A Guide for Developers and Administrators

Hadoop Operations

Free Sampler

O'REILLY®

Eric Sammer

Hadoop Operations
by Eric Sammer

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Courtney Nash
Production Editor: Melanie Yarbrough
Copyeditor: Audrey Doyle
Proofreader:

Indexer:
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

September 2012: First Edition.

Revision History for the First Edition:
2012-09-21 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449327057* for release details.

ISBN: 978-1-449-32705-7

[]

1345146532

Table of Contents

CHAPTER 1
Introduction

Over the last few years, there has been a fundamental shift in data storage, management, and processing. Companies are storing more data from more sources in more formats than ever before. This isn't just about being a "data pack-rat," but instead building products, features, and intelligence predicated on knowing more about the world (where the world can be users, searches, machine logs or whatever is relevant to an organization). Organizations are finding new ways to use data that was previously believed to be of little value, or far too expensive to retain, to better serve their constituents. Sourcing and storing data is one half of the equation. Processing that data to produce *information* is fundamental to the daily operations of every modern business.

Data storage and processing isn't a new problem, though. Fraud detection in commerce and finance, anomaly detection in operational systems, demographic analysis in advertising, and many other applications have had to deal with these issues for decades. What has happened is that the volume, velocity, and variety of this data has changed, and in some cases, rather dramatically. This makes sense, as many algorithms benefit from access to more data. Take, for instance, the problem of recommending products to a visitor of an e-commerce website. You could simply show each visitor a rotating list of products they could buy, hoping that one would appeal to them. It's not exactly an informed decision, but it's a start. The question is what do you need to improve the chance of showing the right person the right product? Maybe it makes sense to show them what you think they like, based on what they've previous looked at. For some products, it's useful to know what they already own. Customers who already bought a specific brand of laptop computer from you may be interested in compatible accessories and upgrades[1]. One of the most common techniques is to cluster users by similar behavior (e.g. purchase patterns) and recommend products purchased by "similar" users. No matter the solution, all of the algorithms behind these options require data,

1. I once worked on a data driven marketing project for a company that sold beauty products. Using purchase transactions of all customers over a long period of time, the company was able to predict when a customer would run out of a given product after purchasing it. As it turned out, simply offering them the same thing about a week before they ran out resulted in a (very) noticeable lift in sales.

and generally improve in quality with more of it. Knowing more about a problem space generally leads to better decisions (or algorithm efficacy), which in turn leads to happier users, more money, reduced fraud, healthier people, safer conditions, or whatever the desired result might be.

Apache Hadoop is a platform that provides pragmatic, cost effective, scalable infrastructure for building many of the types of applications described earlier. Made up of a distributed filesystem called the Hadoop Distributed Filesystem, or HDFS, and a computation layer that implements a processing paradigm called MapReduce, Hadoop is an open source, batch data processing system for enormous amounts of data. We live in a flawed world, and Hadoop is designed to survive in it by not only tolerating hardware and software failures, but treating it as a first class condition that happens regularly. Hadoop uses a cluster of plain old commodity servers with no specialized hardware or network infrastructure to form a single, logical, storage and compute platform, or *cluster*, that can be shared by multiple individuals or groups. Computation in Hadoop MapReduce is performed in parallel, automatically, with a simple abstraction for developers that obviates complex synchronization and network programming. Unlike many other distributed data processing systems, Hadoop runs the user provided processing logic on the machine where the data lives rather than dragging the data across the network; a huge win for performance.

For those interested in the history, Hadoop was modeled after two papers produced by Google, one of the many companies to have these kinds of data intensive processing problems. The first, presented in 2003, described a pragmatic, scalable, distributed filesystem optimized for storing enormous datasets called The Google Filesystem, or *GFS*. In addition to simple storage, GFS was built to support large scale, data intensive, distributed processing applications. The following year, another paper titled MapReduce: Simplified Data Processing on Large Clusters was presented, and defined a programming model and accompanying framework that provided automatic parallelization, fault tolerance, and the scale to process hundreds of terabytes of data in a single job, over thousands of machines. When paired, these two systems could be used to build large data processing clusters on relatively inexpensive, commodity machines. These papers directly inspired the development of HDFS and Hadoop MapReduce, respectively.

Interest and investment in Hadoop has led to an entire ecosystem of related software both open source and commercial. Within the Apache Software Foundation alone, projects that explicitly make use of, or integrate with, Hadoop are springing up regularly. Some of these projects make authoring MapReduce jobs easier and more accessible, others focus on getting data in and out of HDFS, simplify operations, enable deployment in cloud environments, and so on. A sampling of the more popular projects with which you should familiarize yourself include:

Apache Hive - http://hive.apache.org
: Hive creates a relational database style abstraction that allows developers to write a dialect of SQL which, in turn, is executed as one or more MapReduce jobs on

the cluster. Developers, analysts, and existing third party packages already know and speak SQL (Hive's dialect of SQL is called HiveQL and implements only a subset of any of the common standards). Hive takes advantage of this and provides a quick way to reduce the learning curve to adopting Hadoop and writing MapReduce jobs. For this reason, Hive is, by far, one of the most popular Hadoop ecosystem projects.

Hive works by defining a table-like schema over an *existing* set of files in HDFS and handling the gory details of extracting records from those files when a query is run. The data on disk is never actually changed, just parsed at query time. HiveQL statements are interpreted and an execution plan of pre-built map and reduce classes are assembled to perform the MapReduce equivalent of the SQL statement.

Apache Pig - http://pig.apache.org

Like Hive, Apache Pig was created to simplify the authoring of MapReduce jobs, obviating the need to write Java code. Instead, users write data processing jobs in a high level scripting language from which Pig builds an execution plan, and executes a series of MapReduce jobs to do the heavy lifting. In cases where Pig doesn't support a necessary function, developers can extend its set of built in operations by writing user defined functions in Java (Hive supports similar functionality as well). If you know Perl, Python, Ruby, Javascript, or even shell script, you can learn Pig's syntax in a morning and be running MapReduce jobs by lunch time.

Apache Sqoop - http://sqoop.apache.org

Not only does Hadoop not want to replace your database, it wants to be friends with it. Exchanging data with relational databases is one of the most popular integration points with Apache Hadoop. Sqoop, short for "SQL to Hadoop," performs bidirectional data transfer between Hadoop and (almost) any database with a JDBC driver. Using MapReduce, Sqoop performs these operations in parallel with no need to write code.

For even greater performance, Sqoop supports database specific plugins that use native features of the RDBMS, rather than incurring the overhead of JDBC. Many of these connectors are open source, while others are free or available from commercial vendors at a cost. Today, Sqoop includes native connectors (called *direct* support) for MySQL and PostgreSQL. Free connectors exist for Teradata, Netezza, SQL Server, and Oracle (from Quest Software), and are available for download from their respective company web sites.

