Hadoop

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Decimal** | | | | **Value** | [**Metric**](https://en.wikipedia.org/wiki/Metric_prefix) | | | 1000 | k | [kilo](https://en.wikipedia.org/wiki/Kilo-) | | 10002 | M | [mega](https://en.wikipedia.org/wiki/Mega-) | | 10003 | G | [giga](https://en.wikipedia.org/wiki/Giga-) | | 10004 | T | [tera](https://en.wikipedia.org/wiki/Tera-) | | 10005 | P | [peta](https://en.wikipedia.org/wiki/Peta-) | | 10006 | E | [exa](https://en.wikipedia.org/wiki/Exa-) | | 10007 | Z | [zetta](https://en.wikipedia.org/wiki/Zetta-) | | 10008 | Y | [yotta](https://en.wikipedia.org/wiki/Yotta-) | |  |

# Data Storage and Analysis

One typical drive from 1990 could store 1,370 MB of data and had

a transfer speed of 4.4 MB/s,4 so you could read all the data from a full drive in around five minutes. Over 20 years later, one terabyte drives are the norm, but the transfer speed is around 100 MB/s, so it takes more than two and a half hours to read all the data off the disk.

This is a long time to read all data on a single drive—and writing is even slower. The obvious way to reduce the time is to read from multiple disks at once. Imagine if we had 100 drives, each holding one hundredth of the data. Working in parallel, we could read the data in under two minutes.

The first problem to solve is hardware failure: as soon as you start using many pieces of hardware, the chance that one will fail is fairly high. A common way of avoiding data loss is through replication: redundant copies of the data are kept by the system so that in the event of failure, there is another copy available.

The second problem is that most analysis tasks need to be able to combine the data in some way, and data read from one disk may need to be combined with the data from any of the other 99 disks. Various distributed systems allow data to be combined from multiple sources, but doing this correctly is notoriously challenging. MapReduce provides a programming model that abstracts the problem from disk reads and writes, transforming it into a computation over sets of keys and values.

For example, Mailtrust, Rackspace’s mail division, used Hadoop for processing email logs. One ad hoc query they wrote was to find the geographic distribution of their users. In their words: This data was so useful that we’ve scheduled the MapReduce job to run monthly and we

will be using this data to help us decide which Rackspace data centers to place new mail servers in as we grow.

# Beyond Batch

MapReduce is fundamentally a batch processing system, and is not

suitable for interactive analysis.

The first component to provide online access was HBase, a key-value store that uses HDFS for its underlying storage. HBase provides both online read/write access of individual rows and batch operations for reading and writing data in bulk, making it a good solution for building applications on.

The real enabler for new processing models in Hadoop was the introduction of YARN (which stands for Yet Another Resource Negotiator) in Hadoop 2. YARN is a cluster resource management system, which allows any distributed program (not just MapReduce) to run on data in a Hadoop cluster.

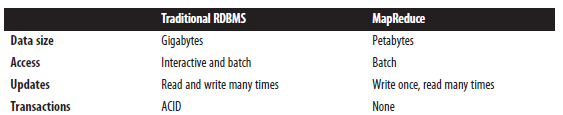
In the last few years there has been a flowering of different processing patterns that work with Hadoop. Here is a sample:

• Interactive SQL. By dispensing with MapReduce, and using a distributed query engine that uses dedicated “always on” daemons (like Impala), or container reuse (like Hive on Tez), it’s possible to achieve low-latency responses for SQL queries on Hadoop, while still scaling up to large dataset sizes.

• Iterative processing. Many algorithms—such as those in machine learning—are iterative in nature, so it’s much more efficient to be able to hold each intermediate working set in memory, compared to loading from disk on each iteration. The architecture of MapReduce does not allow this, but it’s straightforward with Spark, for example, and it enables a highly-exploratory style of working with datasets.

• Stream processing. Streaming systems like Storm, Spark Streaming, or Samza make it possible to run real-time, distributed computations on unbounded streams of data and emit results to Hadoop storage or external systems.

• Search. The Solr search platform can run on a Hadoop cluster, indexing documents as they are added to HDFS, and serving search queries from indexes stored in HDFS.





Another difference between Hadoop and an RDBMS is the amount of structure in the datasets on which they operate. Structured data is data that is organized into entities that have a defined format, such as XML documents or database tables that conform to a particular predefined schema. This is the realm of the RDBMS. Semi-structured data,

on the other hand, is looser, and though there may be a schema, it is often ignored, so it may be used only as a guide to the structure of the data: for example, a spreadsheet, in which the structure is the grid of cells, although the cells themselves may hold any form of data. Unstructured data does not have any particular internal structure: for example, plain text or image data.

Hadoop works well on unstructured or semi structured data because it is designed to interpret the data at processing time, so called schema-on-read. This provides flexibility, and avoids the costly data loading phase of

an RDBMS, since in Hadoop it is just a file copy.

Relational data is often *normalized* to retain its integrity and remove redundancy.

MapReduce—and the other processing models in Hadoop—scale linearly with the size

of the data. Data is partitioned, and the functional primitives (like map and reduce) can

work in parallel on separate partitions. This means that if you double the size of the

input data, a job will run twice as slowly. But if you also double the size of the cluster, a job will run as fast as the original one. This is not generally true of SQL queries.

Hadoop tries to co-locate the data with the compute node, so data access is fast because

it is local.6 This feature, known as *data locality*, is at the heart of data processing in

Hadoop and is the reason for its good performance. Recognizing that network bandwidth

is the most precious resource in a data center environment (it is easy to saturate

network links by copying data around), Hadoop goes to great lengths to conserve it by

explicitly modelling network topology. Notice that this arrangement does not preclude

high-CPU analyses in Hadoop.

Distributed processing frameworks like MapReduce spare the programmer from having

to think about failure, since the implementation detects failed tasks and reschedules

replacements on machines that are healthy. MapReduce is able to do this because it is a

*shared-nothing* architecture, meaning that tasks have no dependence on one other. (This

is a slight oversimplification, since the output from mappers is fed to the reducers, but

this is under the control of the MapReduce system; in this case, it needs to take more

care rerunning a failed reducer than rerunning a failed map because it has to make sure

it can retrieve the necessary map outputs, and if not, regenerate them by running the

relevant maps again

# MapReduce

MapReduce is a programming model for data processing. Hadoop can run MapReduce programs written in various languages; Most important, MapReduce programs are inherently parallel,

Map and Reduce

MapReduce works by breaking the processing into two phases: the map phase and the

reduce phase. Each phase has key-value pairs as input and output, the types of which

may be chosen by the programmer. The programmer also specifies two functions: the

map function and the reduce function.

A Weather Dataset

For our example, we will write a program that mines weather data. Weather sensors

collect data every hour at many locations across the globe and gather a large volume of

log data, which is a good candidate for analysis with MapReduce because we want to

process all the data, and the data is semi-structured and record-oriented.

To visualize the way the map works, consider the following sample lines of input data

(some unused columns have been dropped to fit the page, indicated by ellipses):

0067011990999991950051507004*...*9999999N9+00001+99999999999*...*

0043011990999991950051512004*...*9999999N9+00221+99999999999*...*

0043011990999991950051518004*...*9999999N9-00111+99999999999*...*

0043012650999991949032412004*...*0500001N9+01111+99999999999*...*

0043012650999991949032418004*...*0500001N9+00781+99999999999*...*

These lines are presented to the map function as the key-value pairs:

(0, 006701199099999**1950**051507004...9999999N9+**0000**1+99999999999...)

(106, 004301199099999**1950**051512004...9999999N9+**0022**1+99999999999...)

(212, 004301199099999**1950**051518004...9999999N9-**0011**1+99999999999...)

(318, 004301265099999**1949**032412004...0500001N9+**0111**1+99999999999...)

(424, 004301265099999**1949**032418004...0500001N9+**0078**1+99999999999...)

The keys are the line offsets within the file, which we ignore in our map function. The

map function merely extracts the year and the air temperature (indicated in bold text),

and emits them as its output (the temperature values have been interpreted as integers):

(1950, 0)

(1950, 22)

(1950, −11)

(1949, 111)

(1949, 78)

The output from the map function is processed by the MapReduce framework before

being sent to the reduce function. This processing sorts and groups the key-value pairs

by key. So, continuing the example, our reduce function sees the following input:

(1949, [111, 78])

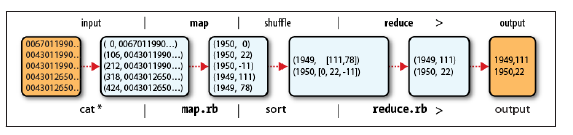
(1950, [0, 22, −11])

Each year appears with a list of all its air temperature readings. All the reduce function has to do now is iterate through the list and pick up the maximum reading:

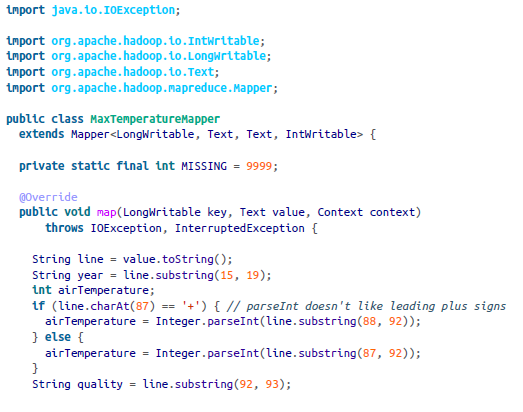
(1949, 111)

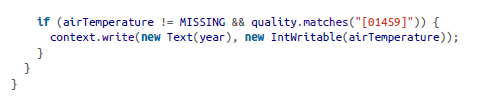
(1950, 22)

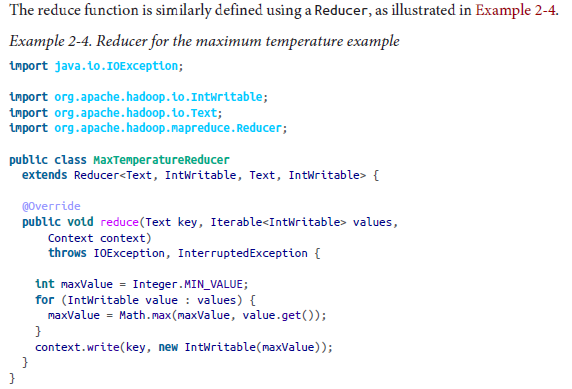
Analyzing

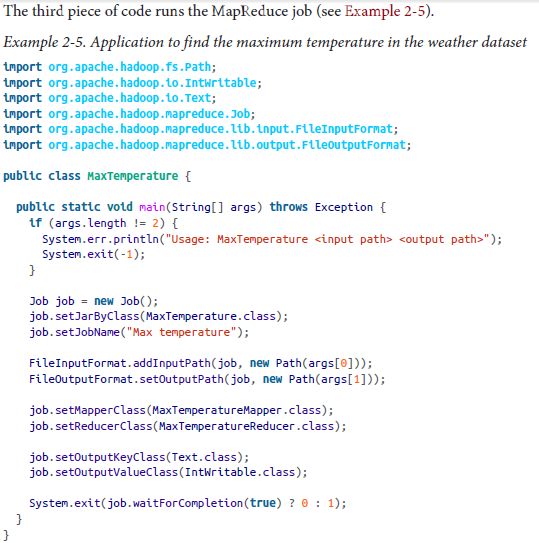


Java MapReduce









# Data Flow

First, some terminology. A MapReduce *job* is a unit of work that the client wants to be

performed: it consists of the input data, the MapReduce program, and configuration

30 | Chapter 2: MapReduce

information. Hadoop runs the job by dividing it into *tasks*, of which there are two types:

*map tasks* and *reduce tasks*. The tasks are scheduled using YARN and run on nodes in

the cluster. If a task fails, it will be automatically rescheduled to run on a different node.

Hadoop divides the input to a MapReduce job into fixed-size pieces called *input splits*,

or just *splits*. Hadoop creates one map task for each split, which runs the user-defined

map function for each *record* in the split.

Having many splits means the time taken to process each split is small compared to the

time to process the whole input. So if we are processing the splits in parallel, the processing is better load-balanced when the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower

machine. Even if the machines are identical, failed processes or other jobs running

concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained.

On the other hand, if splits are too small, the overhead of managing the splits and of

map task creation begins to dominate the total job execution time. For most jobs, a good

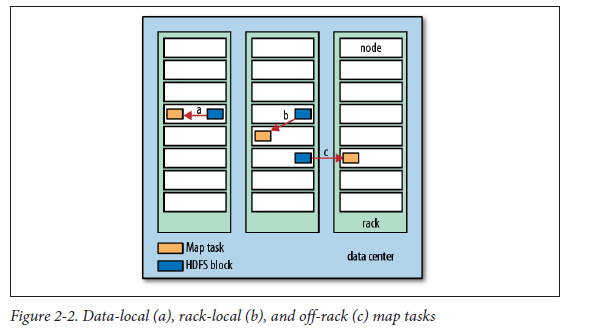
split size tends to be the size of an HDFS block, 128 MB by default, although this can

be changed for the cluster (for all newly created files) or specified when each file is

created.

Hadoop does its best to run the map task on a node where the input data resides in

HDFS. This is called the *data locality optimization* because it doesn’t use valuable cluster bandwidth. Sometimes, however, all three nodes hosting the HDFS block replicas for a map task’s input split are running other map tasks, so the job scheduler will look for a free map slot on a node in the same rack as one of the blocks. Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer. The three possibilities are illustrated in Figure 2-2.



It should now be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so

some of the split would have to be transferred across the network to the node running

the map task, which is clearly less efficient than running the whole map task using local

data.

