



BMS INSTITUTE OF TECHNOLOGY & MANAGEMENT

Yelahanka, Bangalore-64

Department of MCA

Big Data Analytics – Lecture Notes

Module 3

Data

We live in the data age. It's not easy to measure the total volume of data stored electronically.

This flood of data is coming from many sources. For example:

- The New York Stock Exchange generates about one terabyte of new trade data per day.
- Facebook hosts approximately 10 billion photos, taking up one petabyte of storage.
- Ancestry.com, the genealogy site, stores around 2.5 petabytes of data.
- The Internet Archive stores around 2 petabytes of data and is growing at a rate of 20 terabytes per month.
- The Large Hadron Collider near Geneva, Switzerland, will produce about 15 petabytes of data per year.

The trend is for every individual's data footprint to grow, but perhaps more important, the amount of data generated by machines will be even greater than that generated by people. Machine logs, RFID readers, sensor networks, vehicle GPS traces, retail transactions—all of these contribute to the growing mountain of data.

The volume of data being made publicly available increases every year, too. Organizations no longer have to merely manage their own data; success in the future will be dictated to a large extent by their ability to extract value from other organizations' data.

However complex your algorithms are, often they can be beaten simply by having more data. Though we are having large volumes of data, we are struggling to store and analyze it.

Data Storage and Analysis

Though the storage capacities of hard drives have increased massively over the years, access speeds—the rate at which data can be read from drives have not kept up. One typical drive from 1990 could store 1,370 MB of data and had a transfer speed of 4.4 MB/s, 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. Using only one hundredth of a disk may seem wasteful. But we can store one hundred datasets, each of which is one terabyte, and provide shared access to them.

The problems encountered during storage and analysis of data

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. This is how RAID works, for instance, although Hadoop's filesystem, the Hadoop Distributed Filesystem (HDFS), takes a slightly different approach.

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. 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. There are two parts to the computation, the map and the reduce, and it's the interface between the two where the "mixing" occurs. Like HDFS, MapReduce has built-in reliability.

This, in a nutshell, is what Hadoop provides: a reliable shared storage and analysis system. The storage is provided by HDFS and analysis by MapReduce. There are other parts to Hadoop, but these capabilities are its kernel.

Comparison with Other Systems

MapReduce is a *batch* query processor, and the ability to run an ad hoc query against your whole dataset and get the results in a reasonable time is transformative. It changes the way you think about data and unlocks data that was previously archived on tape or disk. It gives people the opportunity to innovate with data. Questions that took too long to get answered before can now be answered, which in turn leads to new questions and new insights. 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. By bringing several hundred gigabytes of data together and having the tools to analyze it, the Rackspace engineers were able to gain an understanding of the data that they otherwise would never have had, and, furthermore, they were able to use what they had learned to improve the service for their customers.

Rational Database Management System

Why can't we use databases with lots of disks to do large-scale batch analysis? Why is MapReduce needed?

The seek time is improving more slowly than transfer rate. Seeking is the process of moving the disk's head to a particular place on the disk to read or write data. It characterizes the latency of a disk operation, whereas the transfer rate corresponds to a disk's bandwidth.

If the **data access pattern is dominated by seeks**, it will take longer to read or write large portions of the dataset than streaming through it, which operates at the transfer rate. On the other hand, for updating a small proportion of records in a database, a traditional B-Tree (the data structure used in relational databases, which is limited by the rate it can perform seeks) works well. For updating the majority of a database, a B-Tree is less efficient than MapReduce, which uses Sort/Merge to rebuild the database.

In many ways, MapReduce can be seen as a complement to a Rational Database Management System (RDBMS). MapReduce is a good fit for problems that need **to analyze the whole dataset in a batch fashion**, particularly for ad hoc analysis. An RDBMS is good for point queries or updates, where the dataset has been indexed to deliver low-latency retrieval and update times of a relatively small amount of data. MapReduce suits applications where the **data is written once and read many times**, whereas a relational database is good for datasets that are continually updated.

Table 1-1. RDBMS compared to MapReduce

	Traditional RDBMS	MapReduce
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Structure	Static schema	Dynamic schema
Integrity	High	Low
Scaling	Nonlinear	Linear

Another difference between MapReduce 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. MapReduce **works well on unstructured or semi-structured data** because it is designed to interpret the data at processing time.

Relational data is often **normalized to retain its integrity and remove redundancy**. Normalization poses problems for MapReduce because it makes reading a record a nonlocal operation, and one of the central assumptions that MapReduce makes is that it is possible to perform (high-speed) streaming reads and writes.

A web server log is a good example of a set of records that is *not* normalized (for example, the client hostnames are specified in full each time, even though the same client may appear many times), and this is one reason that logfiles of all kinds are particularly well-suited to analysis with MapReduce.

MapReduce is a **linearly scalable programming model**. The programmer writes two functions—a map function and a reduce function—each of which defines a mapping from one set of key-value pairs to another. These functions are oblivious to the size of the data or the cluster that they are operating on, so they can be used unchanged for a small dataset and for a massive one. More important, if you double the size of the input data, a job will run twice as slow. 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.

Over time, however, the differences between relational databases and MapReduce systems are likely to blur—both as relational databases start incorporating some of the ideas from MapReduce and, from the other direction, as higher-level query languages built on MapReduce (such as Pig and Hive) make MapReduce systems more approachable for traditional database programmers.

Grid Computing

The High Performance Computing (HPC) and Grid Computing communities have been doing large-scale data processing for years, using such Application Program Interfaces (APIs) as Message Passing Interface (MPI). The approach in HPC is to distribute the work across a cluster of machines, which access a shared file system, hosted by a Storage Area Network (SAN). This works well for predominantly compute-intensive jobs, but it becomes a problem when nodes need to access larger data volumes since the network bandwidth is the bottleneck and compute nodes become idle.