Apache Flume - http://flume.apache.org

Apache Flume is a streaming data collection and aggregation system designed to transport massive volumes of data into systems like Hadoop. It supports native connectivity and support for writing directly to HDFS, and simplifies reliable, streaming, data delivery from a variety of sources including RPC services, log4j appenders, syslog, and even the output from OS commands. Data can be routed, load balanced, replicated to multiple destinations, and aggregated from thousands of hosts by a tier of agents.

Apache Oozie - http://incubator.apache.org/oozie/

It's not uncommon for large production clusters to run many coordinated MapReduce jobs in a workfow. Apache Oozie is a workflow engine and scheduler built specifically for large scale job orchestration on a Hadoop cluster. Workflows can be triggered by time or events such as data arriving in a directory, and job failure handling logic can be implemented so that policies are adhered to. Oozie presents a REST service for programmatic management of workflows and status retrieval.

Apache Whirr - http://whirr.apache.org

Apache Whirr was created to simplify the creation and deployment of ephemeral clusters in cloud environments like Amazon's AWS. Run as a command line tool either locally or within the cloud, Whirr can spin up instances, deploy Hadoop, configure the software, and tear it down on demand. Under the hood, Whirr uses the powerful jclouds library so that it is cloud provider neutral. The developers have put in the work to make Whirr support both Amazon EC2 and Rackspace Cloud. In addition to Hadoop, Whirr understands how to provision Apache Cassandra, Apache ZooKeeper, Apache HBase, ElasticSearch, Voldemort, and Apache Hama.

Apache HBase - http://hbase.apache.org

Apache HBase is a low latency, distributed, (non-relational) database built on top of HDFS. Modeled after Google's BigTable, HBase presents a flexible data model with scale out properties, and a very simple API. Data in HBase is stored in a semi-columnar format, partitioned by rows, into *regions*. It's not uncommon for a single table in HBase to be well into the hundreds of terabytes or, in some cases, petabytes. Over the last few years, HBase has gained a massive following based on some very public deployments like Facebook's Messages platform. Today, HBase is used to serve huge amounts of data to realtime systems in major production deployments.

Apache ZooKeeper - http://zookeeper.apache.org

A true workhorse, Apache ZooKeeper is a distributed, consensus-based, coordination system used to support distributed applications. Distributed applications that require leader election, locking, group membership, service location, and configuration services can use ZooKeeper rather than reimplement the complex coordination and error handling that comes with these functions. In fact, many projects within the Hadoop ecosystem use ZooKeeper for exactly this purpose, most notably, HBase.

Apache HCatalog - http://incubator.apache.org/hcatalog/

A relatively new entry, HCatalog is a service that provides shared schema and data access abstraction services to applications with the ecosystem. The long term goal of HCatalog is to enable interoperability between tools like Apache Hive and Pig so they may share dataset metadata information.

The Hadoop ecosystem is exploding into the commercial world as well. Vendors like Oracle, SAS, MicroStrategy, Tableau, Informatica, Microsoft, Pentaho, Talend, HP, Dell, and dozens of others have all developed integration or support for Hadoop within

one or more of their products. Hadoop is fast becoming (or, as an increasingly growing group would believe, already has become) the defacto standard for truly large scale data processing in the data center.

If you're reading this book, you may be a developer with some exposure to Hadoop looking to learn more about managing the system in a production environment. Alternatively, it could be that you're an application or system administrator tasked with owning the current or planned production cluster. Those in the latter camp may be rolling their eyes at the prospect of dealing with yet another system. That's fair, and we won't spend a ton of time talking about writing applications, APIs, and other pesky code problems. There are other fantastic books on those topics, especially Hadoop: The Definitive Guide by Tom White (O'Reilly). Administrators do, however, play an absolutely critical role in planning, installing, configuring, maintaining, and monitoring Hadoop clusters. Hadoop is a comparatively low level system, leaning heavily on the host operating system for many features, and works best when developers and administrators collaborate regularly. What you do impacts how things work.

It's an extremely exciting time to get into Apache Hadoop. The so called *big data* space is all the rage, sure, but more importantly, Hadoop is growing and changing at a staggering rate. Each new version - and there have been a few big ones in the last year or two - brings another truck load of features for both developers and administrators alike. You could say that Hadoop is experiencing software puberty; with the rapid growth and adoption, it's also a little awkward at times. You'll find, throughout this book, that there are significant changes between even minor versions. It's a lot to keep up with, admittedly, but don't let it overwhelm you. Where necessary, the differences are called out, and a section in Chapter 4 is devoted to walking you through the most commonly encountered versions.

This book is intended to be a pragmatic guide to running Hadoop in production. Those who have some familiarity with Hadoop may already know alternative methods for installation, or have differing thoughts on how to properly tune the number of map slots based on CPU utilization[2]. That's expected, and more than fine. The goal is not to enumerate all possible scenarios, but rather call out what works, as demonstrated in critical deployments.

Chapters two and three provide the necessary background, and describe what HDFS and MapReduce are, why they exist, and, at a high level, how they work. Chapter four walks you through the process of planning for a Hadoop deployment including hardware selection, basic resource planning, operating system selection and configuration, Hadoop distribution and version selection, and network concerns for Hadoop clusters. If you are looking for the meat and potatoes, Chapter five is where it's at, with configuration and setup information, including a listing of the most critical properties, organized by topic. Those that have strong security requirements or want to understand

2. We also briefly cover the flux capacitor and discuss the burn rate of energon cubes during combat.

identity, access, and authorization within Hadoop will want to pay particular attention to chapter six. Chapter seven explains the nuts and bolts of sharing a single large cluster across multiple groups and why this is beneficial, while still adhering to service level agreements by managing and allocating resources accordingly. Once everything is up and running, chapter eight acts as a run book for the most common operations and tasks. Chapter nine is the rainy day chapter, covering the theory and practice of troubleshooting complex distributed systems like Hadoop, including some real world war stories. In an attempt to minimize those rainy days, chapter ten is all about how to effectively monitor your Hadoop cluster. Finally, chapter eleven provides some basic tools and techniques for backing up Hadoop and dealing with catastrophic failure.

CHAPTER 10
Monitoring

An Overview

It's hard to talk about building large, shared, mission critical systems without having a way to know their operational state and performance metrics. Most organizations (I hope) have some form of monitoring system that keeps track of various systems that occupy the data center. No one runs a large Hadoop cluster by itself, and while a lot of time is spent on data integration, monitoring integration sometimes falls by the wayside.

Most monitoring systems can be divided into two major components: metric collection, and consumption of the resultant data. Hadoop is simply another source from which metrics should be collected. Consumption of the data can mean presenting aggregate metrics as dashboards, raw metrics as time series data for diagnoses and analysis, and, very commonly, rule evaluation for alerting. In fact, many monitoring systems provide more than one of these features. It helps to further divide monitoring into two distinct types: health monitoring, where the goal is to determine that a service is in an expected operational state, and performance monitoring, where the goal is to use regular samples of performance metrics, over time, to gain a better understanding of how the system functions. Performance monitoring tends to be harder to cover outside of the context of a specific environment and set of workloads, and so instead we'll focus primarily on health monitoring.