Map tasks write their output to the local disk, not to HDFS. Why is this? Map output is

intermediate output: it’s processed by reduce tasks to produce the final output, and once the job is complete, the map output can be thrown away. So storing it in HDFS with

replication would be overkill. If the node running the map task fails before the map

output has been consumed by the reduce task, then Hadoop will automatically rerun

the map task on another node to re-create the map output.

Reduce tasks don’t have the advantage of data locality; the input to a single reduce task is normally the output from all mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore, the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the

reduce is normally stored in HDFS for reliability. As explained in Chapter 3, for each

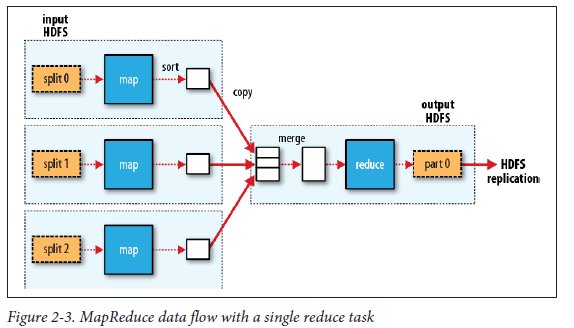
HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes for reliability. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline

consumes.

The whole data flow with a single reduce task is illustrated in Figure 2-3. The dotted

boxes indicate nodes, the light arrows show data transfers on a node, and the heavy

arrows show data transfers between nodes.



The number of reduce tasks is not governed by the size of the input, but instead is

specified independently. In “The Default MapReduce Job” on page 216, you will see how

to choose the number of reduce tasks for a given job.

When there are multiple reducers, the map tasks *partition* their output, each creating

one partition for each reduce task. There can be many keys (and their associated values)

in each partition, but the records for any given key are all in a single partition. The

partitioning can be controlled by a user-defined partitioning function, but normally the

default partitioner—which buckets keys using a hash function—works very well.

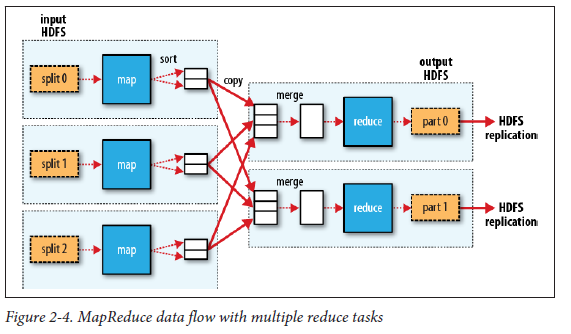
The data flow for the general case of multiple reduce tasks is illustrated in Figure 2-4.

This diagram makes it clear why the data flow between map and reduce tasks is colloquially

known as “the shuffle,” as each reduce task is fed by many map tasks. The shuffle

is more complicated than this diagram suggests, and tuning it can have a big impact on

job execution time, as you will see in “Shuffle and Sort” on page 199.

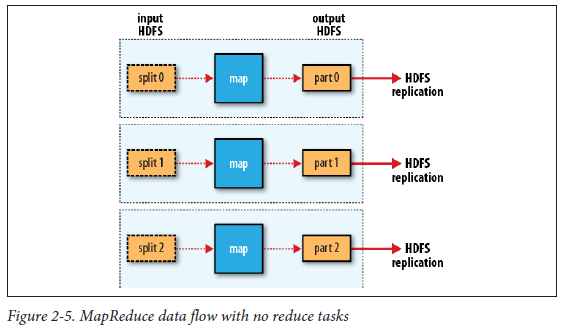


Finally, it’s also possible to have zero reduce tasks. This can be appropriate when you

don’t need the shuffle because the processing can be carried out entirely in parallel (a

few examples are discussed in “NLineInputFormat” on page 236). In this case, the only

off-node data transfer is when the map tasks write to HDFS (see Figure 2-5).



Combiner Functions

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays

to minimize the data transferred between map and reduce tasks. Hadoop allows the user

to specify a *combiner function* to be run on the map output, and the combiner function’s

output forms the input to the reduce function. Because the combiner function is an

optimization, Hadoop does not provide a guarantee of how many times it will call it for

a particular map output record, if at all. In other words, calling the combiner function

zero, one, or many times should produce the same output from the reducer

The contract for the combiner function constrains the type of function that may be used.

This is best illustrated with an example. Suppose that for the maximum temperature

example, readings for the year 1950 were processed by two maps (because they were in

different splits). Imagine the first map produced the output:

(1950, 0)

(1950, 20)

(1950, 10)

and the second produced:

(1950, 25)

(1950, 15)

The reduce function would be called with a list of all the values:

(1950, [0, 20, 10, 25, 15])

with output:

(1950, 25)

since 25 is the maximum value in the list. We could use a combiner function that, just

like the reduce function, finds the maximum temperature for each map output. The

reduce would then be called with:

(1950, [20, 25])

The combiner function doesn’t replace the reduce function. (How could it? The reduce

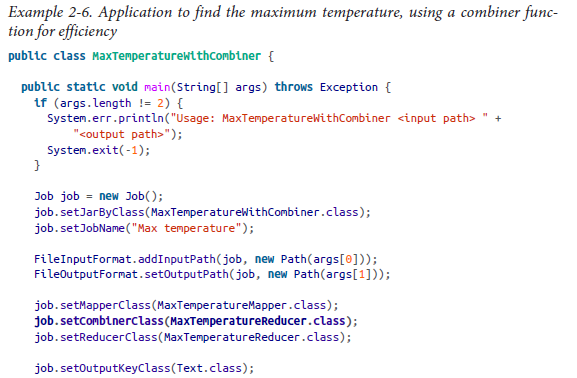
function is still needed to process records with the same key from different maps.) But

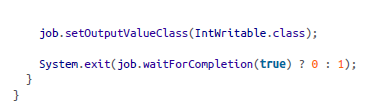
it can help cut down the amount of data shuffled between the mappers and the reducers,

and for this reason alone it is always worth considering whether you can use a combiner

function in your MapReduce job.

Specifying a combiner function





Hadoop Streaming

Hadoop provides an API to MapReduce that allows you to write your map and reduce

functions in languages other than Java. *Hadoop Streaming* uses Unix standard streams

as the interface between Hadoop and your program, so you can use any language that

can read standard input and write to standard output to write your MapReduce

program.

# The Hadoop Distributed Filesystem

The Design of HDFS

HDFS is a filesystem designed for storing very large files with streaming data access

patterns, running on clusters of commodity hardware.

*Very large files*

“Very large” in this context means files that are hundreds of megabytes, gigabytes,

or terabytes in size. There are Hadoop clusters running today that store petabytes

of data.2

*Streaming data access*

HDFS is built around the idea that the most efficient data processing pattern is a

write-once, read-many-times pattern. A dataset is typically generated or copied

from source, and then various analyses are performed on that dataset over time.

Each analysis will involve a large proportion, if not all, of the dataset, so the time

to read the whole dataset is more important than the latency in reading the first

record.

*Commodity hardware*

Hadoop doesn’t require expensive, highly reliable hardware. It’s designed to run on

clusters of commodity hardware (commonly available hardware that can be obtained

from multiple vendors)3 for which the chance of node failure across the

cluster is high, at least for large clusters. HDFS is designed to carry on working

without a noticeable interruption to the user in the face of such failure.

HDFS is not a good fit for

*Low-latency data access*

Applications that require low-latency access to data, in the tens of milliseconds

range, will not work well with HDFS. Remember, HDFS is optimized for delivering

a high throughput of data, and this may be at the expense of latency. HBase (Chapter

20) is currently a better choice for low-latency access.

*Lots of small files*

Because the namenode holds filesystem metadata in memory, the limit to the number

of files in a filesystem is governed by the amount of memory on the namenode.

As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for

example, if you had one million files, each taking one block, you would need at least

300 MB of memory. Although storing millions of files is feasible, billions is beyond

the capability of current hardware.

*Multiple writers, arbitrary file modifications*

Files in HDFS may be written to by a single writer. Writes are always made at the

end of the file, in append-only fashion. There is no support for multiple writers or

for modifications at arbitrary offsets in the file.

Blocks

HDFS, too, has the concept of a block, but it is a much larger unit—128 MB by default.

Having a block abstraction for a distributed filesystem brings several benefits. The first benefit is the most obvious: a file can be larger than any single disk in the network.

There’s nothing that requires the blocks from a file to be stored on the same disk, so

they can take advantage of any of the disks in the cluster. In fact, it would be possible,

if unusual, to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.

Having a block abstraction for a distributed filesystem brings several benefits. The first benefit is the most obvious: a file can be larger than any single disk in the network.

There’s nothing that requires the blocks from a file to be stored on the same disk, so

they can take advantage of any of the disks in the cluster. In fact, it would be possible,

if unusual, to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.

Second, making the unit of abstraction a block rather than a file simplifies the storage

subsystem. Simplicity is something to strive for in all systems, but is especially important for a distributed system in which the failure modes are so varied. The storage subsystem deals with blocks, simplifying storage management (because blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns (because blocks are just a chunk of data to be stored, file metadata such as permissions information does not need to be stored with the blocks, so another system can handle metadata separately).

Furthermore, blocks fit well with replication for providing fault tolerance and availability.

To insure against corrupted blocks and disk and machine failure, each block is

replicated to a small number of physically separate machines (typically three). If a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client. A block that is no longer available due to corruption or machine failure can be replicated from its alternative locations to other live machines to bring the replication factor back to the normal level.

Namenodes and Datanodes

An HDFS cluster has two types of nodes operating in a master-worker pattern: a *namenode*

(the master) and a number of *datanodes* (workers). The namenode manages the

filesystem namespace. It maintains the filesystem tree and the metadata for all the files

and directories in the tree. This information is stored persistently on the local disk in

the form of two files: the namespace image and the edit log. The namenode also knows

the datanodes on which all the blocks for a given file are located; however, it does

not store block locations persistently, because this information is reconstructed from

datanodes when the system starts.

A *client* accesses the filesystem on behalf of the user by communicating with the namenode

and datanodes. The client presents a filesystem interface similar to a Portable

Operating System Interface (POSIX), so the user code does not need to know about the

namenode and datanode to function.

Datanodes are the workhorses of the filesystem. They store and retrieve blocks when

they are told to (by clients or the namenode), and they report back to the namenode

periodically with lists of blocks that they are storing.

Without the namenode, the filesystem cannot be used. In fact, if the machine running

the namenode were obliterated, all the files on the filesystem would be lost since there

would be no way of knowing how to reconstruct the files from the blocks on the

datanodes. For this reason, it is important to make the namenode resilient to failure,

and Hadoop provides two mechanisms for this.

The first way is to back up the files that make up the persistent state of the filesystem

metadata. Hadoop can be configured so that the namenode writes its persistent state to

multiple filesystems. These writes are synchronous and atomic. The usual configuration

choice is to write to local disk as well as a remote NFS mount.

It is also possible to run a *secondary namenode*, which despite its name does not act as

a namenode. Its main role is to periodically merge the namespace image with the edit

log to prevent the edit log from becoming too large. The secondary namenode usually

runs on a separate physical machine because it requires plenty of CPU and as much

memory as the namenode to perform the merge. It keeps a copy of the merged namespace

image, which can be used in the event of the namenode failing. However, the state

of the secondary namenode lags that of the primary, so in the event of total failure of

the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode’s metadata files that are on NFS to the secondary and run it as the new primary.

Block Caching

Normally a datanode reads blocks from disk, but for frequently-accessed files the blocks

may be explicitly cached in the datanode’s memory, in an off-heap *block cache*. By default a block is cached in only one datanode’s memory, although the number is configurable on a per-file basis. Job schedulers (for MapReduce, Spark and other frameworks) can take advantage of cached blocks by running tasks on the datanode where a block is cached, for increased read performance. A small lookup table used in a join is a good candidate for caching, for example.

Users or applications instruct the namenode which files to cache (and for how long) by

adding a *cache directive* to a *cache pool*. Cache pools are an administrative grouping for

managing cache permissions and resource usage.

HDFS Federation

The namenode keeps a reference to every file and block in the filesystem in memory,

which means that on very large clusters with many files, memory becomes the limiting

factor for scaling (see “How Much Memory Does a Namenode Need?” on page 296). HDFS

Federation, introduced in the 2.x release series, allows a cluster to scale by adding

namenodes, each of which manages a portion of the filesystem namespace. For example,

one namenode might manage all the files rooted under */user*, say, and a second namenode

might handle files under */share*.

Under federation, each namenode manages a *namespace volume*, which is made up of

the metadata for the namespace, and a *block pool* containing all the blocks for the files

in the namespace. Namespace volumes are independent of each other, which means

namenodes do not communicate with one another, and furthermore the failure of one

namenode does not affect the availability of the namespaces managed by other namenodes.

Block pool storage is *not* partitioned, however, so datanodes register with each

namenode in the cluster and store blocks from multiple block pools.

To access a federated HDFS cluster, clients use client-side mount tables to map file paths to namenodes. This is managed in configuration using ViewFileSystem and the

viewfs:// URIs.

HDFS High-Availability

The combination of replicating namenode metadata on multiple filesystems and using

the secondary namenode to create checkpoints protects against data loss, but it does

not provide high-availability of the filesystem. The namenode is still a *single point of*

*failure* (SPOF). If it did fail, all clients—including MapReduce jobs—would be unable

to read, write, or list files, because the namenode is the sole repository of the metadata

and the file-to-block mapping. In such an event the whole Hadoop system would effectively

be out of service until a new namenode could be brought online.

To recover from a failed namenode in this situation, an administrator starts a new primary

namenode with one of the filesystem metadata replicas and configures datanodes

and clients to use this new namenode. The new namenode is not able to serve requests

until it has i) loaded its namespace image into memory,

ii) replayed its edit log, and

iii) received enough block reports from the datanodes to leave safe mode. On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.