MapReduce tries to colocate the data with the compute node, so data access is fast because it is local. This feature, known as **data locality**, is at the heart of MapReduce 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), MapReduce implementations conserve it by explicitly modelling network topology. MapReduce operates only at the higher level: the programmer thinks in terms of functions of key and value pairs, and the data flow is implicit.

Coordinating the processes in a large-scale distributed computation is a challenge. The hardest aspect is **gracefully handling partial failure**—when you don't know whether or not a remote process has failed—and still making progress with the overall computation. MapReduce spares the programmer from having to think about failure, since the implementation detects failed map or reduce 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. So from the programmer's point of view, the order in which the tasks run doesn't matter. By contrast, MPI programs have to explicitly manage their own checkpointing and recovery, which gives more control to the programmer but makes them more difficult to write.

MapReduce might sound like a restrictive programming model as we are limited to key and value types that are related in specified ways, and mappers and reducers run with very limited coordination between one another (the mappers pass keys and values to reducers). Large range of algorithms can be expressed in MapReduce, from image analysis, to graph-based problems, to machine learning algorithms.

Volunteer Computing

SETI, the Search for Extra-Terrestrial Intelligence, runs a project called [SETI@home](#) in which volunteers donate CPU time from their otherwise idle computers to analyze radio telescope data for signs of intelligent life outside earth. SETI@home is the most well-known of many *volunteer computing* projects. Others include the Great Internet Mersenne Prime Search (to search for large prime numbers) and Folding@home (to understand protein folding and how it relates to disease).

Volunteer computing projects work by breaking the problem they are trying to solve into chunks called **work units**, which are sent to computers around the world to be analyzed. For example, a SETI@home work unit is about 0.35 MB of radio telescope data, and takes hours or days to analyze on a typical home computer. When the analysis is completed, the results are sent back to the server, and the client gets another work unit. As a precaution to combat cheating, each work unit is sent to three different machines and needs at least two results to agree to be accepted.

Although SETI@home may be similar to MapReduce (breaking a problem into independent pieces to be worked on in parallel), there are some significant differences. The SETI@home problem is very CPU-intensive, which makes it suitable for running on hundreds of thousands of computers across the world because the time to transfer the work unit is dwarfed by the time to run the computation on it. Volunteers are donating CPU cycles, not bandwidth.

MapReduce is designed to **run jobs that last minutes or hours on trusted, dedicated hardware running in a single data center with very high aggregate bandwidth interconnects**. By contrast, SETI@home runs a perpetual computation on untrusted machines on the Internet with highly variable connection speeds and no data locality.

A Brief History of Hadoop

Hadoop was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library. Hadoop has its origins in Apache Nutch, an open source web search engine, itself a part of the Lucene project. The following is the timeline of the progress of Hadoop:

- 2004: Initial versions of what is now Hadoop Distributed Filesystem and MapReduce implemented by Doug Cutting and Mike Cafarella.
- December 2005: Nutch ported to the new framework. Hadoop runs reliably on 20 nodes.
- January 2006: Doug Cutting joins Yahoo!.
- February 2006: Apache Hadoop project officially started to support the standalone development of MapReduce and HDFS.
- February 2006: Adoption of Hadoop by Yahoo! Grid team.
- April 2006: Sort benchmark (10 GB/node) run on 188 nodes in 47.9 hours.
- May 2006: Yahoo! set up a Hadoop research cluster—300 nodes.
- May 2006: Sort benchmark run on 500 nodes in 42 hours (better hardware than April benchmark).
- October 2006: Research cluster reaches 600 nodes.
- December 2006: Sort benchmark run on 20 nodes in 1.8 hours, 100 nodes in 3.3 hours, 500 nodes in 5.2 hours, 900 nodes in 7.8 hours.
- January 2007: Research cluster reaches 900 nodes.
- April 2007: Research clusters—two clusters of 1000 nodes.
- April 2008: Won the 1-terabyte sort benchmark in 209 seconds on 900 nodes.
- October 2008: Loading 10 terabytes of data per day onto research clusters.
- March 2009: 17 clusters with a total of 24,000 nodes.
- April 2009: Won the minute sort by sorting 500 GB in 59 seconds (on 1,400 nodes) and the 100-terabyte sort in 173 minutes (on 3,400 nodes).

Apache Hadoop and the Hadoop Ecosystem

Although Hadoop is best known for MapReduce and its distributed file system, the term is also used for a family of related projects that fall under the umbrella of infrastructure for distributed computing and large-scale data processing.

All of the core projects are hosted by the Apache Software Foundation which provides support for a community of open source software projects, including the original HTTP Server.

The Hadoop projects are described briefly here:

Common

A set of components and interfaces for distributed file systems and general I/O (serialization, Java RPC, persistent data structures).

Avro

A serialization system for efficient, cross-language RPC and persistent data storage.

MapReduce

A distributed data processing model and execution environment that runs on large clusters of commodity machines.

HDFS

A distributed file system that runs on large clusters of commodity machines.

Pig

A data flow language and execution environment for exploring very large datasets. Pig runs on HDFS and MapReduce clusters.

Hive

A distributed data warehouse. Hive manages data stored in HDFS and provides a query language based on SQL (and which is translated by the runtime engine to MapReduce jobs) for querying the data.

HBase

A distributed, column-oriented database. HBase uses HDFS for its underlying storage, and supports both batch-style computations using MapReduce and point queries (random reads).

ZooKeeper

A distributed, highly available coordination service. ZooKeeper provides primitives such as distributed locks that can be used for building distributed applications.

Sqoop

A tool for efficient bulk transfer of data between structured data stores (such as relational databases) and HDFS.

Oozie

A service for running and scheduling workflows of Hadoop jobs (including Map- Reduce, Pig, Hive, and Sqoop jobs).