Hadoop, like most distributed systems, provides for a monitoring challenge because the monitoring system must know about how the multiple services interact as a larger system. When monitoring HDFS, for example, we may want to see each daemon running, within normal memory consumption limits, responding to RPC requests in a defined window, and other "simple" metrics, but this doesn't tell us if the entirety of the service is functional (although one may infer such things). Instead, it can be necessary to know that a certain percentage of datanodes are alive and communicating with the namenode, or what the block distribution is across the cluster to truly know the state of the system. Zooming in too close on a single daemon or host, or too far out

on the cluster, can both be equally deceptive when trying to get a complete picture of performance or health of a service like HDFS.

Worse still, we want to know how MapReduce is performing on top of HDFS. Alert thresholds and performance data of MapReduce is inherently coupled to that of HDFS when services are stacked in this manner, making it difficult to detect the root cause of a failure across service and host boundaries. Existing tools are very good at identifying problems within localized systems (those with little to no external dependency) but require quite a bit of effort to understand the intricacies of distributed systems. These complexities make defining the difference between a daemon being up and responding to basic checks, and it being safe to walk away from a computer (or go to sleep) difficult to ascertain.

In the following sections, we'll cover what Hadoop metrics are, how to configure them, and some common ways to integrate with existing monitoring services and custom code.

Hadoop Metrics

Hadoop has built-in support for exposing various metrics to outside systems. Each daemon can be configured to collect this data from its internal components at a regular interval and then handle the metrics in some way using a plugin. A number of these plugins ship with Hadoop, ready for use in common deployment scenarios (more on that later). Related metrics are grouped into a named *context*, and each context can be treated independently. Some contexts are common to all daemons like the information about the JVM and RPC operations performed, while others only apply to daemons of a specific service like HDFS metrics that come from only the namenode and datanodes. In most cases, administrators find that all contexts are useful and necessary when monitoring for health, understanding performance, and diagnosing problems with a cluster.

The metrics system has evolved over time, gaining features and improvements along the way. In April of 2010 a project[1] was started to refactor the metrics subsystem to support features that had been too difficult to provide under the existing implementation. Notably, the new metrics subsystem (referred to as *metrics2*) supports sending metrics to multiple plugins, filtering of metrics in various ways, and more complete support for JMX. Work was completed in mid to late 2011 and included in Apache Hadoop 0.20.205 and CDH4 and later. Earlier versions of Apache Hadoop and CDH use the original metrics implementation. For the sake of clarity, we'll refer to the old metrics system as *metrics1* when discussing the specific implementation and use the more general term *metrics* to refer to the functionality of either version.

1. See Apache Hadoop JIRA HADOOP-6728.

Apache Hadoop 0.20.0 and CDH3 (metrics1)

The original implementation of the metrics system groups related metrics into contexts, as described earlier. Each context can be individually configured with a plugin that specifies how metric data should be handled. A number of plugins exist including the default `NullContext` which simply discards all metrics received. Internal components of the daemon are polled at a regular, user-defined interval, with the resulting data being handled by the configured plugin. The primary four contexts are:

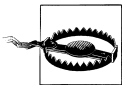
jvm
: Contains Java virtual machine information and metrics. Example data includes the maximum heap size, occupied heap, and average time spent in garbage collection. All daemons produce metrics for this context.

dfs
: Contains HDFS metrics. The metrics provided vary by daemon role. For example, the namenode provides information about total HDFS capacity, consumed capacity, missing and under-replicated blocks, and active datanodes in the cluster, while datanodes provide the number of failed disk volumes and remaining capacity on that particular worker node. Only the HDFS daemons output metrics for this context.

mapred
: Contains MapReduce metrics. The metrics provided vary by daemon role. For example, the jobtracker provides information about the total number of map and reduce slots, blacklisted tasktrackers, and failures where as tasktrackers provide counts of running, failed, and killed tasks at the worker node level. Only MapReduce daemons output metrics for this context.

rpc
: Contains remote procedure call metrics. Example data includes the time each RPC spends in the queue before being processed, the average time it takes to process an RPC, and the number of open connections. All daemons output metrics for this context.

While all of Hadoop is instrumented to capture this information, it's not available to external systems by default. One must configure a plugin for each context to handle this data in some way. The metrics system configuration is specified by the *hadoop-metrics.properties* file within the standard Hadoop configuration directory. The configuration file is a simple Java properties format file and largely self explanatory, although examples follow below. The following metric plugins are included with Hadoop.



Metric plugin class names tend to reuse the word *context* but to mean something other than the context described above. When we refer to *context*, we mean one of the four groups of metrics. We'll use the term *metric plugin* to refer to the class that can be configured to handle metrics.

`org.apache.hadoop.metrics.spi.NullContext`
: Hadoop's default metric plugin for all four contexts, `NullContext` is the */dev/null* of plugins. Metrics are not collected from the internal components nor are they output to any external system. This plugin effectively disables access to metrics, shown in Example 10-1.

Example 10-1. Using NullContext to disable metric collection

```
# hadoop-metrics.properties

jvm.class = org.apache.hadoop.metrics.spi.NullContext

dfs.class = org.apache.hadoop.metrics.spi.NullContext

...
```

`org.apache.hadoop.metrics.spi.NoEmitMetricsContext`
: The `NoEmitMetricsContext` is a slight variation on `NullContext` with an important difference. While metrics are still not output to an external system, the thread that runs within Hadoop, updating the metric values in memory *does* run. Systems like JMX and the metrics servlet (described later) rely on this information being collected and available to function[2]. There's almost no reason not to enable this plugin for all contexts, for all clusters. The added insight into the system is significant and well worth the minimal overhead introduced. Additionally, updating the metric configuration requires a daemon restart which can temporarily disrupt service down the road. The only parameter `NoEmitMetricsContext` supports is `period`, which defines the frequency with which collection occurs. See Example 10-2 for an example of a configuration using `NoEmitMetricsContext`.

Example 10-2. Using NoEmitMetricsContext for metrics collection

```
# hadoop-metrics.properties

jvm.class = org.apache.hadoop.metrics.spi.NoEmitMetricsContext
jvm.period = 10

dfs.class = org.apache.hadoop.metrics.spi.NoEmitMetricsContext
dfs.period = 10

...
```

2. More specifically, JMX will always be able to access the standard JVM instrumented components, but this is different than the `jvm` context provided by Hadoop metrics.

`org.apache.hadoop.metrics.file.FileContext`

`FileContext` polls the internal components of Hadoop for metrics periodically and writes them out to a file on the local filesystem. Two parameters are available (see Example 10-3) to the control the file name to write to (`fileName`) and the desired frequency of updates (`period`) in seconds.