The long recovery time is a problem for routine maintenance too. In fact, because unexpected failure of the namenode is so rare, the case for planned downtime is actually

more important in practice.

Hadoop 2 remedied this situation by adding support for HDFS high-availability (HA).

In this implementation there is a pair of namenodes in an active-standby configuration.

In the event of the failure of the active namenode, the standby takes over its duties to

continue servicing client requests without a significant interruption. A few architectural changes are needed to allow this to happen:

• The namenodes must use highly-available shared storage to share the edit log.

When a standby namenode comes up, it reads up to the end of the shared edit log

to synchronize its state with the active namenode, and then continues to read new

entries as they are written by the active namenode.

• Datanodes must send block reports to both namenodes because the block mappings

are stored in a namenode’s memory, and not on disk.

• Clients must be configured to handle namenode failover, using a mechanism that

is transparent to users.

• The secondary namenode’s role is subsumed by the standby, which takes periodic

checkpoints of the active namenode’s namespace.

There are two choices for the highly-available shared storage: an NFS filer, or a *quorum*

*journal manager* (QJM). The QJM is a dedicated HDFS implementation, designed for

the sole purpose of providing a highly-available edit log, and is the recommended choice

for most HDFS installations. The QJM runs as a group of *journal nodes*, and each edit

must be written to a majority of the journal nodes. Typically, there are three journal

nodes, so the system can tolerate the loss of one of them. This arrangement is similar

to the way ZooKeeper works, although it is important to realize that the QJM implementation does not use ZooKeeper. (Note, however, that HDFS HA *does* use ZooKeeper

for electing the active namenode; see below.)

If the active namenode fails, the standby can take over very quickly (in a few tens of

seconds) because it has the latest state available in memory: both the latest edit log entries and an up-to-date block mapping. The actual observed failover time will be longer in practice (around a minute or so), because the system needs to be conservative in deciding that the active namenode has failed.

In the unlikely event of the standby being down when the active fails, the administrator

can still start the standby from cold. This is no worse than the non-HA case, and from

an operational point of view it’s an improvement, because the process is a standard

operational procedure built into Hadoop.

Failover and fencing

The transition from the active namenode to the standby is managed by a new entity in

the system called the *failover controller*. There are various failover controllers, but the

default implementation uses ZooKeeper to ensure that only one namenode is active.

Each namenode runs a lightweight failover controller process whose job it is to monitor

its namenode for failures (using a simple heartbeating mechanism) and trigger a failover

should a namenode fail.

In the case of an ungraceful failover, however, it is impossible to be sure that the failed namenode has stopped running. For example, a slow network or a network partition

can trigger a failover transition, even though the previously active namenode is still

running and thinks it is still the active namenode. The HA implementation goes to great

lengths to ensure that the previously active namenode is prevented from doing any

damage and causing corruption—a method known as *fencing*.

The Command-Line Interface

copying a file from the local filesystem to HDFS:

% **hadoop fs -copyFromLocal input/docs/quangle.txt \**

**hdfs://localhost/user/tom/quangle.txt**

Let’s copy the file back to the local filesystem and check whether it’s the same:

% **hadoop fs -copyToLocal quangle.txt quangle.copy.txt**

% **hadoop fs -mkdir books**

File Permissions in HDFS

By default, Hadoop runs with security disabled, which means that a client’s identity is

not authenticated. Because clients are remote, it is possible for a client to become an

arbitrary user simply by creating an account of that name on the remote system. This

is not possible if security is turned on; see “Security” on page 310. Either way, it is worthwhile having permissions enabled (as it is by default; see the dfs.permissions.en

abled property), to avoid accidental modification or deletion of substantial parts of the

filesystem, either by users or by automated tools or programs.

Interfaces

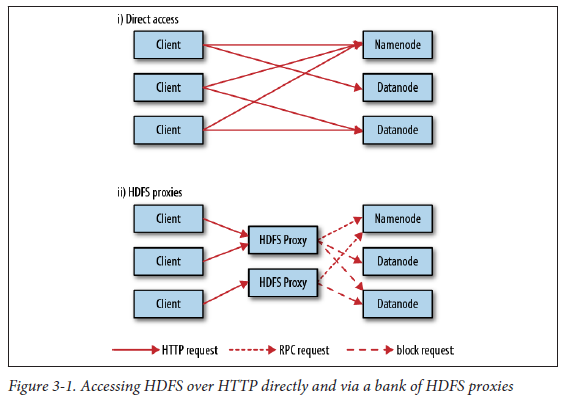
HTTP

By exposing its filesystem interface as a Java API, Hadoop makes it awkward for non-

Java applications to access HDFS. The HTTP REST API exposed by the WebHDFS

protocol makes it easier for other languages to interact with HDFS. Note that the HTTP

interface is slower than the native Java cliet.



In the first case, the embedded web servers in the namenode and datanodes act as

WebHDFS endpoints. (WebHDFS is enabled by default, since dfs.webhdfs.enabled is

set to true.) File metadata operations are handled by the namenode, while file read (and

write) operations are sent first to the namenode, which sends an HTTP redirect to the

client indicating the datanode to stream file data from (or to).

The second way of accessing HDFS over HTTP relies on one or more standalone proxy

servers. (The proxies are stateless so they can run behind a standard load balancer.) All

traffic to the cluster passes through the proxy, so the client never accesses the namenode or datanode directly. This allows for stricter firewall and bandwidth-limiting policies to be put in place. It’s common to use a proxy for transfers between Hadoop clusters located in different data centers, or when accessing a Hadoop cluster running in the cloud from an external network.

C

Hadoop provides a C library called *libhdfs* that mirrors the Java FileSystem interface

(it was written as a C library for accessing HDFS, but despite its name it can be used to

Hadoop Filesystems | 55

6. In Hadoop 2 and later there is a new filesystem interface called FileContext with better handling of multiple

filesystems (so a single FileContext can resolve multiple filesystem schemes, for example) and a cleaner,

more consistent interface. FileSystem is still more widely used, however.

access any Hadoop filesystem). It works using the *Java Native Interface* (JNI) to call a

Java filesystem client. There is also a *libwebhdfs* library that uses the WebHDFS interface

described in the previous section.

NFS

It is possible to mount HDFS on a local client’s filesystem using Hadoop’s NFSv3 gateway.

You can then use Unix utilities (such as ls and cat) to interact with the filesystem,

upload files, and in general use POSIX libraries to access the filesystem from any programming

language. Appending to a file works, but random modifications of a file do

not, since HDFS can only write to the end of a file.

FUSE

*Filesystem in Userspace* (FUSE) allows filesystems that are implemented in user space

to be integrated as a Unix filesystem. Hadoop’s Fuse-DFS contrib module allows HDFS

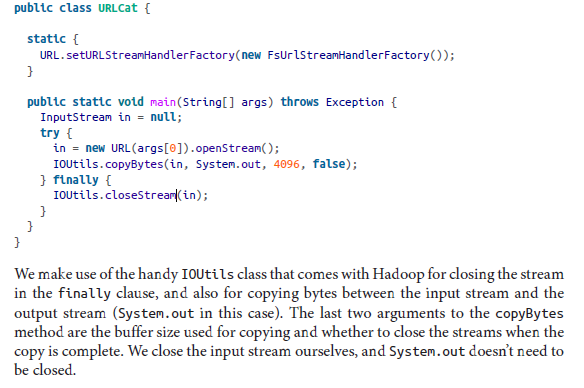
(or any Hadoop filesystem) to be mounted as a standard local filesystem. Fuse-DFS is

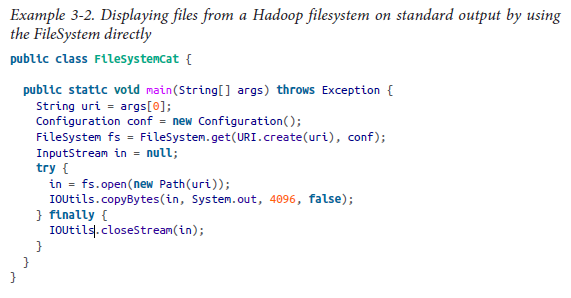
implemented in C using *libhdfs* as the interface to HDFS. At the time of writing, the

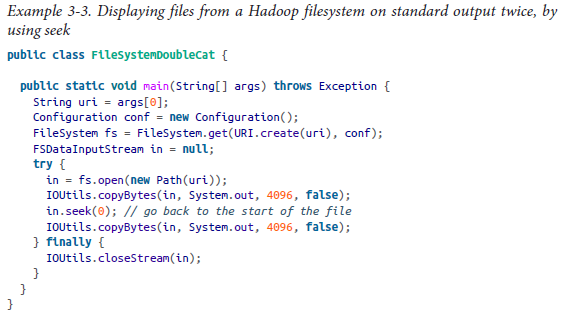
Hadoop NFS gateway is the more robust solution to mounting HDFS, so should be

preferred over Fuse-DFS.

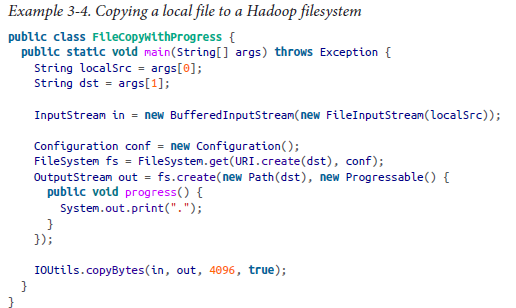
The Java Interface







Writing Data



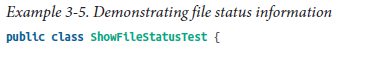
Directories

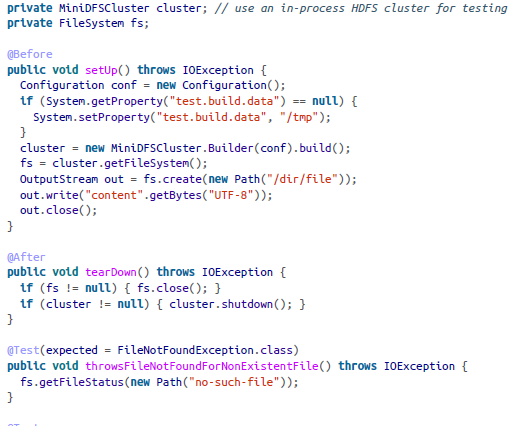
FileSystem provides a method to create a directory:

**public boolean** mkdirs(Path f) **throws** IOException

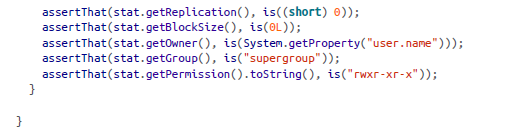
Querying the Filesystem

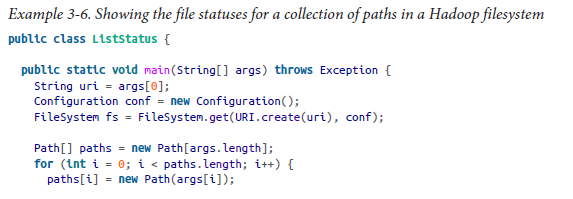
File metadata: FileStatus

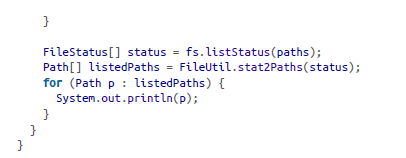












File patterns

The filter passes only those files that *don’t* match the regular expression. After the glob

picks out an initial set of files to include, the filter is used to refine the results. For

example:

fs.globStatus(**new** Path("/2007/\*/\*"), **new** RegexExcludeFilter("^.\*/2007/12/31$"))

will expand to */2007/12/30*.

Deleting Data

Use the delete() method on FileSystem to permanently remove files or directories:

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

Data Flow

Anatomy of a File Read

The client opens the file it wishes to read by calling open() on the FileSystem object,

which for HDFS is an instance of DistributedFileSystem (step 1 in Figure 3-2).

DistributedFileSystem calls the namenode, using RPC, to determine the locations of

the blocks for the first few blocks in the file (step 2). For each block, the namenode

returns the addresses of the datanodes that have a copy of that block. Furthermore, the

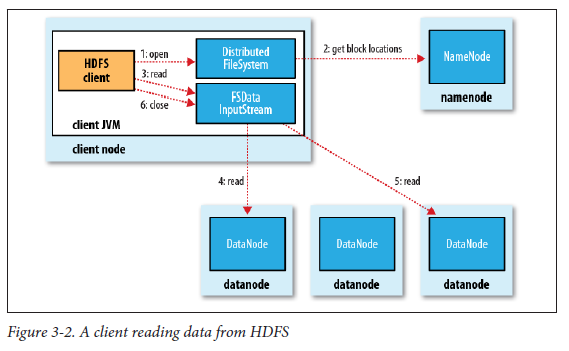
datanodes are sorted according to their proximity to the client (according to the topology

of the cluster’s network; see “Network Topology and Hadoop” on page 70). If the

client is itself a datanode (in the case of a MapReduce task, for instance), the client will

read from the local datanode if that datanode hosts a copy of the block (see also

Figure 2-2 and “Short-circuit local reads” on page 310).



The DistributedFileSystem returns an FSDataInputStream (an input stream that

supports file seeks) to the client for it to read data from. FSDataInputStream in turn

wraps a DFSInputStream, which manages the datanode and namenode I/O.

The client then calls read() on the stream (step 3). DFSInputStream, which has stored

the datanode addresses for the first few blocks in the file, then connects to the first

(closest) datanode for the first block in the file. Data is streamed from the datanode back

to the client, which calls read() repeatedly on the stream (step 4). When the end of the

block is reached, DFSInputStream will close the connection to the datanode, then find

the best datanode for the next block (step 5). This happens transparently to the client,

which from its point of view is just reading a continuous stream.