Hadoop Releases

There are a few active release series. The 1.x release series is a continuation of the 0.20 release series and contains the most stable versions of Hadoop currently available. This series includes secure Kerberos authentication, which prevents unauthorized access to Hadoop data.

The 0.22 and 2.x release series are not currently stable. 2.x includes several major new features:

- A new MapReduce runtime, called MapReduce 2, implemented on a new system called YARN (Yet Another Resource Negotiator), which is a general resource management system for running distributed applications.
- HDFS federation, which partitions the HDFS namespace across multiple namenodes to support clusters with very large numbers of files.
- HDFS high-availability, which removes the namenode as a single point of failure by supporting standby namenodes for failover.

Table 1-2. Features supported by Hadoop release series

Feature	1.x	0.22	2.x
Secure authentication	Yes	No	Yes
Old configuration names	Yes	Deprecated	Deprecated
New configuration names	No	Yes	Yes
Old MapReduce API	Yes	Yes	Yes
New MapReduce API	Yes (with some missing libraries)	Yes	Yes
MapReduce 1 runtime (Classic)	Yes	Yes	No
MapReduce 2 runtime (YARN)	No	No	Yes
HDFS federation	No	No	Yes
HDFS high-availability	No	No	Yes



BMS INSTITUTE OF TECHNOLOGY & MANAGEMENT

Yelahanka, Bangalore-64

Department of MCA

Big Data Analytics – Lecture Notes

Module 4

When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. File systems that manage the storage across a network of machines are called *distributed file systems*. Since they are network-based, all the complications of network programming kick in, thus making distributed file systems more complex than regular disk file systems. For example, one of the biggest challenges is making the file system tolerate node failure without suffering data loss.

Hadoop comes with a distributed file system called HDFS, which stands for ***Hadoop Distributed File system***.

The Design of HDFS

HDFS is a file system 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.

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 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) 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.

It is also worth examining the applications for which using HDFS does not work so well. These are areas where HDFS is not a good fit today:

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 is currently a better choice for low-latency access.

Lots of small files

Because the namenode holds file system metadata in memory, the limit to the number of files in a file system 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.

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. There is no support for multiple writers or for modifications at arbitrary offsets in the file.

HDFS Concepts

Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. File systems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. File system blocks are typically a few kilobytes in size, whereas disk blocks are normally 512 bytes. This is generally transparent to the file system user who is simply reading or writing a file of whatever length. However, there are tools to perform file system maintenance, such as ***df*** and ***fsck***, that operate on the file system block level.

HDFS, too, has the concept of a block, but it is a much larger unit—64 MB by default. Like in a file system for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a file system for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage.

Having a block abstraction for a distributed file system 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 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. 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.

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. Similarly, some applications may choose to set a high replication factor for the blocks in a popular file to spread the read load on the cluster.

Like its disk file system cousin, HDFS's `fsck` command understands blocks. For example, running:

```
% hadoop fsck / -files -blocks
```

will list the blocks that make up each file in the file system.

Namenodes and Datanodes

An HDFS cluster has two types of nodes operating in a master-worker pattern: a *name-node* (the master) and a number of *datanodes* (workers). The namenode manages the file system namespace. It maintains the file system 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 file system 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 file system cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the file system 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 file system metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple file systems. 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.

HDFS Federation

The namenode keeps a reference to every file and block in the file system in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling. 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 file systems and using the secondary namenode to create checkpoints protects against data loss, but it does not provide high-availability of the file system. 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 file system 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.

The 2.x release series of Hadoop remedies 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.

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**. Failover controllers are pluggable, but the first 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.

Failover may also be initiated manually by an administrator, for example, in the case of routine maintenance. This is known as a **graceful failover**, since the failover controller arranges an orderly transition for both namenodes to switch roles.

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 system employs a range of fencing mechanisms, including killing the namenode's process, revoking its access to the shared storage directory, and disabling its network port via a remote management command. As a last resort, the previously active namenode can be fenced with a technique known as **STONITH**, or “shoot the other node in the head,” which uses a specialized power distribution unit to forcibly power down the host machine.

Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname that is mapped to a pair of namenode addresses (in the configuration file), and the client library tries each namenode address until the operation succeeds.

The Command-Line Interface

We're going to have a look at HDFS by interacting with it from the command line. There are many other interfaces to HDFS, but the command line is one of the simplest ways.

We are going to run HDFS on one machine, so first follow the instructions for setting up Hadoop in pseudo distributed mode. Later we'll see how to run HDFS on a cluster of machines to give us scalability and fault tolerance.

There are two properties that we set in the pseudo distributed configuration. The first is **fs.default.name**, set to `hdfs://localhost/`, which is used to set a default file system for Hadoop. File systems are specified by a URI, and here we have used an hdfs URI to configure Hadoop to

use HDFS by default. The HDFS daemons will use this property to determine the host and port for the HDFS namenode. We'll be running it on localhost, on the default HDFS port, 8020. And HDFS clients will use this property to work out where the namenode is running so they can connect to it.

We set the second property, **dfs.replication**, to 1 so that HDFS doesn't replicate file system blocks by the default factor of three. When running with a single datanode, HDFS can't replicate blocks to three datanodes, so it would perpetually warn about blocks being under-replicated. This setting solves that problem.

Basic Filesystem Operations

The filesystem is ready to be used, and we can do all of the usual filesystem operations, such as reading files, creating directories, moving files, deleting data, and listing directories. You can type **hadoop fs -help** to get detailed help on every command.