Example 10-3. Using FileContext for metrics collection

```
# hadoop-metrics.properties

jvm.class = org.apache.hadoop.metrics.file.FileContext
jvm.period = 10
jvm.fileName = /tmp/jvm-metrics.log

dfs.class = org.apache.hadoop.metrics.file.FileContext
dfs.period = 10
dfs.fileName = /tmp/dfs-metrics.log

...
```

Practically speaking, `FileContext` is flawed and shouldn't be used in production clusters. This is because the plugin never rotates the specified file, leading to indefinite growth. Truncating the file doesn't work because the JVM will continue to write to the open file descriptor until the daemon is restarted. During that period of time, disk space on the local filesystem will continue to evaporate and it won't be immediately clear as to why (the classic anonymous file situation). The other major problem is that a unique file name must be given for each context and, when multiple daemons live on the same machine (e.g. worker nodes), additionally across daemons. This is a pain to manage and instead, you're encouraged to consider using `NoEmitMetricsContext` and the metrics servlet, or Hadoop's JMX support.

`org.apache.hadoop.metrics.ganglia.GangliaContext` *and* `org.apache.hadoop.metrics.ganglia.GangliaContext31`

Hadoop includes first class support for integrating with the popular open source performance monitoring system Ganglia. Ganglia was specifically built by a group at the University of California, Berkeley, to collect, aggregate, and plot a large number of metrics from large clusters of machines. Because of its ability to scale, Ganglia is a fantastic system to use with Hadoop. It works by running a small monitoring daemon on each host called *gmond* which collects metrics locally. Each *gmond* process relays data to a central *gmetad* process that records data in a series of RRD, or *round-robin database* files, which are fixed-size files that efficiently store time series data. A PHP web application displays this data in a simple, but effective view.

The `GangliaContext` is normally configured, as in Example 10-4, to send metrics to the local *gmond* process using the `servers` property in *hadoop-metrics.properties*.

Like `FileContext` and `NoEmitMetricsContext`, a `period` parameter specifies how frequently data is collected.

Example 10-4. Using GangliaContext for metrics collection

```
# hadoop-metrics.properties

jvm.class = org.apache.hadoop.metrics.file.FileContext
jvm.period = 10
jvm.servers = 10.0.0.191
# The server value may be a comma separated list of host:port pairs.
# The port is optional, in which case it defaults to 8649.
# jvm.servers = gmond-host-a, gmond-host-b:8649

dfs.class = org.apache.hadoop.metrics.file.FileContext
dfs.period = 10
dfs.servers = 10.0.0.191

...
```

The difference between `GangliaContext` and `GangliaContext31` is that the former works with Ganglia 3.0 and older while the latter supports versions 3.1 and newer. For more information about installing and configuring Ganglia, see the project website at http://ganglia.sourceforge.net.

JMX Support

Since Hadoop is a Java-based system, supporting JMX is a relatively simple decision to make from a development perspective. Some monitoring systems, notably those written in Java themselves, even have first class support for JMX. JMX tends to be nice because it supports self-describing end points that enable monitoring systems (or any JMX client) to discover the available *MBeans* and their *attributes* (which would be analogous JMX-speak for a context and its metrics). JMX terminology, its RPC stack, security, and configuration, however, are tricky at best and downright convoluted the rest of the time. Nevertheless, if you know it, or already have a monitoring system that is natively JMX-aware, it's a perfectly valid option for integration.

JMX functionality is related to the metric plugins in a slightly unusual way. Internal MBeans in Hadoop rely on a metric plugin that has an update thread running to collect data from the system. With the default `NullContext` plugin, for instance, while it's possible to connect a JMX client to any of the Hadoop daemons and see a list of MBeans, their attributes will never update. This can be confusing and difficult to debug. Enabling any of the metric plugins that have an update thread (e.g. `NoEmitMetricsContext`, `GangliaContext`) will cause MBeans to show the correct information, as you would normally expect.

Table 10-1. Hadoop supported JMX MBeans

MBean Object Name	Description	Daemon
hadoop:service=NameNode,name=FSNamesystemState	Namenode metadata information.	Namenode
hadoop:service=NameNode,name=NameNodeActivity	Activity statistics on the namenode.	Namenode
hadoop:service=NameNode,name=NameNodeInfo	Descriptive namenode information.	Namenode
hadoop:service=DataNode,name=DataNodeActivity-*hostname-port*	Activity statistics for the datanode running on *hostname* and *port*.	Datanode
hadoop:service=DataNode,name=DataNodeInfo	Descriptive datanode information.	Datanode
hadoop:service=DataNode,name=FSDatasetState-DS-*ID*-127.0.0.1-*port-timestamp*	Datanode storage location statistics (`dfs.data.dir`) and information.	Datanode
hadoop:service=JobTracker,name=JobTrackerInfo	Descriptive jobtracker information.	Jobtracker
hadoop:service=TaskTracker,name=TaskTrackerInfo	Descriptive tasktracker information.	Tasktracker
hadoop:service=*ServiceName*,name=RpcActivityForPort*1234*	RPC information for *ServiceName* port *1234*.	All
hadoop:service=*ServiceName*,name=RpcDetailedActivityForPort*1234*	Detailed RPC information for *ServiceName* port *1234*. Tends not to be particularly useful.	All

REST Interface

The other common (and some might argue slightly more modern) method of getting at metric data is by way of a single-call REST / JSON servlet running in each daemon's embedded web server. The ubiquity of HTTP-based services, bevy of JSON parsing libraries, and the simplicity make this an extremely compelling option for many. Downsides to using REST / JSON include the lack of standardization and discoverability, HTTP overhead, and requirement for custom polling code to do something with the JSON blob returned from the service.

There are actually two such servlets: the original metrics servlet that lives at */metrics* and its updated replacement at */jmx*. Both provide similar functionality - an HTTP GET request that produces a JSON format response - but they work in very different ways. The */metrics* servlet uses the Hadoop metrics system directly, only works with metrics1, and is available in all versions of CDH3 as well as Apache Hadoop until version 0.20.203 (where it stops working due to the switch to metrics2). The newer */jmx* servlet, on the other hand, exposes Hadoop's JMX MBeans as JSON (which, we learned earlier, get

their information from the metrics system). This servlet is available in CDH starting with CDH3u1, but does not exist in Apache Hadoop until version 0.20.205.