Blocks are read in order, with the DFSInputStream opening new connections to datanodes

as the client reads through the stream. It will also call the namenode to retrieve the

datanode locations for the next batch of blocks as needed. When the client has finished

reading, it calls close() on the FSDataInputStream (step 6).

During reading, if the DFSInputStream encounters an error while communicating with

a datanode, it will try the next closest one for that block. It will also remember datanodes

that have failed so that it doesn’t needlessly retry them for later blocks. The DFSInput

Stream also verifies checksums for the data transferred to it from the datanode. If a

corrupted block is found, the DFSInputStream attempts to read a replica of the block

from another datanode; it also reports the corrupted block to the namenode.

One important aspect of this design is that the client contacts datanodes directly to

retrieve data and is guided by the namenode to the best datanode for each block. This

design allows HDFS to scale to a large number of concurrent clients because the data

traffic is spread across all the datanodes in the cluster. Meanwhile, the namenode merely

has to service block location requests (which it stores in memory, making them very

efficient) and does not, for example, serve data, which would quickly become a bottleneck

as the number of clients grew.

Network Topology and Hadoop

• Processes on the same node

• Different nodes on the same rack

• Nodes on different racks in the same data center

• Nodes in different data centers8

Anatomy of a File Write

The client creates the file by calling create() on DistributedFileSystem (step 1 in

Figure 3-4). DistributedFileSystem makes an RPC call to the namenode to create a

new file in the filesystem’s namespace, with no blocks associated with it (step 2). The

namenode performs various checks to make sure the file doesn’t already exist and that

the client has the right permissions to create the file. If these checks pass, the namenode

makes a record of the new file; otherwise, file creation fails and the client is thrown an

IOException. The DistributedFileSystem returns an FSDataOutputStream for the

client to start writing data to. Just as in the read case, FSDataOutputStream wraps a

DFSOutputStream, which handles communication with the datanodes and namenode.

As the client writes data (step 3), DFSOutputStream splits it into packets, which it writes

to an internal queue, called the *data queue*. The data queue is consumed by the Data

Streamer, which is responsible for asking the namenode to allocate new blocks by

picking a list of suitable datanodes to store the replicas. The list of datanodes forms a

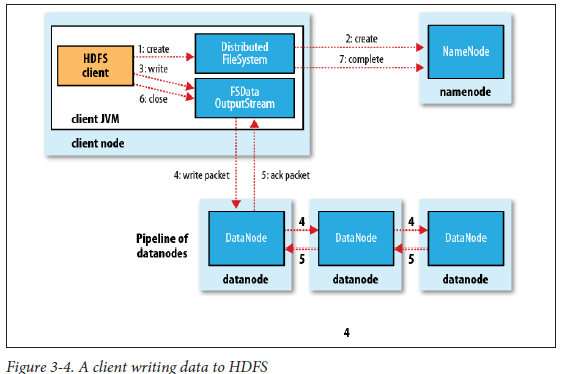
pipeline, and here we’ll assume the replication level is three, so there are three nodes in

the pipeline. The DataStreamer streams the packets to the first datanode in the pipeline,

which stores the packet and forwards it to the second datanode in the pipeline. Similarly,

the second datanode stores the packet and forwards it to the third (and last) datanode

in the pipeline (step 4).



DFSOutputStream also maintains an internal queue of packets that are waiting to be

acknowledged by datanodes, called the *ack queue*. A packet is removed from the ack

queue only when it has been acknowledged by all the datanodes in the pipeline (step 5).

If any datanode fails while data is being written to it, then the following actions are

taken, which are transparent to the client writing the data. First, the pipeline is closed,

and any packets in the ack queue are added to the front of the data queue so that datanodes

that are downstream from the failed node will not miss any packets. The current

block on the good datanodes is given a new identity, which is communicated to the

namenode, so that the partial block on the failed datanode will be deleted if the failed

datanode recovers later on. The failed datanode is removed from the pipeline, and a

new pipeline is constructed from the two good datanodes. The remainder of the block’s

data is written to the good datanodes in the pipeline. The namenode notices that the

block is under-replicated, and it arranges for a further replica to be created on another

node. Subsequent blocks are then treated as normal.

It’s possible, but unlikely, that multiple datanodes fail while a block is being written. As

long as dfs.namenode.replication.min replicas (which defaults to one) are written,

the write will succeed, and the block will be asynchronously replicated across the cluster

until its target replication factor is reached (dfs.replication, which defaults to three).

When the client has finished writing data, it calls close() on the stream (step 6). This

action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments

before contacting the namenode to signal that the file is complete (step

7). The namenode already knows which blocks the file is made up of (via DataStream

er asking for block allocations), so it only has to wait for blocks to be minimally replicated

before returning successfully.

Replica Placement

Hadoop’s default strategy is to place the first replica on the same node as the client (for

clients running outside the cluster, a node is chosen at random, although the system

tries not to pick nodes that are too full or too busy). The second replica is placed on a

different rack from the first (*off-rack*), chosen at random. The third replica is placed on

the same rack as the second, but on a different node chosen at random. Further replicas are placed on random nodes on the cluster, although the system tries to avoid placing too many replicas on the same rack.

Once the replica locations have been chosen, a pipeline is built, taking network topology

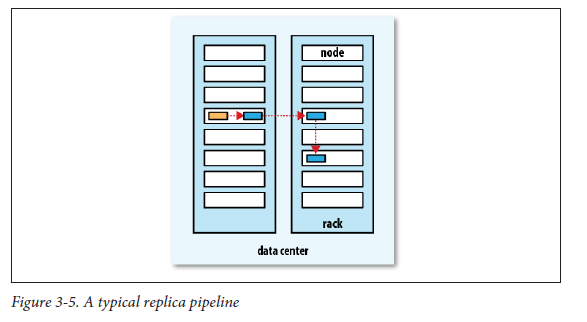
into account. For a replication factor of 3, the pipeline might look like Figure 3-5.

Overall, this strategy gives a good balance among reliability (blocks are stored on two

racks), write bandwidth (writes only have to traverse a single network switch), read

performance (there’s a choice of two racks to read from), and block distribution across

the cluster (clients only write a single block on the local rack).



Coherency Model

A coherency model for a filesystem describes the data visibility of reads and writes for

a file.

HDFS provides a method for forcing all buffers to be flushed to the datanodes via the

hflush() method on FSDataOutputStream. After a successful return from hflush(),

HDFS guarantees that the data written up to that point in the file has reached all the

datanodes in the write pipeline and is visible to all new readers:

Path p = **new** Path("p");

FSDataOutputStream out = fs.create(p);

out.write("content".getBytes("UTF-8"));

**out.hflush();**

assertThat(fs.getFileStatus(p).getLen(), is(((**long**) "content".length())));

Note that hflush() does not guarantee that the datanodes have written the data to disk,

only that it’s in the datanodes’ memory (so in the event of a data center power outage,

for example, data could be lost). For this stronger guarantee, use hsync() instead.9

The behavior of hsync() is similar to the fsync system call in POSIX that commits

buffered data for a file descriptor. For example, using the standard Java API to write a

local file, we are guaranteed to see the content after flushing the stream and synchronizing:

FileOutputStream out = **new** FileOutputStream(localFile);

out.write("content".getBytes("UTF-8"));

out.flush(); *// flush to operating system*

out.getFD().sync(); *// sync to disk*

assertThat(localFile.length(), is(((**long**) "content".length())));

Closing a file in HDFS performs an implicit hflush(), too:

Path p = **new** Path("p");

OutputStream out = fs.create(p);

out.write("content".getBytes("UTF-8"));

**out.close();**

assertThat(fs.getFileStatus(p).getLen(), is(((**long**) "content".length())));

Parallel Copying with distcp

One use for *distcp* is as an efficient replacement for hadoop fs -cp. For example, you

can copy one file to another with:10

% **hadoop distcp file1 file2**

You can also copy directories:

% **hadoop distcp dir1 dir2**

% **hadoop distcp -update dir1 dir2**

A very common use case for *distcp* is for transferring data between two HDFS clusters.

For example, the following creates a backup of the first cluster’s */foo* directory on the

second:

% **hadoop distcp -update -delete -p hdfs://namenode1/foo hdfs://namenode2/foo**

Keeping an HDFS Cluster Balanced

When copying data into HDFS, it’s important to consider cluster balance. HDFS works

best when the file blocks are evenly spread across the cluster, so you want to ensure that

*distcp* doesn’t disrupt this. For example, by specifying -m 1, a single map would do the

copy, which—apart from being slow and not using the cluster resources efficiently—

would mean that the first replica of each block would reside on the node running the

map (until the disk filled up). The second and third replicas would be spread across the

cluster, but this one node would be unbalanced. By having more maps than nodes in

the cluster, this problem is avoided. For this reason, it’s best to start by running *distcp*

with the default of 20 maps per node.

YARN

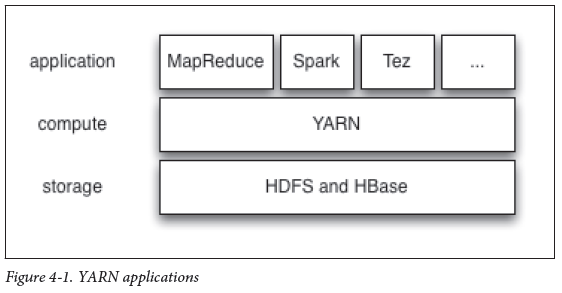
Apache YARN (Yet Another Resource Negotiator) is Hadoop’s cluster resource management

system.

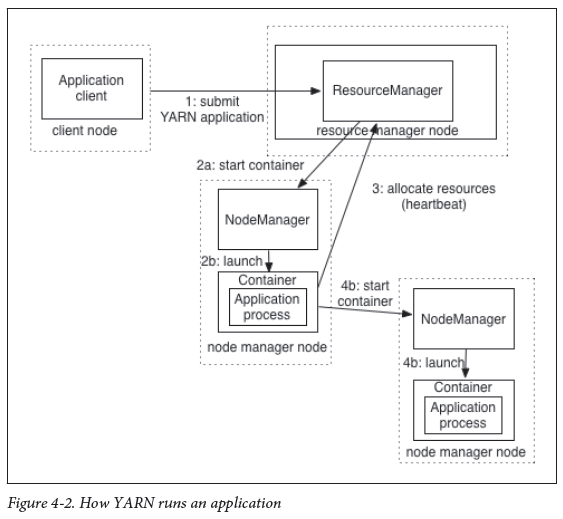
Figure 4-1, which shows some distributed computing frameworks (MapReduce, Spark,

and so on) running as *YARN applications* on the cluster compute layer (YARN) and the

cluster storage layer (HDFS and HBase).



Anatomy of a YARN Application Run



YARN provides its core services via two types of long-running daemon: a *resource*

*manager* (one per cluster) to manage the use of resources across the cluster, and *node*

*managers* running on all the nodes in the cluster to launch and monitor *containers*. A

container executes an application-specific process with a constrained set of resources

(memory, CPU, and so on). Depending on how YARN is configured (see “YARN” on

page 302), a container may be a Unix process, or a Linux cgroup.

To run an application on YARN, a client contacts the resource manager and asks it to

run an *application master* process (step 1 in Figure 4-2). The resource manager then

finds a node manager that can launch the application master in a container (step 2a and

2b).1 Precisely what the application master does once it is running depends on the application.

It could simply run a computation in the container it is running in and return

the result to the client. Or it could request more containers from the resource managers

(step 3), and use them to run a distributed computation (step 4a and 4b). The latter is

what the MapReduce YARN application does, which we’ll look at in more detail in

“Anatomy of a MapReduce Job Run” on page 187.

Notice from Figure 4-2 that YARN itself does not provide any way for the parts of the

application (client, master, process) to communicate with one another. Most non-trivial

YARN applications use some form of remote communication (such as Hadoop’s RPC

layer) to pass status updates and results back to the client, but these are specific to the application.

Resource Requests

YARN has a flexible model for making resource requests. A request for a set of containers

can express the amount of computer resources required for each container (memory

and CPU), as well as locality constraints for the containers in that request.

Locality is critical in ensuring that distributed data processing algorithms use the cluster bandwidth efficiently2, so YARN allows an application to specify locality constraints for the containers it is requesting. Locality constraints can be used to request a container on a specific node or rack, or on a specific rack, or anywhere on the cluster (off-rack).

Sometimes the locality constraint cannot be met, in which case either no allocation is

made or, optionally, the constraint can be loosened. For example, if a specific node was

requested and it is not possible to start a container on it (because other containers are

running on it), then YARN would try to start a container on a node in the same rack,

or if that’s not possible, on any node in the cluster.

In the common case of launching a container to process an HDFS block (to run a map

task in MapReduce, say), the application would request a container on one of the nodes

hosting the block’s three replicas, or on a node in one of the racks hosting the replicas, or, failing that, on any node in the cluster.

A YARN application can make resource requests at any time while it is running. For

example, an application can make all of its requests up front, or it can take a more

dynamic approach whereby it requests more resources dynamically to meet the changing

needs of the application.

Spark takes the first approach, starting a fixed number of executors on the cluster (see

“Spark on YARN” on page 573). MapReduce, on the other hand, has two phases: the map

task containers are requested up front, and the reduce task containers are not started

until later. Also, if any tasks fail then additional containers will be requested so the failed tasks can be rerun.

Application Lifespan

The lifespan of a YARN application can vary dramatically: from a short-lived application

of a few seconds to a long-running application that runs for days or even months. Rather

than look at how long the application runs for, it’s useful to categorize applications in terms of how they map to the jobs that users run. The simplest case is one application

per user job, which is the approach that MapReduce takes.

The second model is to run one application per workflow or user session of (possibly

unrelated) jobs. This approach can be more efficient than the first, since containers can

be reused between jobs, and there is also the potential to cache intermediate data between jobs. Spark is an example that uses this model.