Start by copying a file from the local filesystem to HDFS:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt  
hdfs://localhost/user/tom/ quangle.txt
```

This command invokes Hadoop's filesystem shell command **fs**, which supports a number of subcommands—in this case, we are running **-copyFromLocal**. The local file **quangle.txt** is copied to the **file/user/tom/quangle.txt** on the HDFS instance running on localhost. In fact, we could have omitted the scheme and host of the URI and picked up the default, **hdfs://localhost**, as specified in **core-site.xml**:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt /user/tom/quangle.txt
```

We also could have used a relative path and copied the file to our home directory in HDFS, which in this case is **/user/tom**:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt 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
```

```
% md5 input/docs/quangle.txt quangle.copy.txt
```

```
MD5 (input/docs/quangle.txt) = a16f231da6b05e2ba7a339320e7dacd9
```

```
MD5 (quangle.copy.txt) = a16f231da6b05e2ba7a339320e7dacd9
```

The MD5 digests are the same, showing that the file survived its trip to HDFS and is back intact.

Finally, let's look at an HDFS file listing. We create a directory first just to see how it is displayed in the listing:

```
% hadoop fs -mkdir books
```

```
% hadoop fs -ls .
```

```
Found 2 items
```

```
drwxr-xr-x - tom supergroup 0 2009-04-02 22:41 /user/tom/books
```

```
-rw-r--r-- 1 tom supergroup 118 2009-04-02 22:29 /user/tom/quangle.txt
```

The information returned is very similar to the Unix command **ls -l**, with a few minor differences. The first column shows the file mode. The second column is the replication factor of the file. Remember we set the default replication factor in the site-wide configuration to be 1, which is why we see the same value here. The entry in this column is empty for directories

because the concept of replication does not apply to them—directories are treated as metadata and stored by the namenode, not the datanodes. The third and fourth columns show the file owner and group. The fifth column is the size of the file in bytes, or zero for directories. The sixth and seventh columns are the last modified date and time. Finally, the eighth column is the absolute name of the file or directory.

Hadoop Filesystems

Hadoop has an abstract notion of filesystem, of which HDFS is just one implementation. The Java abstract class `org.apache.hadoop.fs.FileSystem` represents a filesystem in Hadoop, and there are several concrete implementations, which are described in the following table:

Table 3-1. Hadoop filesystems

Filesystem	URI scheme	Java implementation (all under <code>org.apache.hadoop</code>)	Description
Local	<i>file</i>	<code>fs.LocalFileSystem</code>	A filesystem for a locally connected disk with client-side checksums. Use <code>RawLocalFileSystem</code> for a local filesystem with no checksums. See “LocalFileSystem” on page 82 .
HDFS	<i>hdfs</i>	<code>hdfs.DistributedFileSystem</code>	Hadoop’s distributed filesystem. HDFS is designed to work efficiently in conjunction with MapReduce.
HFTP	<i>hftp</i>	<code>hdfs.HftpFileSystem</code>	A filesystem providing read-only access to HDFS over HTTP. (Despite its name, HFTP has no connection with FTP.) Often used with <i>distcp</i> (see “Parallel Copying with distcp” on page 75) to copy data between HDFS clusters running different versions.
HSFTP	<i>hsftp</i>	<code>hdfs.HsftpFileSystem</code>	A filesystem providing read-only access to HDFS over HTTPS. (Again, this has no connection with FTP.)
WebHDFS	<i>webhdfs</i>	<code>hdfs.web.WebHdfsFileSystem</code>	A filesystem providing secure read-write access to HDFS over HTTP. WebHDFS is intended as a replacement for HFTP and HSFTP.
HAR	<i>har</i>	<code>fs.HarFileSystem</code>	A filesystem layered on another filesystem for archiving files. Hadoop Archives are typically used for archiving files in HDFS to reduce the namenode’s memory usage. See “Hadoop Archives” on page 77 .
KFS (Cloud-Store)	<i>kfs</i>	<code>fs.kfs.KosmosFileSystem</code>	CloudStore (formerly Kosmos filesystem) is a distributed filesystem like HDFS or Google’s GFS, written in C++. Find more information about it at http://code.google.com/p/kosmosfs/ .
FTP	<i>ftp</i>	<code>fs.ftp.FTPFileSystem</code>	A filesystem backed by an FTP server.
S3 (native)	<i>s3n</i>	<code>fs.s3native.NativeS3FileSystem</code>	A filesystem backed by Amazon S3. See http://wiki.apache.org/hadoop/AmazonS3 .

S3 (block-based)	s3	fs.s3.S3FileSystem	A filesystem backed by Amazon S3, which stores files in blocks (much like HDFS) to overcome S3's 5 GB file size limit.
Distributed RAID	hdfs	hdfs.DistributedRaidFileSystem	A "RAID" version of HDFS designed for archival storage. For each file in HDFS, a (smaller) parity file is created, which allows the HDFS replication to be reduced from three to two, which reduces disk usage by 25% to 30% while keeping the probability of data loss the same. Distributed RAID requires that you run a <code>RaidNode</code> daemon on the cluster.
View	viewfs	viewfs.ViewFileSystem	A client-side mount table for other Hadoop filesystems. Commonly used to create mount points for federated namenodes (see " HDFS Federation " on page 47).

Hadoop provides many interfaces to its filesystems, and it generally uses the URI scheme to pick the correct filesystem instance to communicate with. For example, the filesystem shell operates with all Hadoop filesystems. To list the files in the root directory of the local filesystem, type:

```
% hadoop fs -ls file:///
```

Although it is possible to run MapReduce programs that access any of these filesystems, when you are processing large volumes of data, you should choose a distributed filesystem that has the data locality optimization, notably HDFS.

Interfaces

Hadoop is written in Java, and all Hadoop filesystem interactions are mediated through the Java API. The filesystem shell, for example, is a Java application that uses the Java **FileSystem** class to provide filesystem operations. These interfaces are most commonly used with HDFS, since the other filesystems in Hadoop typically have existing tools to access the underlying filesystem (FTP clients for FTP, S3 tools for S3, etc.), but many of them will work with any Hadoop filesystem.

HTTP

There are two ways of accessing HDFS over HTTP: directly, where the HDFS daemons serve HTTP requests to clients; and via a proxy (or proxies), which accesses HDFS on the client's behalf using the usual **DistributedFileSystem** API. The two ways are illustrated in the following Figure:

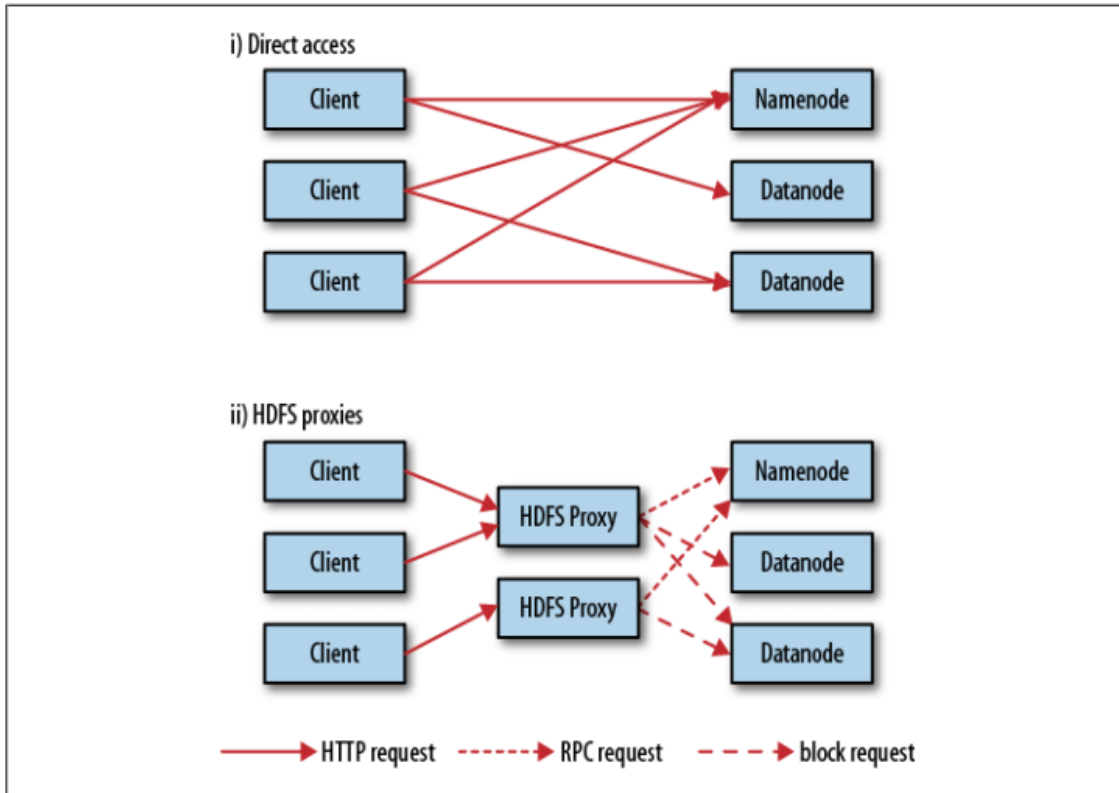


Figure 3-1. Accessing HDFS over HTTP directly and via a bank of HDFS proxies

In the first case, directory listings are served by the namenode’s embedded web server (which runs on port 50070) formatted in XML or JSON, whereas file data is streamed from datanodes by their web servers (running on port 50075).

The original direct HTTP interface (HFTP and HSFTP) was read-only, but the new WebHDFS implementation supports all filesystem operations, including Kerberos authentication. WebHDFS must be enabled by setting `dfs.webhdfs.enabled` to true, which allows you to use *webhdfs* URIs.

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. 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.

The original HDFS proxy was read-only and could be accessed by clients using the HSFTP **FileSystem** implementation (*hsftp* URIs). From release 1.0.0, there is a new proxy called **HttpFS** that has read and write capabilities and exposes the same HTTP interface as WebHDFS, so clients can access both using *webhdfs* URIs.

The HTTP REST API that WebHDFS exposes is formally defined in a specification, so it is expected that over time clients in languages other than Java will be written that use it directly.

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 access any

Hadoop filesystem). It works using the *Java Native Interface* (JNI) to call a Java filesystem client. The C API is very similar to the Java one, but it typically lags the Java one, so newer features may not be supported. You can find the generated documentation for the C API in the *libhdfs/docs/api* directory of the Hadoop distribution.

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 any Hadoop filesystem (but typically HDFS) to be mounted as a standard filesystem. You can then use Unix utilities (such as `ls` and `cat`) to interact with the filesystem, as well as POSIX libraries to access the filesystem from any programming language.

Fuse-DFS is implemented in C using *libhdfs* as the interface to HDFS. Documentation for compiling and running Fuse-DFS is located in the *src/contrib/fuse-dfs* directory of the Hadoop distribution.

The Java Interface

In this section, we dig into the Hadoop's **FileSystem** class: the API for interacting with one of Hadoop's filesystems. We should to write our code against the **FileSystem** abstract class, to retain portability across filesystems. This is very useful when testing your program, for example, because you can rapidly run tests using data stored on the local filesystem.

Reading Data from a Hadoop URL

One of the simplest ways to read a file from a Hadoop filesystem is by using a **java.net.URL** object to open a stream to read the data from. The general idiom is:

```
InputStream in = null;
try {
    in = new URL("hdfs://host/path").openStream();
    // process in
} finally {
    IOUtils.closeStream(in);
}
```

There's a little bit more work required to make Java recognize Hadoop's **hdfs** URL scheme. This is achieved by calling the **setURLStreamHandlerFactory** method on `URL` with an instance of **FsUrlStreamHandlerFactory**. This method can be called only once per JVM, so it is typically executed in a static block. This limitation means that if some other part of your program—perhaps a third-party component outside your control—sets a **URLStreamHandlerFactory**, you won't be able to use this approach for reading data from Hadoop.

This example shows a program for displaying files from Hadoop file systems on standard output, like the Unix **cat** command.

Example 3-1. Displaying files from a Hadoop filesystem on standard output using a `URLStreamHandler`