Using the Metrics Servlet. Provided a metric plugin with an update thread is configured, pointing an HTTP client at the path /metrics will yield a plain text version of the metric data. Using the `format=json` query parameter, one can retrieve the same content in JSON format.

```
[esammer@hadoop01:~ ]$ curl http://hadoop117:50070/metrics
dfs
  FSDirectory
    {hostName=hadoop117,sessionId=}:
      files_deleted=100
  FSNamesystem
    {hostName=hadoop117,sessionId=}:
      BlockCapacity=2097152
      BlocksTotal=9
      CapacityRemainingGB=24
      CapacityTotalGB=31
      CapacityUsedGB=0
      CorruptBlocks=0
      ExcessBlocks=0
      FilesTotal=33
      MissingBlocks=0
      PendingDeletionBlocks=0
      PendingReplicationBlocks=0
      ScheduledReplicationBlocks=0
      TotalLoad=1
      UnderReplicatedBlocks=1
  namenode
    {hostName=vm02.localdomain,sessionId=}:
      AddBlockOps=1
      CreateFileOps=1
      DeleteFileOps=1
      FileInfoOps=12
      FilesAppended=0
...
```

Adding the query parameter `format=json` gives us the same information, but in JSON format.

```
[esammer@hadoop01:~ ]$ curl 'http://hadoop117:50070/metrics?format=json'
{"dfs":{
  "FSDirectory":[
    [{"hostName":"hadoop117","sessionId":""},{"files_deleted":100}]
  ],
  "FSNamesystem":[
    [
      {"hostName":"hadoop117","sessionId":""},
      {"BlockCapacity":2097152,"BlocksTotal":9,"CapacityRemainingGB":24,
       "CapacityTotalGB":31,"CapacityUsedGB":0,"CorruptBlocks":0,"ExcessBlocks":0,
       "FilesTotal":33,"MissingBlocks":0,"PendingDeletionBlocks":0,
       ...
```

Using the JMX JSON Servlet. Designed as a replacement to */metrics*, the */jmx* servlet produces logically similar output, but the data is sourced from Hadoop's JMX MBeans instead of the metrics system directly. This is the preferred servlet going forward as it works across both metrics1 and metrics2 based releases. Unlke */metrics*, this servlet does not support text output, only JSON. Its output should be self explanatory.

```
[esammer@hadoop01:~ ]$ curl http://hadoop117:50070/jmx
{
    "name" : "hadoop:service=NameNode,name=NameNodeActivity",
    "modelerType" : "org.apache.hadoop.hdfs.server.namenode.metrics.NameNodeActivtyMBean",
    "AddBlockOps" : 0,
    "fsImageLoadTime" : 2756,
    "FilesRenamed" : 0,
    "SyncsNumOps" : 0,
    "SyncsAvgTime" : 0,
    "SyncsMinTime" : 0,
    "SyncsMaxTime" : 70,
    "JournalTransactionsBatchedInSync" : 0,
    "FileInfoOps" : 0,
    "CreateFileOps" : 0,
    "GetListingOps" : 0,
    "TransactionsNumOps" : 0,
    "TransactionsAvgTime" : 0,
    "TransactionsMinTime" : 0,
    "TransactionsMaxTime" : 1,
    "GetBlockLocations" : 0,
    "BlocksCorrupted" : 0,
    "FilesInGetListingOps" : 0,
    "SafemodeTime" : 40117,
    "FilesCreated" : 0,
    "FilesAppended" : 0,
    "DeleteFileOps" : 0,
    "blockReportNumOps" : 0,
    "blockReportAvgTime" : 0,
    "blockReportMinTime" : 0,
    "blockReportMaxTime" : 3
}, {
    "name" : "java.lang:type=Threading",
    "modelerType" : "sun.management.ThreadImpl",
    "ThreadContentionMonitoringEnabled" : false,
    "DaemonThreadCount" : 29,
    "PeakThreadCount" : 38,
    "CurrentThreadCpuTimeSupported" : true,
    "ObjectMonitorUsageSupported" : true,
    "SynchronizerUsageSupported" : true,
    "ThreadContentionMonitoringSupported" : true,
    "ThreadCpuTimeEnabled" : true,
    ...
```

Apache Hadoop 0.20.203 and Later, and CDH4 (metrics2)

Starting with Apache Hadoop 0.20.203, the new metrics2 system is included and must be used. From the perspective of an administrator, the most noteworthy change is the

method of configuration and some of the nomenclature. Many of the concepts and functionality of metrics1 are preserved, albeit in a more general capacity.

One of the primary drawbacks of metrics1 was the one to one relationship of context to plugin. Supporting a more generic system where metrics could be handled by multiple plugins was necessary. In metrics2, we refer to metrics sources and sinks. The former are components that generate metrics while the latter consume them. This is analogous to the relationship between a context and its configured plugin in the older metrics1 system. Components of Hadoop that wish to produce metrics implement the `MetricsSource` interface or use a set of simple Java annotations, while those that wish to receive and process metric data implement the `MericsSink` interface. The framework, based on the administrator-provided configuration, handles getting the metrics from sources to sinks.

By default, all metrics from all sources are delivered to all sinks. This is the desired behaviour in most cases; to deliver all metrics to a single file, or to Ganglia, for example. When more elaborate routing of data is required, one can filter metrics by the context to which they belong, as well as other so called tags. Filters can be applied to a source, record, or even metric name. Note that the more specific the filter, the greater the overhead incurred during metric processing, as you might expect.

System configuration is done by way of the *hadoop-metrics2.properties* (note the subtle number two in the name) file in the standard Hadoop configuration directory. Like its predecessor, *hadoop-metrics2.properties* is a Java properties file, but uses a few special conventions to express defaults and overrides. In Example 10-5, a simple metrics2 configuration file is shown.

Example 10-5. Sample hadoop-metrics2.properties configuration file

```
# hadoop-metrics2.properties

# By default, send metrics from all sources to the sink
# named 'file', using the implementation class FileSink.
*.sink.file.class = org.apache.hadoop.metrics2.sink.FileSink

# Override the parameter 'filename' in 'file' for the namenode.
namenode.sink.file.filename = namenode-metrics.log

# Send the jobtracker metrics into a separate file.
jobtracker.sink.file.filename = jobtracker-metrics.log
```

Each property name in the configuration file has four components: the prefix, type, instance, and option, in that order. For instance, in the property name-node.sink.file.filename, namenode is the prefix, sink is the type, file is the instance, and filename is the option. These components can be replaced by an asterisk (*) to indicate an option should act as a default. For more information on the advanced features of metrics2, see the Javadoc.

What About SNMP?

Most system administrators have encountered the Simple Network Management Protocol (or SNMP) at one time or another. SNMP, like JMX, is a standard for extracting metrics from a service or device and is supported by many, if not all, monitoring systems. If JMX could be called cryptic, SNMP borders on alien, with equally obtuse terminology and a long history of security issues (while this is theoretically resolved with SNMPv3 it's still exceedingly complicated). Hadoop has no direct SNMP support and no defined MIB module. Users are encouraged to use JMX as it offers similar functionality and industry adoption if they'd rather not write custom code to interface with the JSON servlet.

Health Monitoring

Once Hadoop is properly configured and integrated with your monitoring system of choice, the next step is to decide which metrics are important and which aren't. Even more so, the question of which metrics most accurately represent the health of a given service is the critical question. Since there are dependencies between services, it usually makes sense to model this relationship in the monitoring system to quell spurious alerts. For instance, if HDFS isn't available, it probably doesn't make sense to ring the alarm for MapReduce. There are, however, a handful of metrics for each service that act as the canary in the coal mine, so to speak.