The third model is a long-running application that is shared by different users. Such an

application often acts in some kind of coordination role. For example, Apache Slider

has a long-running application master for launching other applications on the cluster.

This approach is also used by Impala (see “SQL-on-Hadoop Alternatives” on page 485) to

provide a proxy application that the Impala daemons communicate with to request

cluster resources. The “always on” application master means that users have very lowlatency responses to their queries since the overhead of starting a new application master is avoided.3

Building YARN Applications

Writing a YARN application from scratch is fairly involved, but in many cases is not

necessary since an existing application fits the bill. For example, if you are interested in running a directed acyclic graph (DAG) of jobs then Spark or Tez are appropriate; or

for stream processing one of Spark, Samza or Storm would be applicable.4

There are a couple of projects that simplify the process of building a YARN application.

Apache Slider, mentioned earlier, makes it possible to run existing distributed applications on YARN. Users can run their own instance of an application (such as HBase) on a cluster, independently of other users, which means that different users can run different versions of the same application. Slider provides controls to change the number

of nodes an application is running on, and to suspend then resume a running application.

Apache Twill is similar to Slider but in addition provides a simple programming model

for developing distributed applications on YARN. Twill allows you to define cluster

processes as an extension of a Java Runnable, then runs them in YARN containers on

the cluster. Twill also provides support for, amongst other things, real-time logging (log

events from runnables are streamed back to the client) and command messages (sent

from the client to runnables).

In cases where none of these options are sufficient—such as an application that has

complex scheduling requirements—then the *distributed shell* application that is a part

of the YARN project itself serves as an example of how to write a YARN application. It

demonstrates how to use YARN’s client APIs to handle communication between the

client or application master and the YARN daemons.

YARN Compared to MapReduce 1

In MapReduce 1, there are two types of daemon that control the job execution process:

a *jobtracker* and one or more *tasktrackers*. The jobtracker coordinates all the jobs run

on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and

send progress reports to the jobtracker, which keeps a record of the overall progress of

each job. If a task fails, the jobtracker can reschedule it on a different tasktracker.

In MapReduce 1, the jobtracker takes care of both job scheduling (matching tasks with

tasktrackers) and task progress monitoring (keeping track of tasks, restarting failed or

slow tasks, and doing task bookkeeping, such as maintaining counter totals). By contrast,

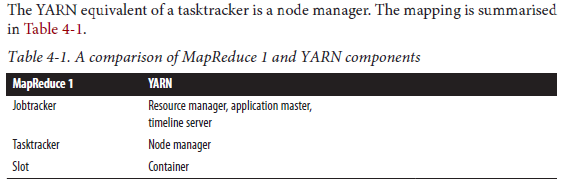
in YARN these responsibilities are split into separate entities: the resource manager

and an application master (one for each MapReduce job). The jobtracker is also responsible

for storing job history for completed jobs, although it is possible to run a job

history server as a separate daemon to take the load off the jobtracker. In YARN, the

equivalent role is the timeline server, which stores application history.5



*Scalability*

YARN can run on larger clusters than MapReduce 1. MapReduce 1 hits scalability

bottlenecks in the region of 4,000 nodes, and 40,000 tasks6, which stem from the

fact that the jobtracker has to manage both jobs *and* tasks. YARN overcomes these

limitations by virtue of its split resource manager/application master architecture,

which means it is designed to scale up to 10,000 nodes, and 100,000 tasks.

In contrast to the jobtracker, each instance of an application—here a MapReduce

job—has a dedicated application master, which runs for the duration of the application.

This model is actually closer to the original Google MapReduce paper, which

describes how a master process is started to coordinate map and reduce tasks running

on a set of workers.

*Availability*

High availability (HA) is usually achieved by replicating the state needed for another

daemon to take over the work needed to provide the service, in the event of the

service daemon failing. However, the large amount of rapidly changing complex

state in the jobtracker’s memory (each task status is updated every few seconds, for

example) makes it very difficult to retrofit HA into the jobtracker service.

With the jobtracker’s reponsibilities split between the resource manager and application

master in YARN, making the service highly-available became a divideand-

conquer problem: provide HA for the resource manager, then for YARN applications

(on a per-application basis). And indeed Hadoop 2 supports HA for both

the resource manager, and for the application master for MapReduce jobs. Failure

recovery in YARN is discussed in more detail in “Failures” on page 195.

*Utilization*

In MapReduce 1, each tasktracker is configured with a static allocation of fixed size

“slots,” which are divided into map slots and reduce slots at configuration time. A

map slot can only be used to run a map task, and a reduce slot can only be used for

a reduce task.

In YARN, a node manager manages a pool of resources, rather than a fixed number

of designated slots. MapReduce running on YARN will not hit the situation where

a reduce task has to wait because only map slots are available on the cluster, which

is the case in MapReduce 1. If the resources to run the task are available, then the

application will be eligible for them.

Furthermore, resources in YARN are fine-grained, so an application can make a

request for what it needs, rather than for an indivisible slot, which may be too big

(which is wasteful of resources) or too small (which may cause a failure) for the

particular task.

*Multitenancy*

In some ways, the biggest benefit of YARN is that it opens up Hadoop to other types

of distributed application beyond MapReduce. MapReduce is just one YARN application

amongst many.

It is even possible for users to run different versions of MapReduce on the same

YARN cluster, which makes the process of upgrading MapReduce more manageable.

(Note, however, that some parts of MapReduce, such as the job history server

and the shuffle handler, as well as YARN itself, still need to be upgraded across the

cluster.)

Scheduling in YARN

The FIFO Scheduler

The FIFO Scheduler places applications in a queue and runs them in the order of submission.

Requests for the first application in the queue are allocated first, then once its

requests have been satisfied the next application in the queue is served, and so on.

The FIFO Scheduler has the merit of being simple to understand and not needing any

configuration, but it’s not suitable for shared clusters. Large applications will use all the

resources in a cluster, so each application has to wait its turn. On a shared cluster it is

better to use the Capacity Scheduler or the Fair Scheduler. Both of these allow longrunning

jobs to complete in a timely manner, while still allowing users who are running

concurrent smaller ad hoc queries to get results back in a reasonable time.

The difference between schedulers is illustrated in Figure 4-3, which shows that under

the FIFO Scheduler (i.) the small job is blocked until the large job completes.

With the Capacity Scheduler (ii. in Figure 4-3), a separate dedicated queue allows the

small job to start as soon as it is submitted, although this is at the cost of overall cluster utilization since the queue capacity is reserved for jobs in that queue. This means that the large job finishes later than when using the FIFO Scheduler.

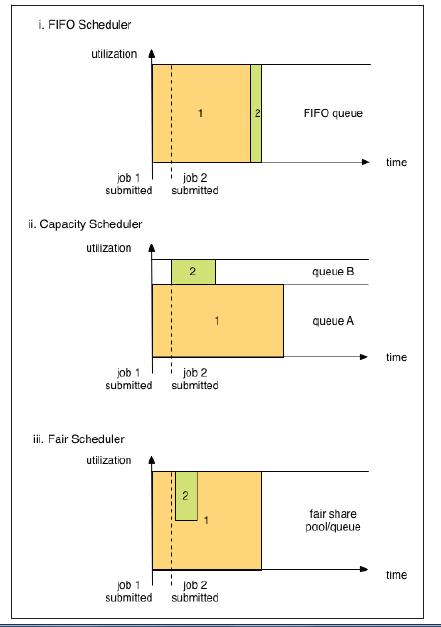
With the Fair Scheduler (iii. in Figure 4-3) there is no need to reserve a set amount of

capacity since it will dynamically balance resources between all running jobs.

Just after the first (large) job starts it is the only job running so it gets all the resources in the cluster. When the second (small) job starts it is allocated half of the cluster resources so that each job is using its fair share of resources.

Note that there is a lag between the time the second job starts and when it receives its

fair share, since it has to wait for resources to free up as containers used by the first job complete. After the small job completes and no longer requires resources, the large job goes back to using the full cluster capacity again. The overall effect is both high cluster utilization and timely small job completion.



The Capacity Scheduler

The Capacity Scheduler allows sharing of a Hadoop cluster along organisational lines,

whereby each organisation is allocated a certain capacity of the overall cluster. Each

organisation is set up with a dedicated queue that is configured to use a given fraction

of the cluster capacity. Queues may be further divided in hierarchical fashion, allowing

organisations to share their cluster allowance between different groups of users within

the organisation. Within a queue applications are scheduled using FIFO scheduling.

As we saw in Figure 4-3, a single job does not use more resources than its queue’s

capacity. However, if there is more than one job in the queue and there are idle resources

available, then the Capacity Scheduler may allocate the spare resources to jobs in the

queue even if that causes the queue’s capacity to be exceeded.7 This behavior is known

as *queue elasticity*.

In normal operation the Capacity Scheduler does not preempt containers by forcibly

killing them8, so if a queue is under capacity due to lack of demand, and then demand

increases, the queue will only return to capacity as resources are released from other

queues as containers complete. It is possible to mitigate this by configuring queues with

a maximum capacity so that they don’t eat into other queues’ capacities too much. This

is at the cost of queue elasticity, of course, so a reasonable trade-off should be found by

trial and error.

Queue placement

The way that you specify which queue an application is placed in is specific to the

application. For example, in MapReduce you set the property mapreduce.job.queue

name to the name of the queue you want to use. If the queue does not exist then you’ll

get an error at submission time. If no queue is specified, applications will be placed in

a queue called default.

The Fair Scheduler

The Fair Scheduler attempts to allocate resources so that all running applications get

the same share of resources. We saw in Figure 4-3 how fair sharing works for applications

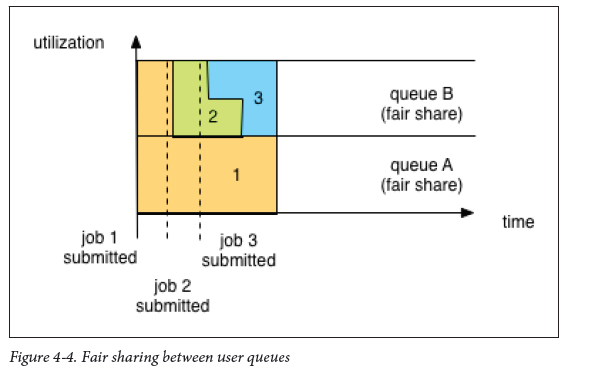
in the same queue; however, fair sharing actually works *between* queues too, as we’ll see

next.

To understand how resources are shared between queues, imagine two users *A* and *B*,

each with their own queue, see Figure 4-4. *A* starts a job and it is allocated all the resources available since there is no demand from *B*. *B* then starts a job while *A*’s job is still running, and after a while each job is using half of the resources, in the way we saw earlier.

Now if *B* starts a second job while the other jobs are still running it will share its resources with *B*’s other job, so each of *B*’s jobs will have one fourth of the resources, while *A*’s will continue to have half. The result is that resources are shared fairly between users.



Enabling the Fair Scheduler

The scheduler in use is determined by the setting of yarn.resourcemanager.schedu

ler.class. The Capacity Scheduler is used by default (although the Fair Scheduler is

the default in some Hadoop distributions, such as CDH), but this can be changed by

setting yarn.resourcemanager.scheduler.class in *yarn-site.xml* to the fullyqualified

classname of the scheduler, org.apache.hadoop.yarn.server.resourceman

ager.scheduler.fair.FairScheduler.

Queue configuration

The Fair Scheduler is configured using an allocation file named *fair-scheduler.xml* that

is loaded from the classpath. (The name can be changed by setting the property

yarn.scheduler.fair.allocation.file.) In the absence of an allocation file the Fair

Scheduler operates as described above: each application is placed in a queue named after

the user and queues are created dynamically when a user submits their first application.

Preemption

When a job is submitted to an empty queue on a busy cluster, the job cannot start until

resources free up from jobs that are already running on the cluster. To make the time

taken for a job to start more predictable, the Fair Scheduler supports *preemption*.

Preemption allows the scheduler to kill containers for queues that are running with

more than their fair share of resources so that the resources can be allocated to a queue

that is under its fair share. Note that preemption reduces overall cluster efficiency, since

the terminated containers need to be re-executed.

Delay Scheduling

All the YARN schedulers try to honor locality requests. On a busy cluster, if an application

requests a particular node then there is a good chance that other containers are

running on it at the time of the request. The obvious course of action is to immediately

loosen the locality requirement and allocate a container on the same rack. However, it

has been observed in practice that waiting a short time (no more than a few seconds)

can dramatically increase the chances of being allocated a container on the requested

node, and therefore increase the efficiency of the cluster. This feature is called *delay*

*scheduling*, and is supported by both the Capacity Scheduler and the Fair Scheduler.

Dominant Resource Fairness

When there is only a single resource type being scheduled, such as memory, then the

concept of capacity or fairness is easy to determine. If two users are running applications

then you can measure the amount of memory that each is using to compare the two

applications. However, when there are multiple resource types in play, things get more

complicated. If one user’s application requires lots of CPU but little memory and the

other’s requires little CPU and lots of memory, how are these two applications compared?

The way that the schedulers in YARN address this problem is to look at each user’s

dominant resource and use it as a measure of the cluster usage, a technique called

*Dominant Resource Fairness*, or DRF for short.9 The idea is best illustrated with a simple

example.

Hadoop I/O

Data Integrity

The usual way of detecting corrupted data is by computing a *checksum* for the data when

it first enters the system, and again whenever it is transmitted across a channel that is

unreliable and hence capable of corrupting the data. The data is deemed to be corrupt

if the newly generated checksum doesn’t exactly match the original. This technique

doesn’t offer any way to fix the data—it is merely error detection. (And this is a reason

for not using low-end hardware; in particular, be sure to use ECC memory.) Note that

it is possible that it’s the checksum that is corrupt, not the data, but this is very unlikely,

because the checksum is much smaller than the data.