```
public class URLCat {
    static {
        URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
    }

    public static void main(String[] args) throws Exception {
        InputStream in = null;
        try {
            in = new URL(args[0]).openStream();
            IOUtils.copyBytes(in, System.out, 4096, false);
        } finally {

            IOUtils.closeStream(in);
        }
    }
}
```

We make use of the handy **IOUtils** class that comes with Hadoop for closing the stream in the **finally** clause, and also for copying bytes between the input stream and the output stream (**System.out** in this case). The last two arguments to the **copyBytes** method are the buffer size used for copying and whether to close the streams when the copy is complete. We close the input stream ourselves, and **System.out** doesn't need to be closed.

Here's a sample run:

```
% hadoop URLCat hdfs://localhost/user/tom/quangle.txt
```

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

Reading Data Using the **FileSystem** API

A file in a Hadoop filesystem is represented by a Hadoop **Path** object (and not a **java.io.File** object, since its semantics are too closely tied to the local filesystem). You can think of a **Path** as a Hadoop filesystem URI, such as ***hdfs://localhost/user/tom/quangle.txt***.

FileSystem is a general filesystem API, so the first step is to retrieve an instance for the filesystem we want to use—HDFS in this case. There are several static factory methods for getting a **FileSystem** instance:

```
public static FileSystem get(Configuration conf) throws IOException
public static FileSystem get(URI uri, Configuration conf) throws IOException
```

```
public static FileSystem get(URI uri, Configuration conf, String user) throws
IOException
```

A **Configuration** object encapsulates a client or server's configuration, which is set using configuration files read from the classpath, such as *conf/core-site.xml*. The first method returns the default filesystem. The second uses the given **URI's** scheme and authority to determine the filesystem to use, falling back to the default filesystem if no scheme is specified in the given **URI**. The third retrieves the filesystem as the given user, which is important in the context of security.

In some cases, you may want to retrieve a local filesystem instance, in which case you can use the convenience method, **getLocal()**:

```
public static LocalFileSystem getLocal(Configuration conf) throws IOException
```

With a **FileSystem** instance in hand, we invoke an **open()** method to get the input stream for a file:

```
public FSDataInputStream open(Path f) throws IOException
public abstract FSDataInputStream open(Path f, int bufferSize) throws IOException
```

The first method uses a default buffer size of 4 KB.

Putting this together, we can rewrite Example 3-1 as follows:

*Example 3-2. Displaying files from a Hadoop filesystem on standard output by using the **FileSystem** directly*

```
public class FileSystemCat {
    public static void main(String[] args) throws Exception {
        String uri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);
        InputStream in = null;
        try {
            in = fs.open(new Path(uri));
            IOUtils.copyBytes(in, System.out, 4096, false);
        } finally {
            IOUtils.closeStream(in);
        }
    }
}
```

The program runs as follows:

```
% hadoop FileSystemCat hdfs://localhost/user/tom/quangle.txt
```

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

FSDatInputStream

The **open()** method on **FileSystem** actually returns a **FSDatInputStream** rather than a standard **java.io** class. This class is a specialization of **java.io.DataInputStream** with support for random access, so you can read from any part of the stream:

```
package org.apache.hadoop.fs;

public class FSDatInputStream extends DataInputStream
    implements Seekable, PositionedReadable {
    // implementation elided
}
```

The **Seekable** interface permits seeking to a position in the file and a query method for the current offset from the start of the file (**getPos()**):

```
public interface Seekable {
    void seek(long pos) throws IOException;
    long getPos() throws IOException;
}
```

Calling **seek()** with a position that is greater than the length of the file will result in an **IOException**. Unlike the **skip()** method of **java.io.InputStream**, which positions the stream at a point later than the current position, **seek()** can move to an arbitrary, absolute position in the file.

Example 3-3 is a simple extension of Example 3-2 that writes a file to standard out twice: after writing it once, it seeks to the start of the file and streams through it once again.

Example 3-3. Displaying files from a Hadoop filesystem on standard output twice, by using seek

```
public class FileSystemDoubleCat {
    public static void main(String[] args) throws Exception {
        String uri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf); FSDatInputStream in = null;
        try {
            in = fs.open(new Path(uri));
            IOUtils.copyBytes(in, System.out, 4096, false);
            in.seek(0); // go back to the start of the file
            IOUtils.copyBytes(in, System.out, 4096, false);
        } finally {
            IOUtils.closeStream(in);
        }
    }
}
```

Here's the result of running it on a small file:

```
% hadoop FileSystemDoubleCat hdfs://localhost/user/tom/quangle.txt
```

```
On the top of the Crumpey Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
On the top of the Crumpey Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

FSDataInputStream also implements the **PositionedReadable** interface for reading parts of a file at a given offset:

```
public interface PositionedReadable {
    public int read(long position, byte[] buffer, int offset, int length) throws IOException;
    public void readFully(long position, byte[] buffer, int offset, int length) throws IOException;
    public void readFully(long position, byte[] buffer) throws IOException;
}
```

The **read()** method reads up to **length** bytes from the given **position** in the file into the **buffer** at the given **offset** in the buffer. The return value is the number of bytes actually read; callers should check this value, as it may be less than length. The **readFully()** methods will read **length** bytes into the buffer unless the end of the file is reached, in which case an **EOFException** is thrown.

All of these methods preserve the current offset in the file and are thread-safe, so they provide a convenient way to access another part of the file metadata while reading the main body of the file.

Finally, bear in mind that calling **seek()** is a relatively expensive operation and should be used sparingly. You should structure your application access patterns to rely on streaming data rather than performing a large number of seeks.