Metric selection is only half the battle when configuring monitoring systems. Equally important are the thresholds we establish to indicate alert conditions. The problem with monitoring, in general, is that, on any active system, as soon as a measurement is taken, it is immediately outdated. Assume for a minute a host check reveals the disk containing the namenode metadata is 93% full. It's unclear that this is alert worthy because, on a 1TB disk, this means almost 72GB is still available. Knowing the total size of the disk is necessary, but it doesn't reveal the full story because we still don't know the *rate* of growth. If disk space is consumed at a rate of 1GB per day on average, 72GB is more than sufficient and shouldn't yield an alert. Remember always that the rate of growth must also factor in the rate of attrition of old data which, in some cases, can be independently controlled. Aggressive log pruning is a good example of compensating for resource consumption by reducing the retention rate of other data that may exist on the same device.

So what does all of this mean when establishing alert thresholds? Here are some basic rules:

- All thresholds are fluid. As cluster usage and resource consumption changes, so too should alert thresholds.
- For new clusters, start with a conservative value for a threshold, measure usage, and refine over time. Remember that utilization patterns can occur at different intervals. For example, it's not uncommon to have a significant bump in activity

for clusters primarily used for analytics during office hours, at the end of the month, the end of the quarter, and the end of the year.

- A high signal to noise ratio for alerts is absolutely critical. Overly sensitive alerting will desensitize an operations team to the criticality of the event, and fatigues staff.

Host Level Checks

The host on which the various daemons run must have the requisite local disk capacity, free memory, and minimal amount of CPU capacity. Some services require significant (read: as much as possible) network bandwidth while others are far more sensitive to latency. Here, we'll focus on what checks should be place on any Hadoop cluster.

Most Hadoop daemons use the local disk in one way or another. Historically, many of the daemons have not taken kindly to exhausting the available storage resources on a host. In the case of the namenode, for instance, it was trivial to accidentally corrupt the metadata by filling the disk on which the edit log was stored until recent versions (which admittedly only reduce the chance, not eliminate the possibility). While the level of maturity of the software has increased over time, better still is to not test such scenarios.

Recommendation: Monitor local disk consumption of the namenode metadata (`dfs.name.dir`) and log data (`HADOOP_LOG_DIR`) directories. To estimate the immediate rate of growth, use a 14 day rolling average and alert when the remaining capacity is below 5 to 30 days' capacity, depending on how long it normally takes you to mitigate the situation. The low end of five days is recommended to provide enough overlap for long weekends.

Datanode (`dfs.data.dir`) directories differ from most others in that they exist to permanently store huge amounts of data. When a datanode directory fills to capacity, it simply stops receiving new blocks. If all datanode directories become full, the namenode will stop placing blocks on that datanode, effectively treating it as read-only. Remember that if you choose to colocate MapReduce local data and datanode data on the same devices that you should properly configure `dfs.datanode.du.reserved` to ensure capacity remains for task temporary data (see `dfs.datanode.du.reserved` on page 89 for more information). MapReduce local (`mapred.local.dir`) directories have spiky usage patterns and, as a result, their capacity can be difficult to monitor in a meaningful way. All data in these directories is transient and will be cleaned up when it's no longer needed by running jobs whether they succeed or fail. Should all volumes supporting `mapred.local.dir` become invalid due to capacity exhaustion or disk failures, the tasktracker becomes unusable by the system. Any tasks assigned to that node will inevitably fail and the tasktracker will eventually be blacklisted by the jobtracker.

Recommendation: Monitor `dfs.data.dir` and `mapred.local.dir` directories' volumes SMART errors and capacity, but avoid alerting at this level, if possible. Instead, prefer higher level checks on the aggregate HDFS and MapReduce service level metrics for alerts, and warnings for individual hosts and disks. To put it another way, a disk failure

in a worker node is not something that an administrator should wake up for unless it impacts the larger service.

Physical and virtual memory utilization are of concern when evaluating the health of a host in a cluster. Each daemon's maximum heap allocation (roughly) defines its worst-case memory footprint. During configuration, the sum of the memory consumption of all daemons should never exceed physical memory. Provided this is true, and there's nothing else running on the hosts in a Hadoop cluster, these machines should never swap. Should a host begin swapping, a ripple effect can occur that drastically decreases performance at best, and causes failures at worst.

Recommendation: Monitor that the number of pages (or amount of data in bytes, whatever is easier) swapped in and out to disk, per second, does not exceed zero, or some very small amount. Note that this is not the same as monitoring the amount of swap space consumed; a common mistake. The Linux kernel is not obliged to reclaim swap space previously consumed, until it is needed, even after the data is swapped back in, so monitoring the utilization of the swap file(s) or partition(s) is almost guaranteed to generate false positives.

Processor utilization - specifically load average - is one of the most frequently misunderstood metrics people monitor on hosts. The CPU load average in Linux is the average number of processes in a runnable state on a machine, over a rolling window of time. Normally, three discreet values are given: a five, ten, and fifteen minute average. A contrived, "perfect utilization" example of this would be a five minute load average of 1.00 on a single core CPU, meaning, over the last five minutes, the core was fully occupied by a running process. Modern server CPUs have between four and eight physical cores (at the time of this writing) and can support a perfect utilization of a number greater than 1.00. In other words, you should fully expect, and target, the load average of a Hadoop worker node to be roughly equal to the number of cores, either physical, or virtual if you have a feature like Hyper-Threading enabled. Of course, it's rare to achieve such perfection in the real world. The point is to absolutely not set alert thresholds on load average on a large Hadoop cluster unless you wish to regularly swim in a sea of false alerts.

Recommendation: Track CPU metrics like load average for performance and utilization measurement, but do not alert on it unless there's an unusually compelling reason (hint: there isn't).

Network bandwidth consumption is similar to CPU utilization in the context of monitoring. Ideally, this is a metric we track because it tells us something about the behavior of HDFS data ingress and egress, or running MapReduce jobs, but there's no prescribed upper limit where it makes sense to alert. Latency, on the other hand, can cause some services to fail, outright. This primarily impacts HBase, where a region server becoming even temporarily disconnected from its ZooKeeper quorum can cause it to shut down and possibly even precipitate a cascading failure, but is still a point of concern. The

RPC metrics from the individual daemons provide better insight into the observed latency and so we measure it there.

Recommendation: Network latency checks are best measured by looking at the total RPC latency, as reported by the daemons themselves since that's really what we're interested in. Bandwidth utilization is interesting from a performance perspective and can help detect (or eliminate) possible bottlenecks; something we'll look at later.

