failures come from—a 10-minute period is considered short in the time of a job, so why not wait for that long? Many organizations have short timelines for jobs and limited budgets for hardware. These organizations must tune their jobs so that the clusters are well utilized.

The job tail really comes down to either a small number of reduce tasks taking much longer than others, either because the partitioning of the key space is very uneven or the duplicate keys fall unevenly in the partitions. The net result is that some reduce tasks have substantially more work to do. This is readily addressed only by turning the partitioning, via a custom partitioner class set via the `JobConf.setPartitionerClass(Class<? extends Partitioner>theClass)` method.

Tuning the number of reduce tasks so they fall evenly on your reduce slots may also help. Having one reduce task start after all the rest of the reduce tasks have finished can drastically increase the job runtime.

Summary

This chapter detailed how jobs are run by the Hadoop Framework and how MapReduce application writers and cluster administrators can tune both jobs and clusters for optimal performance.

The NameNode, JobTracker, DataNodes, and TaskTrackers have a number of start time parameters that directly affect how jobs are executed by the cluster and the overall run time of the jobs. The execution of a job is performed in several steps: setup, map, shuffle/sort, and reduce. It's possible to do some tuning to improve performance in each step.

This chapter discussed several tools for monitoring clusters and jobs. Ganglia is the tool I prefer for tuning and general dashboard-level awareness, and Nagios is the one I use for operational support. Using these tools enables rapid recognition of problems with jobs and clusters and provides insight into what parameters may need to be tuned, before the CEO calls you into the front office to explain why the mission-critical jobs haven't been running successfully. Ganglia also provides the informative, pretty graphs that higher-level management like so much.