A commonly used error-detecting code is CRC-32 (cyclic redundancy check), which

computes a 32-bit integer checksum for input of any size. CRC-32 is used for checksumming

in Hadoop’s ChecksumFileSystem, while HDFS uses the more efficient variant

called CRC-32C.

Data Integrity in HDFS

HDFS transparently checksums all data written to it and by default verifies checksums

when reading data. A separate checksum is created for every dfs.bytes-perchecksum

bytes of data. The default is 512 bytes, and because a CRC-32C checksum is

4 bytes long, the storage overhead is less than 1%.

HDFS transparently checksums all data written to it and by default verifies checksums

when reading data. A separate checksum is created for every dfs.bytes-perchecksum

bytes of data. The default is 512 bytes, and because a CRC-32C checksum is

4 bytes long, the storage overhead is less than 1%.

Datanodes are responsible for verifying the data they receive before storing the data and

its checksum. This applies to data that they receive from clients and from other datanodes

during replication. A client writing data sends it to a pipeline of datanodes (as

explained in Chapter 3), and the last datanode in the pipeline verifies the checksum. If

the datanode detects an error, the client receives an subclass of IOException, which it

should handle in an application-specific manner; for example, by retrying the operation.

When clients read data from datanodes, they verify checksums as well, comparing them

with the ones stored at the datanode. Each datanode keeps a persistent log of checksum

verifications, so it knows the last time each of its blocks was verified. When a client

successfully verifies a block, it tells the datanode, which updates its log. Keeping statistics

such as these is valuable in detecting bad disks.

Aside from block verification on client reads, each datanode runs a DataBlockScan

ner in a background thread that periodically verifies all the blocks stored on the datanode.

This is to guard against corruption due to “bit rot” in the physical storage media.

See “Datanode block scanner” on page 330 for details on how to access the scanner

reports.

Because HDFS stores replicas of blocks, it can “heal” corrupted blocks by copying one

of the good replicas to produce a new, uncorrupt replica. The way this works is that if

a client detects an error when reading a block, it reports the bad block and the datanode

it was trying to read from to the namenode before throwing a ChecksumException. The

namenode marks the block replica as corrupt so it doesn’t direct clients to it or try to

copy this replica to another datanode. It then schedules a copy of the block to be replicated

on another datanode, so its replication factor is back at the expected level. Once

this has happened, the corrupt replica is deleted.

It is possible to disable verification of checksums by passing false to the setVerify

Checksum() method on FileSystem before using the open() method to read a file. The

same effect is possible from the shell by using the -ignoreCrc option with the -get or

the equivalent -copyToLocal command. This feature is useful if you have a corrupt file

that you want to inspect so you can decide what to do with it. For example, you might

want to see whether it can be salvaged before you delete it.

You can find a file’s checksum with hadoop fs -checksum. This is useful to check

whether two files in HDFS have the same contents, something that *distcp* does, for

example

LocalFileSystem

The Hadoop LocalFileSystem performs client-side checksumming. This means that

when you write a file called *filename*, the filesystem client transparently creates a hidden

file, *.filename.crc*, in the same directory containing the checksums for each chunk of the

file. The chunk size is controlled by the file.bytes-per-checksum property, which

defaults to 512 bytes. The chunk size is stored as metadata in the *.crc* file, so the file can

be read back correctly even if the setting for the chunk size has changed. Checksums

are verified when the file is read, and if an error is detected, LocalFileSystem throws

a ChecksumException

Checksums are fairly cheap to compute (in Java, they are implemented in native code),

typically adding a few percent overhead to the time to read or write a file.For most

applications, this is an acceptable price to pay for data integrity. It is, however, possible

to disable checksums, typically when the underlying filesystem supports checksums

natively. This is accomplished by using RawLocalFileSystem in place of Local

FileSystem. To do this globally in an application, it suffices to remap the implementation

for file URIs by setting the property fs.file.impl to the value org.apache.

hadoop.fs.RawLocalFileSystem. Alternatively, you can directly create a RawLocalFi

leSystem instance, which may be useful if you want to disable checksum verification

for only some reads, for example:

Configuration conf = ...

FileSystem fs = **new** RawLocalFileSystem();

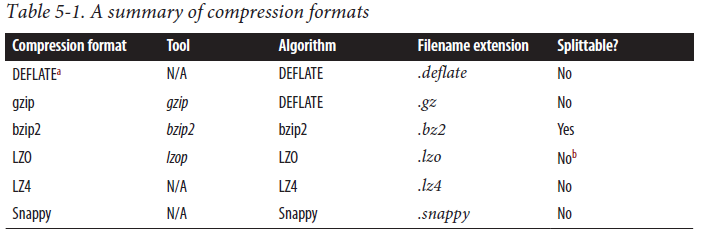
fs.initialize(**null**, conf);

Compression

File compression brings two major benefits: it reduces the space needed to store files,

and it speeds up data transfer across the network or to or from disk. When dealing with

large volumes of data, both of these savings can



All compression algorithms exhibit a space/time trade-off: faster compression and decompression

speeds usually come at the expense of smaller space savings. The tools

listed in Table 5-1 typically give some control over this trade-off at compression time

by offering nine different options: –1 means optimize for speed, and -9 means optimize

for space. For example, the following command creates a compressed file *file.gz* using

the fastest compression method:

gzip -1 file

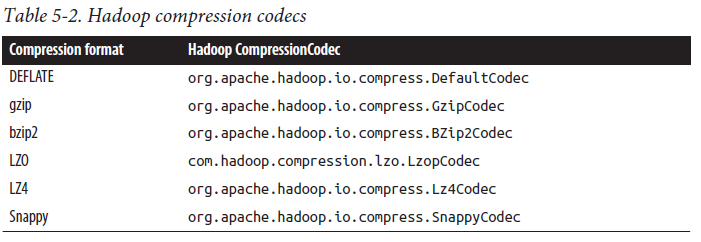
Codecs

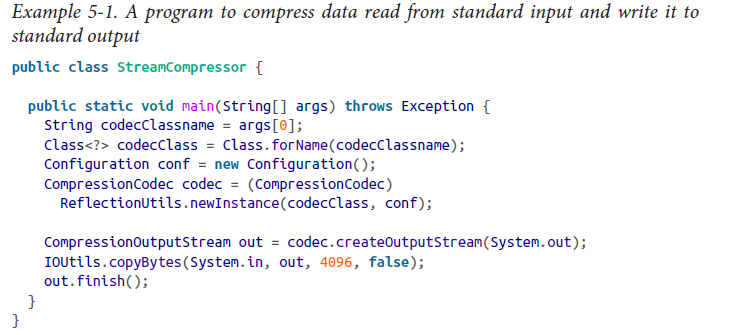
A *codec* is the implementation of a compression-decompression algorithm. In Hadoop,

a codec is represented by an implementation of the CompressionCodec interface. So, for

example, GzipCodec encapsulates the compression and decompression algorithm for

gzip. Table 5-2 lists the codecs that are available for Hadoop.





Inferring CompressionCodecs using CompressionCodecFactory

If you are reading a compressed file, normally you can infer which codec to use by

looking at its filename extension. A file ending in *.gz* can be read with GzipCodec, and

so on.



Native libraries

For performance, it is preferable to use a native library for compression and

decompression. For example, in one test, using the native gzip libraries reduced decompression

times by up to 50% and compression times by around 10% (compared to

the built-in Java implementation).

CodecPool. If you are using a native library and you are doing a lot of compression or

decompression in your application, consider using CodecPool, which allows you to reuse compressors and decompressors, thereby amortizing the cost of creating these

objects.

Compression and Input Splits

When considering how to compress data that will be processed by MapReduce, it is

important to understand whether the compression format supports splitting. Consider

an uncompressed file stored in HDFS whose size is 1 GB. With an HDFS block size of

128 MB, the file will be stored as 8 blocks, and a MapReduce job using this file as input

will create 8 input splits, each processed independently as input to a separate map task.

Imagine now that the file is a gzip-compressed file whose compressed size is 1 GB. As

before, HDFS will store the file as 8 blocks. However, creating a split for each block won’t

work, because it is impossible to start reading at an arbitrary point in the gzip stream

and therefore impossible for a map task to read its split independently of the others.

The gzip format uses DEFLATE to store the compressed data, and DEFLATE stores

data as a series of compressed blocks. The problem is that the start of each block is not

distinguished in any way that would allow a reader positioned at an arbitrary point in

the stream to advance to the beginning of the next block, thereby synchronizing itself

with the stream. For this reason, gzip does not support splitting.

In this case, MapReduce will do the right thing and not try to split the gzipped file, since

it knows that the input is gzip-compressed (by looking at the filename extension) and

that gzip does not support splitting. This will work, but at the expense of locality: a single

map will process the 8 HDFS blocks, most of which will not be local to the map. Also,

with fewer maps, the job is less granular and so may take longer to run.

If the file in our hypothetical example were an LZO file, we would have the same problem

because the underlying compression format does not provide a way for a reader to

synchronize itself with the stream. However, it is possible to preprocess LZO files using

an indexer tool that comes with the Hadoop LZO libraries, which you can obtain from

the sites listed in “Codecs” on page 103. The tool builds an index of split points, effectively

making them splittable when the appropriate MapReduce input format is used.

A bzip2 file, on the other hand, does provide a synchronization marker between blocks

(a 48-bit approximation of pi), so it does support splitting. (Table 5-1 lists whether each

compression format supports splitting.)

Which Compression Format Should I Use?

Hadoop applications process large datasets, so you should strive to take advantage of

compression. Which compression format you use depends on such considerations as

file size, format, and the tools you are using for processing. Here are some suggestions,

arranged roughly in order of most to least effective:

• Use a container file format such as sequence file ( on page 129), Avro datafile ( on page

354), ORCFile ( on page 138), or Parquet file ( on page 372), all of which support both

compression and splitting. A fast compressor such as LZO, LZ4, or Snappy is generally

a good choice.

• Use a compression format that supports splitting, such as bzip2 (although bzip2 is

fairly slow), or one that can be indexed to support splitting, such as LZO.

• Split the file into chunks in the application, and compress each chunk separately

using any supported compression format (it doesn’t matter whether it is splittable).

In this case, you should choose the chunk size so that the compressed chunks are

approximately the size of an HDFS block.

• Store the files uncompressed.

For large files, you should *not* use a compression format that does not support splitting

on the whole file, because you lose locality and make MapReduce applications very

inefficient.

Using Compression in MapReduce

As described in “Inferring CompressionCodecs using CompressionCodecFactory” on

page 104, if your input files are compressed, they will be decompressed automatically

as they are read by MapReduce, using the filename extension to determine which codec

to use.

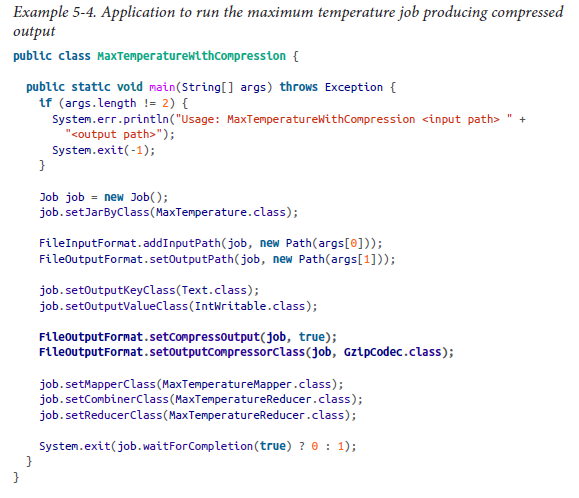
To compress the output of a MapReduce job, in the job configuration, set the mapre

duce.output.fileoutputformat.compress property to true and the mapreduce.out

put.fileoutputformat.compress.codec property to the classname of the compression

codec you want to use. Alternatively, you can use the static convenience methods

on FileOutputFormat to set these properties as shown in Example 5-4.



We run the program over compressed input (which doesn’t have to use the same compression

format as the output, although it does in this example) as follows:

% **hadoop MaxTemperatureWithCompression input/ncdc/sample.txt.gz output**

Each part of the final output is compressed; in this case, there is a single part:

% **gunzip -c output/part-r-00000.gz**

1949 111

1950 22

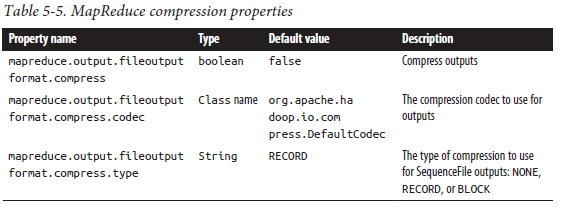
If you are emitting sequence files for your output, you can set the mapreduce.out

put.fileoutputformat.compress.type property to control the type of compression

to use. The default is RECORD, which compresses individual records. Changing this to

BLOCK, which compresses groups of records, is recommended because it compresses

better (see “The SequenceFile format” on page 135).