Arguably a host level check, process presence is a common, simple, check performed to indicate good health. Unfortunately, the mere presence of a Hadoop process isn't a reliable indication that a service is alive or healthy because of the distributed nature of the system. It's very possible for a daemon to be alive and running, but be unable to communicate with other nodes in the cluster for whatever reason. The uncertainty of this method of monitoring coupled with the fact that Hadoop does provide meaningful metrics from each daemon means process presence checks are probably best skipped in Hadoop clusters. This also reduces the amount of potential noise generated from process level restarts during maintenance operations, which is a nice side effect.

Recommendation: Skip process presence checks altogether when monitoring hosts, instead deferring to daemon and service level metric checks described later.

All Hadoop Processes

All Hadoop processes are Java processes and, as a result, have a common set of checks that should be performed. One of the most important checks to perform, especially in the instance of the namenode and jobtracker, is the available heap memory remaining within the JVM. Using the JMX REST servlet, for example, it's possible to discover the initial, maximum, and used heap (Example 10-6). Unfortunately, determining the health of a process is not as simple as just subtracting the used heap from the max because of the way garbage collection works. The amount of *used* memory is always less than or equal to the committed memory, and represents the amount of heap that actually used by objects. The *committed* memory is the current size of the heap. That is, the amount of memory that has been allocated from the operating system and can be immediately consumed by objects, if it is needed. Committed memory is always less than or equal to the maximum heap size. Should the amount of used memory exceed (or need to exceed) the amount of committed memory, and the committed memory is below the max, the amount of committed memory is increased. On the other hand, if an application attempts to grow its usage beyond the current committed memory, but the committed memory can not expand anymore due to the max, the JVM will run out of memory, resulting in the dreaded `OutOfMemoryError`.

Example 10-6. Sample JVM memory JSON metric data from the namenode

```
{
    "name" : "java.lang:type=Memory",
    "modelerType" : "sun.management.MemoryImpl",
    "Verbose" : false,
```

```
    "HeapMemoryUsage" : {
      "committed" : 1234829312,
      "init" : 1584914432,
      "max" : 7635533824,
      "used" : 419354712
    },
    "NonHeapMemoryUsage" : {
      "committed" : 47841280,
      "init" : 24313856,
      "max" : 136314880,
      "used" : 47586032
    },
    "ObjectPendingFinalizationCount" : 0
  }
```

It is very common that the amount of used memory will increase over time until it comes extremely close to the committed or even maximum thresholds, but immediately jump back down. This is the result of a garbage collection event, and it is perfectly normal. In fact, this is how it is supposed to work. It's also why monitoring memory can be difficult.

One option to monitoring a dynamic heap like this is to measure the median used heap over a set of samples rather than the absolute value at a point in time. If the median remains high for a period of time (which needs to be long enough for at least one full garbage collection to occur), it's probably not going to change and an alert should be triggered. Alternatively, if you are concerned about false positives and you want to ensure a full garbage collection has occurred, there are metrics that expose the number of full GC events per pool that can be combined with a check of the median.

Recommendation: Perform heap monitoring on the namenode, jobtracker, and secondary namenode processes using the technique described above.

In addition to the amount of memory available, some applications are sensitive to the duration of garbage collection events. This tends to plague low latency services especially where, because the JVM spends a long time dealing with garbage, it's unable to service requests in a timely manner. For an application like the namenode, this can be particularly disruptive. Being unable to answer a request for even a couple of seconds can be the same as not answering it at all. Included in the JVM level metrics, you will find a metrics for each memory pool that indicate both the number of GC events as well as the average time it took.

Recommendation: Monitor the average time spent performing garbage collection for the namenode and jobtracker. Tolerance for these pauses before failure occurs will vary by application, but almost all will be negatively impacted in terms of performance.

HDFS Checks

One of the advantages of HDFS is that the namenode, as a side effect of keeping track of all datanodes, has an authoritative picture of the state of the cluster. If, for some

reason, the namenode becomes unavailable, it's impossible to perform any HDFS operations. Whatever the namenode sees is what clients will see when they access the filesystem. This saves us the otherwise difficult task of correlating the events and state information of the datanodes, leaving no room for nondeterminism in health checks.

Earlier, a general recommendation was made to prefer service level health and metric checks. Using the aggregate view from the namenode, we see a more complete picture of HDFS, as a service. Most of the critical health checks performed use the metric data produced by the namenode. In the following sections, we'll use the output of the JMX JSON servlet because of its readability, but you should be able to access this data using any of the methods described earlier. Within the namenode's metrics you will find an MBean called `Hadoop:service=NameNode,name=NameNodeInfo`, shown in Example 10-7, which contains high level HDFS information.

Example 10-7. Sample NameNodeInfo JSON metric data from the namenode

```
{
    "name" : "Hadoop:service=NameNode,name=NameNodeInfo",
    "modelerType" : "org.apache.hadoop.hdfs.server.namenode.FSNamesystem",
    "Threads" : 55,
    "Total" : 193868941611008,
    "ClusterId" : "CID-908b11f7-c752-4753-8fbc-3d209b3088ff",
    "BlockPoolId" : "BP-875753299-172.29.121.132-1340735471649",
    "Used" : 50017852084224,
    "Version" : "2.0.0-cdh4.0.0, r5d678f6bb1f2bc49e2287dd69ac41d7232fc9cdc",
    "PercentUsed" : 25.799828,
    "PercentRemaining" : 68.18899,
    "Free" : 132197265809408,
    "Safemode" : "",
    "UpgradeFinalized" : false,
    "NonDfsUsedSpace" : 11653823717376,
    "BlockPoolUsedSpace" : 50017852084224,
    "PercentBlockPoolUsed" : 25.799828,
    "TotalBlocks" : 179218,
    "TotalFiles" : 41219,
    "NumberOfMissingBlocks" : 4725,
    "LiveNodes" : "{
      \"hadoop02.cloudera.com\":{
        \"numBlocks\":40312,
        \"usedSpace\":5589478342656,
        \"lastContact\":2,
        \"capacity\":21540992634880,
        \"nonDfsUsedSpace\":1287876358144,
        \"adminState\":\"In Service\"
      },
      ...
    "DeadNodes" : "{}",
    "DecomNodes" : "{}",
    "NameDirStatuses" : "{
      \"failed\":{},
      \"active\":{
        \"/data/2/hadoop/dfs/nn\":\"IMAGE_AND_EDITS\",
        \"/data/1/hadoop/dfs/nn\":\"IMAGE_AND_EDITS\"
```

```
    }
  }"
}
```

The output above is slightly reformatted for readability, and only a single datanode is shown in the `LiveNodes` map, but it should otherwise be self explanatory. Some fields contain strings that are, in turn, distinct JSON blobs. While this is awkward in the JSON output, it's a side effect of the way the underlying JMX MBeans store these values. If you do decide to use the JMX JSON servlet, expect to encounter this in a few places. In the above example, you can see this in the `LiveNodes`, `DeadNodes`, and `NameDirStatuses`, for example.