Writing Data

The **FileSystem** class has a number of methods for creating a file. The simplest is the method that takes a **Path** object for the file to be created and returns an output stream to write to:

```
public FSDataOutputStream create(Path f) throws IOException
```

There are overloaded versions of this method that allow you to specify whether to forcibly overwrite existing files, the replication factor of the file, the buffer size to use when writing the file, the block size for the file, and file permissions.

There's also an overloaded method for passing a callback interface, **Progressable**, so your application can be notified of the progress of the data being written to the datanodes:

```
package org.apache.hadoop.util;

public interface
Progressable {

    public void progress();

}
```

As an alternative to creating a new file, you can append to an existing file using the **append()** method:

```
public FSDataOutputStream append(Path f) throws IOException
```

The append operation allows a single writer to modify an already written file by opening it and writing data from the final offset in the file. With this API, applications that produce unbounded files, such as logfiles, can write to an existing file after having closed it. The append operation is optional and not implemented by all Hadoop file systems.

Example 3-4 shows how to copy a local file to a Hadoop filesystem. We illustrate progress by printing a period every time the **progress()** method is called by Hadoop, which is after each 64

KB packet of data is written to the datanode pipeline.

Example 3-4. Copying a local file to a Hadoop filesystem

```
public class FileCopyWithProgress {
    public static void main(String[] args) throws Exception {
        String localSrc = args[0];
        String dst = args[1];

        InputStream in = new BufferedInputStream(new FileInputStream(localSrc));

        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(dst), conf);
        OutputStream out = fs.create(new Path(dst), new Progressable() {
            public void progress() {
                System.out.print(".");
            }
        });

        IOUtils.copyBytes(in, out, 4096, true);
    }
}
```

Typical usage:

```
% hadoop FileCopyWithProgress input/docs/1400-8.txt
hdfs://localhost/user/tom/ 1400-8.txt
```

.....

Currently, none of the other Hadoop filesystems call **progress()** during writes. Progress is important in MapReduce applications.

FSDataOutputStream

The **create()** method on **FileSystem** returns an **FSDataOutputStream**, which, like **FSDataInputStream**, has a method for querying the current position in the file:

```
package org.apache.hadoop.fs;

public class FSDataOutputStream extends DataOutputStream implements Syncable {
    public long getPos() throws IOException {

        // implementation elided
    }

    // implementation elided
}
}
```

However, unlike **FSDataInputStream**, **FSDataOutputStream** does not permit seeking. This is because HDFS allows only sequential writes to an open file or appends to an already written file. In other words, there is no support for writing to anywhere other than the end of the file, so there is no value in being able to seek while writing.

Directories

FileSystem provides a method to create a directory:

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

This method creates all of the necessary parent directories if they don't already exist, just like the **java.io.File's mkdirs()** method. It returns **true** if the directory (and all parent directories) was (were) successfully created.

Often, you don't need to explicitly create a directory, because writing a file by calling **create()** will automatically create any parent directories.

Querying the Filesystem

File metadata: **FileStatus**

An important feature of any filesystem is the ability to navigate its directory structure and retrieve information about the files and directories that it stores. The **FileStatus** class encapsulates filesystem metadata for files and directories, including file length, block size, replication, modification time, ownership, and permission information.

The method **getFileStatus()** on **FileSystem** provides a way of getting a **FileStatus** object for a single file or directory. Example 3-5 shows an example of its use.

Example 3-5. Demonstrating file status information

```
public class ShowFileStatusTest {

    private MiniDFSCluster cluster; // use an in-process HDFS cluster for testing
    private FileSystem fs;

    @Before
    public void setUp() throws IOException {
        Configuration conf = new Configuration();
        if (System.getProperty("test.build.data") == null) {
            System.setProperty("test.build.data", "/tmp");
        }
        cluster = new MiniDFSCluster(conf, 1, true, null);
        fs = cluster.getFileSystem();
        OutputStream out = fs.create(new Path("/dir/file"));
        out.write("content".getBytes("UTF-8"));
        out.close();
    }

    @After
    public void tearDown() throws IOException {
        if (fs != null) { fs.close(); }
        if (cluster != null) { cluster.shutdown(); }
    }

    @Test(expected = FileNotFoundException.class)
    public void throwsFileNotFoundExceptionForNonExistentFile() throws IOException {
        fs.getFileStatus(new Path("no-such-file"));
    }
}
```

```

@Test
public void fileStatusForFile() throws IOException {
    Path file = new Path("/dir/file");
    FileStatus stat = fs.getFileStatus(file);
    assertThat(stat.getPath().toUri().getPath(), is("/dir/file"));
    assertThat(stat.isDir(), is(false));
    assertThat(stat.getLen(), is(7L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 1));
    assertThat(stat.getBlockSize(), is(64 * 1024 * 1024L));
    assertThat(stat.getOwner(), is("tom"));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rw-r--r--"));
}

```

```

@Test
public void fileStatusForDirectory() throws IOException {
    Path dir = new Path("/dir");
    FileStatus stat = fs.getFileStatus(dir);
    assertThat(stat.getPath().toUri().getPath(), is("/dir"));
    assertThat(stat.isDir(), is(true));
    assertThat(stat.getLen(), is(0L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 0));
    assertThat(stat.getBlockSize(), is(0L));
    assertThat(stat.getOwner(), is("tom"));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rwxr-xr-x"));
}
}

```

If no file or directory exists, a **FileNotFoundException** is thrown. However, if you are interested only in the existence of a file or directory, the **exists()** method on **FileSystem** is more convenient:

```
public boolean exists(Path f) throws IOException
```

Listing files

Finding information on a single file or directory is useful, but you also often need to be able to list the contents of a directory. That's what **FileSystem's listStatus()** methods are for:

```

public FileStatus[] listStatus(Path f) throws IOException
public FileStatus[] listStatus(Path f, PathFilter filter) throws IOException
public FileStatus[] listStatus(Path[] files) throws IOException
public FileStatus[] listStatus(Path[] files, PathFilter filter) throws IOException

```

When the argument is a file, the simplest variant returns an array of **FileStatus** objects of length 1. When the argument is a directory, it returns zero or more **FileStatus** objects representing the files and directories contained in the directory.