Compressing map output

Even if your MapReduce application reads and writes uncompressed data, it may benefit

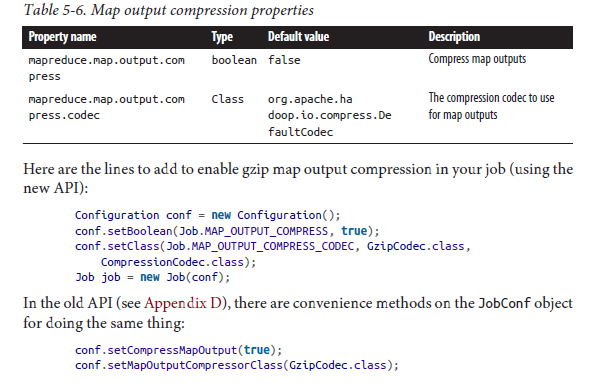
from compressing the intermediate output of the map phase. Since the map output is

written to disk and transferred across the network to the reducer nodes, by using a fast

compressor such as LZO, LZ4, or Snappy, you can get performance gains simply because

the volume of data to transfer is reduced. The configuration properties to enable compression

for map outputs and to set the compression format are shown in



Serialization

*Serialization* is the process of turning structured objects into a byte stream for transmission

over a network or for writing to persistent storage. *Deserialization* is the reverse

process of turning a byte stream back into a series of structured objects.

In Hadoop, interprocess communication between nodes in the system is implemented

using *remote procedure calls* (RPCs). The RPC protocol uses serialization to render the

message into a binary stream to be sent to the remote node, which then deserializes the

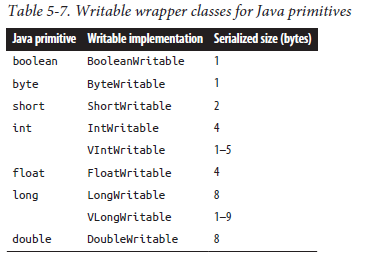
binary stream into the original message. In general, it is desirable that an RPC serialization

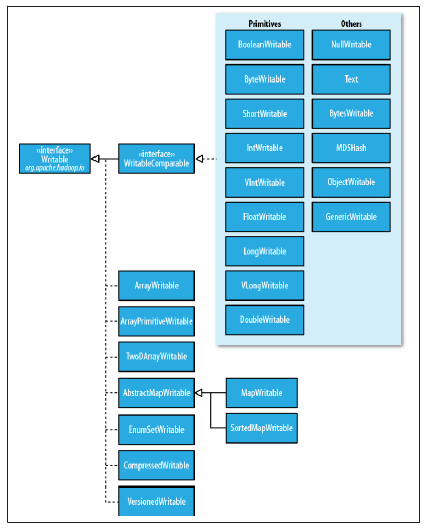
format is:

Writable Classes

Hadoop comes with a large selection of Writable classes in the org.apache.ha

doop.io package.





NullWritable

NullWritable is a special type of Writable, as it has a zero-length serialization. No bytes

are written to or read from the stream. It is used as a placeholder; for example, in Map‐

Reduce, a key or a value can be declared as a NullWritable when you don’t need to use

that position, effectively storing a constant empty value. NullWritable can also be useful

as a key in SequenceFile when you want to store a list of values, as opposed to keyvalue

pairs. It is an immutable singleton, and the instance can be retrieved by calling

NullWritable.get().

SequenceFile

Imagine a logfile where each log record is a new line of text. If you want to log binary

types, plain text isn’t a suitable format. Hadoop’s SequenceFile class fits the bill in

this situation, providing a persistent data structure for binary key-value pairs. To use it

as a logfile format, you would choose a key, such as timestamp represented by a

LongWritable, and the value would be Writable that represents the quantity being

logged.

SequenceFiles also work well as containers for smaller files. HDFS and MapReduce are

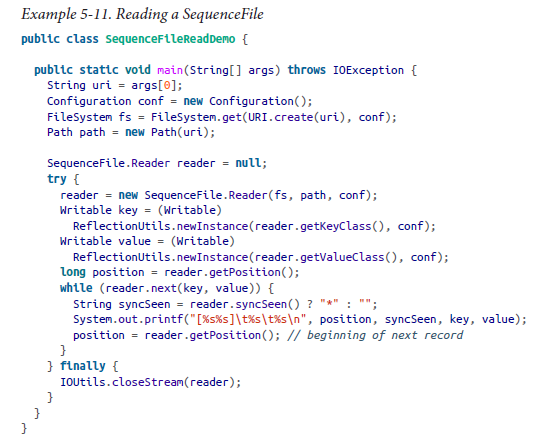
optimized for large files, so packing files into a SequenceFile makes storing

and processing the smaller files more efficient. (“Processing a whole file as a record” on

page 230 contains a program to pack files into a SequenceFile).



Reading a SequenceFile



Displaying a SequenceFile with the command-line interface

% **hadoop fs -text numbers.seq | head**

The SequenceFile format

A sequence file consists of a header followed by one or more records (see Figure 5-2).

The first three bytes of a sequence file are the bytes SEQ, which acts as a magic number,

followed by a single byte representing the version number. The header contains other

fields, including the names of the key and value classes, compression details, userdefined

metadata, and the sync marker.5 Recall that the sync marker is used to allow a

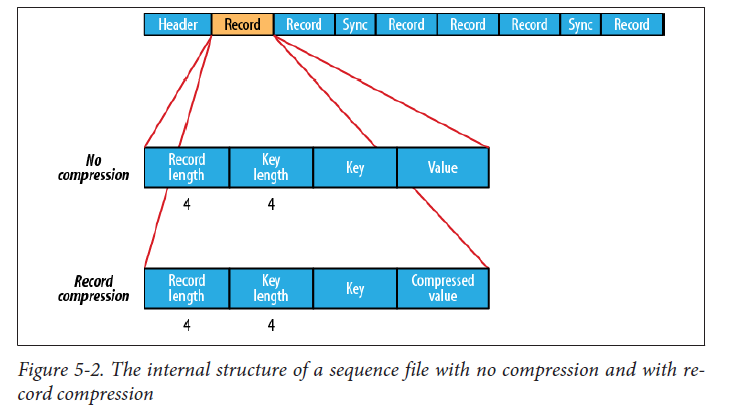
reader to synchronize to a record boundary from any position in the file. Each file has

a randomly generated sync marker, whose value is stored in the header. Sync markers

appear between records in the sequence file. They are designed to incur less than a 1%

storage overhead, so they don’t necessarily appear between every pair of records (such

is the case for short records).



The internal format of the records depends on whether compression is enabled, and if

it is, whether it is record compression or block compression.

If no compression is enabled (the default), each record is made up of the record length

(in bytes), the key length, the key, and then the value. The length fields are written as

four-byte integers adhering to the contract of the writeInt() method of java.io.Da

taOutput. Keys and values are serialized using the Serialization defined for the class

being written to the sequence file.

The format for record compression is almost identical to no compression, except the

value bytes are compressed using the codec defined in the header. Note that keys are

not compressed.

Block compression compresses multiple records at once; it is therefore more compact

than and should generally be preferred over record compression because it has the

opportunity to take advantage of similarities between records. (See Figure 5-3.) Records

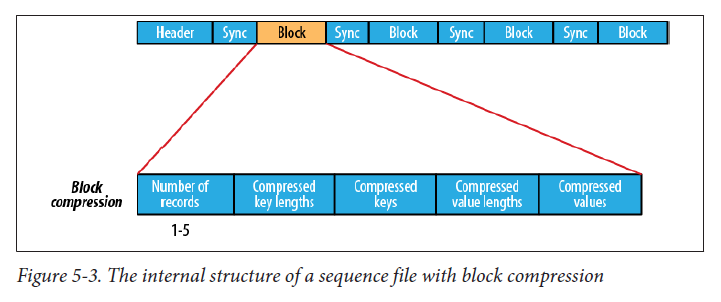
are added to a block until it reaches a minimum size in bytes, defined by the

io.seqfile.compress.blocksize property; the default is 1 million bytes. A sync

marker is written before the start of every block. The format of a block is a field indicating

the number of records in the block, followed by four compressed fields: the key lengths,

the keys, the value lengths, and the values.



MapFile

A MapFile is a sorted SequenceFile with an index to permit lookups by key. The index

is itself a SequenceFile that contains a fraction of the keys in the map (every 128th key

by default). The idea is that the index can be loaded into memory to provide fast lookups

from the main data file, which is another SequenceFile containing all the map entries

in sorted key order.

Other File Formats and Column-Oriented Formats

Avro datafiles (covered in “Avro Datafiles” on page 354) are like sequence files in that they

are designed for large-scale data processing (they are compact and splittable), but they

are portable across different programming languages. Objects stored in Avro datafiles

are described by a schema, rather than in the Java code of the implementation of a

Writable object, as is the case for sequence files, making them very Java-centric. Avro

datafiles are widely supported across components in the Hadoop ecosystem, so they are

a good default choice for a binary format.

Sequence file, map file, and Avro datafiles are all row-oriented file formats, which means

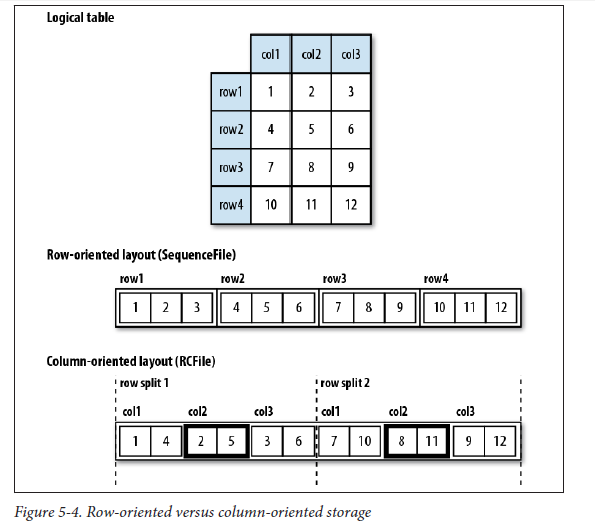
that the values for each row are stored contiguously in the file. In a column-oriented

format the rows in a file (or, equivalently, a table in Hive) are broken up into row splits,

then each split is stored in column-oriented fashion: the values for each row in the first

column are stored first, followed by the values for each row in the second column, and

so on.



A column-oriented layout permits columns that are not accessed in a query to be skipped.

Consider a query of the table in Figure 5-4 that processes only column 2. With

row-oriented storage, like a sequence file, the whole row (stored in a sequence file record)

is loaded into memory, even though only the second column is actually read. Lazy

deserialization saves some processing cycles by deserializing only the column fields that

are accessed, but it can’t avoid the cost of reading each row’s bytes from disk.

With column-oriented storage, only the column 2 parts of the file (highlighted in the

figure) need to be read into memory. In general, column-oriented formats work well

when queries access only a small number of columns in the table. Conversely, roworiented

formats are appropriate when a large number of columns of a single row are

needed for processing at the same time.

Column-oriented formats need more memory for reading and writing, since they have

to buffer a row-split in memory, rather than just a single row. Also, since it’s not usually possible to control when writes occur (via flush or sync operations), column-oriented

formats are not suited to streaming writes, since if the writer process fails the current

file cannot be recovered. On the other hand, row-oriented formats like sequence file

and Avro datafile can be read up to the last sync point after a writer failure. It is for this

reason that Flume (see Chapter 14) uses row-oriented formats like sequence file and

Avro datafiles.

The first column-oriented file format in Hadoop was Hive’s *RCFile*, short for *Record*

*Columnar File*. It has since been superceded by Hive’s *ORCFile* (*Optimized Record Columnar*

*File*), and *Parquet* (covered in Chapter 13). Parquet is a general-purpose columnoriented

file format based on Google’s Dremel, and has wide support across Hadoop

components. Avro also has a column-oriented format called *Trevni*.

Developing a MapReduce Application

Resources are added to a Configuration in order:

Configuration conf = **new** Configuration();

conf.addResource("configuration-1.xml");

conf.addResource("configuration-2.xml");

Properties defined in resources that are added later override the earlier definitions. So

the size property takes its value from the second configuration file, *configuration-2.xml*:

assertThat(conf.getInt("size", 0), is(12));

However, properties that are marked as final cannot be overridden in later definitions.

The weight property is final in the first configuration file, so the attempt to override

it in the second fails, and it takes the value from the first:

assertThat(conf.get("weight"), is("heavy"));

Managing Configuration

A pseudodistributed cluster is one whose daemons all run on the local machine.

The *hadoop-local.xml* file contains the default Hadoop configuration for the default

filesystem and the local (in-JVM) framework for running MapReduce jobs:

<?xml version="1.0"?>

<configuration>

<property>

<name>fs.defaultFS</name>

<value>file:///</value>

</property>

<property>

<name>mapreduce.framework.name</name>

<value>local</value>

</property>

</configuration>

Finally, *hadoop-cluster.xml* contains details of the cluster’s namenode and YARN resource

manager addresses. In practice, you would name the file after the name of the

cluster, rather than “cluster” as we have here:

<?xml version="1.0"?>

<configuration>

<property>

<name>fs.defaultFS</name>

<value>hdfs://namenode/</value>

</property>

<property>

<name>mapreduce.framework.name</name>

<value>yarn</value>

</property>

<property>

<name>yarn.resourcemanager.address</name>

<value>resourcemanager:8032</value>

</property>

</configuration>

Writing a Unit Test with MRUnit

The map and reduce functions in MapReduce are easy to test in isolation, which is a

consequence of their functional style. MRUnit is a testing library that makes it easy to

pass known inputs to a mapper or a reducer and check that the outputs are as expected.

MRUnit is used in conjunction with a standard test execution framework, such as JUnit,

so you can run the tests for MapReduce jobs as a part of your normal development

environment.