Recommendation: From this data, perform the following critical health checks:

- The absolute amount of free HDFS capacity in bytes (`Free`) is greater than an acceptable threshold.
- The absolute number of active (`NameDirStatuses["active"]`) metadata paths is equal to those specified in `dfs.name.dir`, or failed (`NameDirStatuses["failed"]`) paths is equal to zero.

The next set of checks uses metrics found in the `Hadoop:service=NameNode,name=FSNamesystem` MBean (see Example 10-8). Here you will find filesystem level metrics about files, blocks, edit log transactions, checkpoint operations and more.

Example 10-8. Sample FSNamesystem JSON metric data from the namenode

```
{
    "name" : "Hadoop:service=NameNode,name=FSNamesystem",
    "modelerType" : "FSNamesystem",
    "tag.Context" : "dfs",
    "tag.HAState" : "active",
    "tag.Hostname" : "hadoop01.cloudera.com",
    "MissingBlocks" : 4725,
    "ExpiredHeartbeats" : 2,
    "TransactionsSinceLastCheckpoint" : 58476,
    "TransactionsSinceLastLogRoll" : 7,
    "LastWrittenTransactionId" : 58477,
    "LastCheckpointTime" : 1340735472996,
    "CapacityTotalGB" : 180555.0,
    "CapacityUsedGB" : 46583.0,
    "CapacityRemainingGB" : 123118.0,
    "TotalLoad" : 9,
    "BlocksTotal" : 179218,
    "FilesTotal" : 41219,
    "PendingReplicationBlocks" : 0,
    "UnderReplicatedBlocks" : 4736,
    "CorruptBlocks" : 0,
    "ScheduledReplicationBlocks" : 0,
    "PendingDeletionBlocks" : 0,
    "ExcessBlocks" : 0,
    "PostponedMisreplicatedBlocks" : 0,
    "PendingDataNodeMessageCount" : 0,
```

```
    "MillisSinceLastLoadedEdits" : 0,
    "BlockCapacity" : 8388608,
    "TotalFiles" : 41219
}
```

You may notice redundant information between `NameNodeInfo` and `FSNamesystem` like the former's `Total` and the latter's `CapacityTotalGB` (albeit in a different unit). In fact, if you look closer, there's even redundant information in FSNamesystem itself (see `TotalFiles` and `FilesTotal`). All of these numbers are computed using the same snapshot of the metrics and, as a result, should be consistent with one another.

Recommendation: Create health checks for the following:

- The absolute number of missing (`MissingBlocks`) and corrupt blocks (`CorruptBlocks`) are less than a acceptable threshold. Both of these metrics should be zero, ideally.
- The absolute number of HDFS blocks that can still be allocated (`BlockCapacity`). This is the maximum number of blocks a namenode will track before it refuses new file creation. It exists to prevent accidental out of memory errors due to over-allocation. Increasing the namenode heap size automatically adjusts the total number of blocks allowed.
- The result of the current epoch time minus the last time a namenode checkpoint was performed (`LastCheckpointTime`) is less than three days. In the above example, `LastCheckpointTime` is 1340735472996 or Tuesday, June 26th 11:31:12 PDT 2012. Assuming the current time is 1341356207664 or Tuesday Jul 03 15:56:47 PDT 2012, this yields 620734668 or approximately 7.1 days (because: (1340735472996 - 1341356207664) / 1000 / 60 / 60 / 24 = 7.1). Note that epoch timestamps generated by Java are expressed in milliseconds, not seconds, hence the division by 1000 in our example.

MapReduce Checks

As with HDFS, most of the monitoring of MapReduce is done at the master process, in this case, the jobtracker. There are two major components to monitoring MapReduce: monitoring the framework, and monitoring individual jobs. Job level monitoring is not a good fit for most monitoring systems, and can be incredibly application specific. Instead, we'll focus on monitoring the MapReduce framework.

Using the same JMX REST servlet as we did earlier, let's take a look at some of the available metrics in Example 10-9.

Example 10-9. Sample JobTrackerInfo JSON metric data from the jobtracker

```
{
    "name" : "hadoop:service=JobTracker,name=JobTrackerInfo",
    "modelerType" : "org.apache.hadoop.mapred.JobTracker",
    "Hostname" : "m0507",
    "Version" : "2.0.0-mr1-cdh4.0.1, rUnknown",
```

```
  "ConfigVersion" : "default",
  "ThreadCount" : 45,
  "SummaryJson" : "{
    \"nodes\":8,
    \"alive\":8,
    \"blacklisted\":0,
    \"slots\":{
      \"map_slots\":128,
      \"map_slots_used\":128,
      \"reduce_slots\":48,
      \"reduce_slots_used\":0
    },
    \"jobs\":3
  }",
  "AliveNodesInfoJson" : "[
    {
      \"hostname\":\"hadoop01.cloudera.com\",
      \"last_seen\":1343782287934,
      \"health\":\"OK\",
      \"slots\":{
        \"map_slots\":16,
        \"map_slots_used\":16,
        \"reduce_slots\":6,
        \"reduce_slots_used\":0
      },
      \"failures\":0,
      \"dir_failures\":0
    },
    // Remaining hosts omitted...
  ]",
  "BlacklistedNodesInfoJson" : "[]",
  "QueueInfoJson" : "{\"default\":{\"info\":\"N/A\"}}"
}
```

Aggregate cluster metrics are provided as JSON blob in the `SummaryJson` field. The total number of nodes currently alive, total map slots available and used, total reduce slots available and used, and total job submissions are also available.

Recommendation: Perform the following cluster level checks:

- Check that the number of alive nodes is within a tolerance that still allows your jobs to complete within their service level agreement. Depending on the size of your cluster and the criticality of jobs, this will vary.
- Check that the number of blacklisted tasktrackers is below some percentage of the total number of tasktrackers in the cluster. A blacklisted tasktracker is one that is alive, but demonstrating repeated failures across jobs. This number should ideally be zero, although it's possible to trigger false positives if a user or system rapidly submits a series of poorly written jobs that ultimately fail. On the other hand, that is almost certainly indicative of a problem that should be dealt with as well, although not specifically a cluster issue.

You can see that like the namenode, the jobtracker reports all of its workers (tasktrackers, in this case) and their status. These are hosts from which the aggregate cluster information is taken. Be careful not to over-monitor individual tasktrackers as it's not unusual to have intermittent failures at scale. You may optionally choose to monitor for `dir_failures`, which represent a directory in `mapred.local.dir` that has been ignored by a tasktracker. This is usually a sign of a impending (or complete) disk failure within the machine. If a datanode shares the same underlying device, expect this to impact it as well. The `failures` field is a counter of the number of task failures that occurred on that specific tasktracker. It's common for some number of failures to occur so it's hard to provide a concrete recommendation as to a specific threshold. Instead, you may optionally monitor the deviation of tasktracker's number of failures from the average over the others.