Reducer

The reducer has to find the maximum value for a given key. Here’s a simple test for this

feature, which uses a ReduceDriver:

@Test

**public void** returnsMaximumIntegerInValues() **throws** IOException,

InterruptedException {

**new** ReduceDriver<Text, IntWritable, Text, IntWritable>()

.withReducer(**new** MaxTemperatureReducer())

.withInput(**new** Text("1950"), Arrays.asList(**new** IntWritable(10), **new** IntWritable(5)))

.withOutput(**new** Text("1950"), **new** IntWritable(10))

.runTest();

}

Running Locally on Test Data

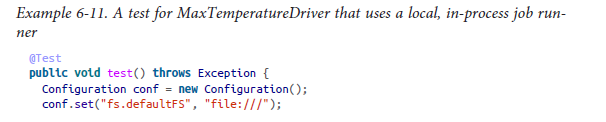


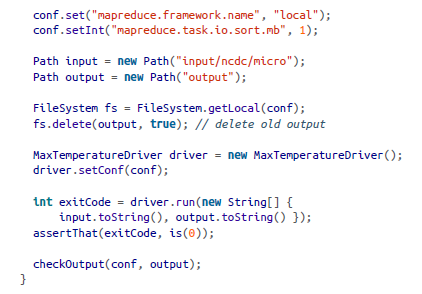
Testing the Driver

There are two approaches to doing this. The first is to use the local job runner and run

the job against a test file on the local filesystem. The code in Example 6-11 gives an idea

of how to do this.





The second way of testing the driver is to run it using a “mini-” cluster. Hadoop has a

set of testing classes, called MiniDFSCluster, MiniMRCluster, and MiniYARNCluster,

that provide a programmatic way of creating in-process clusters. Unlike the local job

runner, these allow testing against the full HDFS, MapReduce, and YARN machinery.

Bear in mind, too, that node managers in a mini-cluster launch separate JVMs to run

tasks in, which can make debugging more difficult.

Mini-clusters are used extensively in Hadoop’s own automated test suite, but they can

be used for testing user code, too. Hadoop’s ClusterMapReduceTestCase abstract class

provides a useful base for writing such a test, handles the details of starting and stopping

the in-process HDFS and YARN clusters in its setUp() and tearDown() methods, and

generates a suitable configuration object that is set up to work with them. Subclasses

need only populate data in HDFS (perhaps by copying from a local file), run a MapReduce

job, and confirm the output is as expected. Refer to the MaxTemperatureDriver

MiniTest class in the example code that comes with this book for the listing.

Running on a Cluster

The MapReduce Web UI

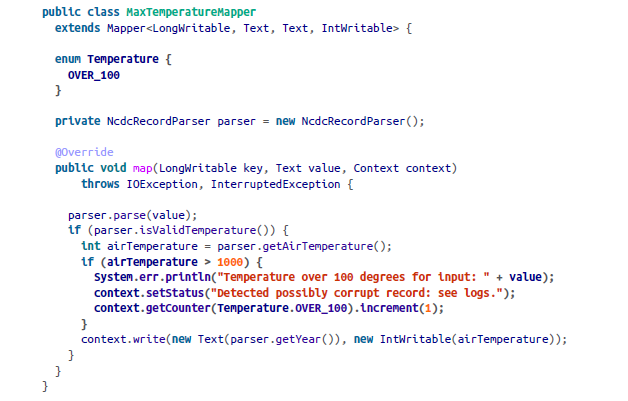
Hadoop comes with a web UI for viewing information about your jobs. It is useful for

following a job’s progress while it is running, as well as finding job statistics and logs

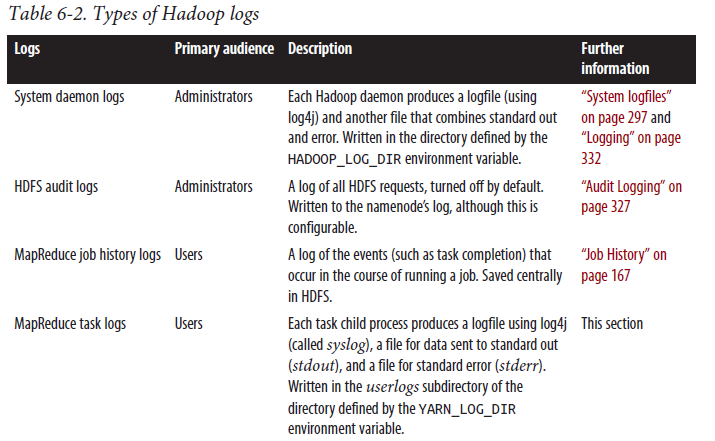
after the job has completed. You can find the UI at http://resource-manager-host:

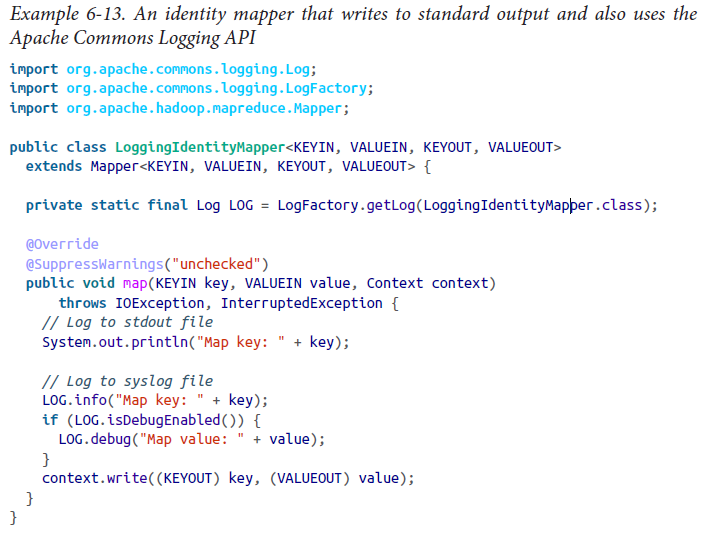
8088/.

Debugging a Job



Hadoop Logs





The default log level is INFO, so DEBUG level messages do not appear in the *syslog* task

logfile. However, sometimes you want to see these messages. To do this, set mapre

duce.map.log.level or mapreduce.reduce.log.level, as appropriate. For example,

in this case we could set it for the mapper to see the map values in the log as follows:

% **hadoop jar hadoop-examples.jar LoggingDriver -conf conf/hadoop-cluster.xml \**

**-D mapreduce.map.log.level=DEBUG input/ncdc/sample.txt logging-out**

There are some controls for managing the retention and size of task logs. By default,

logs are deleted after a minimum of three hours (set this using the

yarn.nodemanager.log.retain-seconds property, although this is ignored if log aggregation

is enabled). You can also set a cap on the maximum size of each logfile using

the mapreduce.task.userlog.limit.kb property, which is 0 by default, meaning there

is no cap.

Remote Debugging

*Reproduce the failure locally*

Often the failing task fails consistently on a particular input. You can try to reproduce

the problem locally by downloading the file that the task is failing on and

running the job locally, possibly using a debugger such as Java’s VisualVM.

*Use JVM debugging options*

A common cause of failure is a Java out of memory error in the task JVM. You can

set mapred.child.java.opts to include -XX:-HeapDumpOnOutOfMemoryError -

XX:HeapDumpPath=*/path/to/dumps*. This setting produces a heap dump that can

be examined afterward with tools such as *jhat* or the Eclipse Memory Analyzer.

Note that the JVM options should be added to the existing memory settings specified

by mapred.child.java.opts.

*Use task profiling*

Java profilers give a lot of insight into the JVM, and Hadoop provides a mechanism

to profile a subset of the tasks in a job.

Another useful property for debugging is yarn.nodemanager.delete.debug-delaysec,

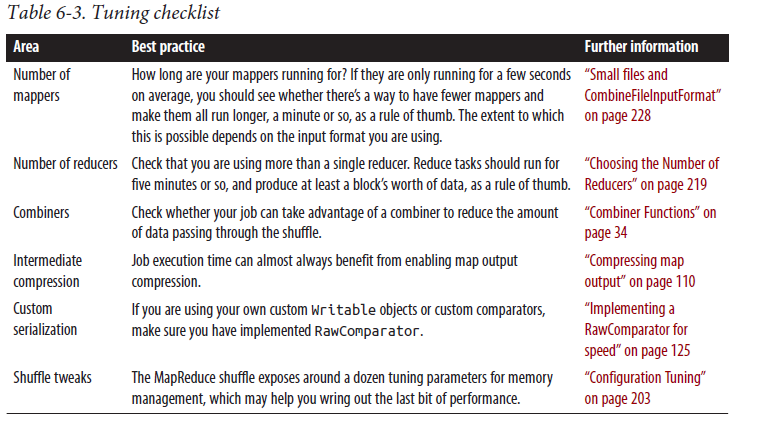
which is the number of seconds to wait to delete localized task attempt files, such

as the script used to launch the task container JVM. If this is set on the cluster to a

reasonably large value (e.g. 600 for 10 minutes), then you have enough time to look at

the files before they are deleted.

Tuning a Job



Profiling Tasks

Like debugging, profiling a job running on a distributed system such as MapReduce

presents some challenges. Hadoop allows you to profile a fraction of the tasks in a job,

and as each task completes, pulls down the profile information to your machine for later

analysis with standard profiling tools

The HPROF profiler

There are a number of configuration properties to control profiling, which are also

exposed via convenience methods on JobConf. Enabling profiling is as simple as setting

the property mapreduce.task.profile to true:

**% hadoop jar hadoop-examples.jar v4.MaxTemperatureDriver \**

**-conf conf/hadoop-cluster.xml \**

**-D mapreduce.task.profile=true \**

**input/ncdc/all max-temp**

This runs the job as normal, but adds an -agentlib parameter to the Java command

used to launch the task containers on the node managers. You can control the precise

parameter that is added by setting the mapreduce.task.profile.params property. The

default uses HPROF, a profiling tool that comes with the JDK that, although basic, can

give valuable information about a program’s CPU and heap usage.

It doesn’t usually make sense to profile all tasks in the job, so by default only those with

IDs 0, 1, and 2 are profiled (for both maps and reduces). You can change this by setting

mapreduce.task.profile.maps and mapreduce.task.profile.reduces to specify the

range of task IDs to profile.

The profile output for each task is saved with the task logs in the *userlogs* subdirectory

of the node manager’s local log directory (alongside the *syslog*, *stdout*, and *stderr* files),

MapReduce Workflows

JobControl

When there is more than one job in a MapReduce workflow, the question arises: how

do you manage the jobs so they are executed in order? There are several approaches,

and the main consideration is whether you have a linear chain of jobs or a more complex

directed acyclic graph (DAG) of jobs.

For a linear chain, the simplest approach is to run each job one after another, waiting

until a job completes successfully before running the next:

JobClient.runJob(conf1);

JobClient.runJob(conf2);

The approach is similar with the new MapReduce API, except you need to examine the

Boolean return value of the waitForCompletion() method on Job: true means the job

succeeded, and false means it failed.

Apache Oozie

Apache Oozie is a system for running workflows of dependent jobs. It is composed of

two main parts: a *workflow engine* that stores and runs workflows composed of different

types of Hadoop jobs (MapReduce, Pig, Hive, and so on), and a *coordinator engine* that

runs workflow jobs based on predefined schedules and data availability. Oozie has been

designed to scale, and it can manage the timely execution of thousands of workflows in

a Hadoop cluster, each composed of possibly dozens of constituent jobs.

Unlike JobControl, which runs on the client machine submitting the jobs, Oozie runs

as a service in the cluster, and clients submit workflow definitions for immediate or later execution. In Oozie parlance, a workflow is a DAG of *action nodes* and *control-flow*

*nodes*.

An action node performs a workflow task, such as moving files in HDFS, running a

MapReduce, Streaming, Pig, or Hive job, performing a Sqoop import, or running an

arbitrary shell script or Java program. A control-flow node governs the workflow execution

between actions by allowing such constructs as conditional logic (so different

execution branches may be followed depending on the result of an earlier action node) or parallel execution. When the workflow completes, Oozie can make an HTTP callback

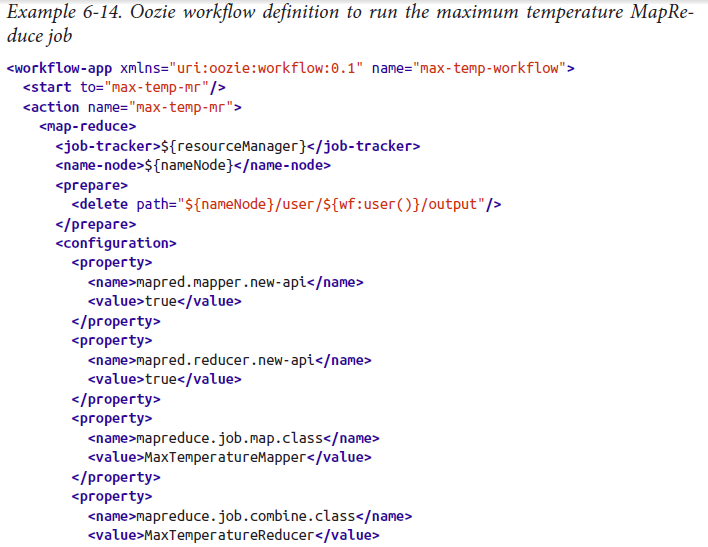
to the client to inform it of the workflow status. It is also possible to receive callbacks every time the workflow enters or exits an action node.

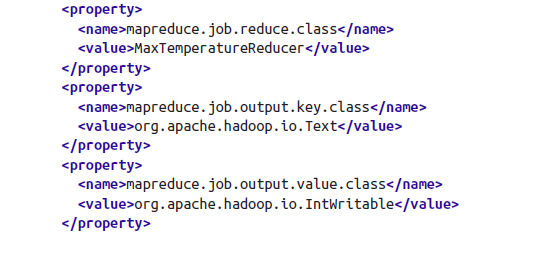
Defining an Oozie workflow

Workflow definitions are written in XML using the Hadoop Process Definition Language,

the specification for which can be found on the Oozie website. Example 6-14

shows a simple Oozie workflow definition for running a single MapReduce job.







This workflow has three control-flow nodes and one action node: a start control node,

a map-reduce action node, a kill control node, and an end control node. The nodes

and allowed transitions between them are shown in Figure 6-4.