Overloaded variants allow a **PathFilter** to be supplied to restrict the files and directories to match. Finally, if you specify an array of paths, the result is a shortcut for calling the equivalent

single-path **listStatus** method for each path in turn and accumulating the **FileStatus** object arrays in a single array. This can be useful for building up lists of input files to process from distinct parts of the filesystem tree. Example 3-6 is a simple demonstration of this idea. Note the use of **stat2Paths()** in **FileUtil** for turning an array of **FileStatus** objects to an array of **Path** objects.

Example 3-6. Showing the file statuses for a collection of paths in a Hadoop filesystem

```
public class ListStatus {

    public static void main(String[] args) throws Exception {
        String uri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);

        Path[] paths = new Path[args.length];
        for (int i = 0; i < paths.length; i++) {
            paths[i] = new Path(args[i]);
        }

        FileStatus[] status = fs.listStatus(paths);
        Path[] listedPaths = FileUtil.stat2Paths(status);
        for (Path p : listedPaths) {
            System.out.println(p);
        }
    }
}
```

We can use this program to find the union of directory listings for a collection of paths:

```
% hadoop ListStatus hdfs://localhost/ hdfs://localhost/user/tom

hdfs://localhost/user
hdfs://localhost/user/tom/books
hdfs://localhost/user/tom/quangle.txt
```

PathFilter

PathFilter is the equivalent of **java.io.FileFilter** for **Path** objects rather than **File** objects.

Example 3-7 shows a **PathFilter** for excluding paths that match a regular expression.

Example 3-7. A PathFilter for excluding paths that match a regular expression

```
public class RegexExcludePathFilter implements PathFilter {
    private final String regex;

    public RegexExcludePathFilter(String regex) {
        this.regex = regex;
    }

    public boolean accept(Path path) {
        return !path.toString().matches(regex);
    }
}
```

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`.

Filters can act only on a file's name, as represented by a **Path**. They can't use a file's properties, such as creation time, as the basis of the filter.

Deleting Data

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

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

If **f** is a file or an empty directory, the value of **recursive** is ignored. A nonempty directory is deleted, along with its contents, only if **recursive** is true (otherwise, an **IOException** is thrown).

Data Flow

Anatomy of a File Read

To get an idea of how data flows between the client interacting with HDFS, the namenode, and the datanodes, consider Figure 3-2, which shows the main sequence of events when reading a file. 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. 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.

The **DistributedFileSystem** returns an **FSDatInputStream** (an input stream that supports file seeks) to the client for it to read data from. **FSDatInputStream** 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 **FSDatInputStream** (step 6).

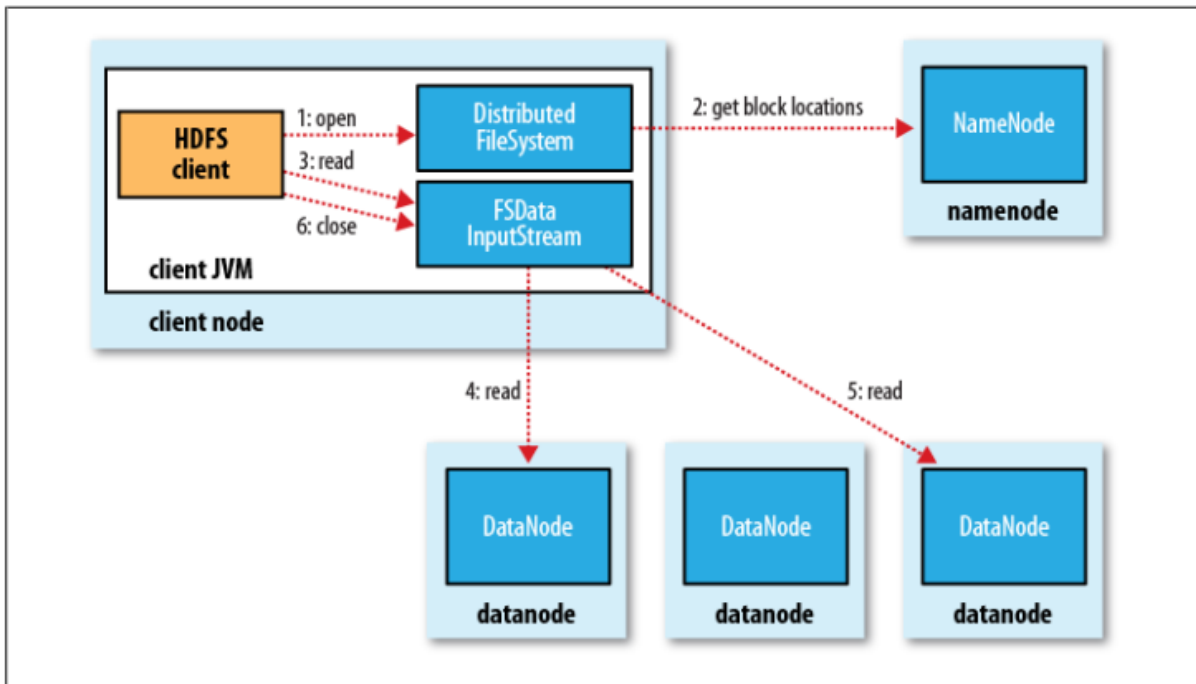


Figure 3-2. A client reading data from HDFS

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 **DFSInputStream** also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, it is reported to the namenode before the **DFSInputStream** attempts to read a replica of the block from another datanode.

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.

Anatomy of a File Write

Next we'll look at how files are written to HDFS.

We're going to consider the case of creating a new file, writing data to it, then closing the file. See Figure 3-4.

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.

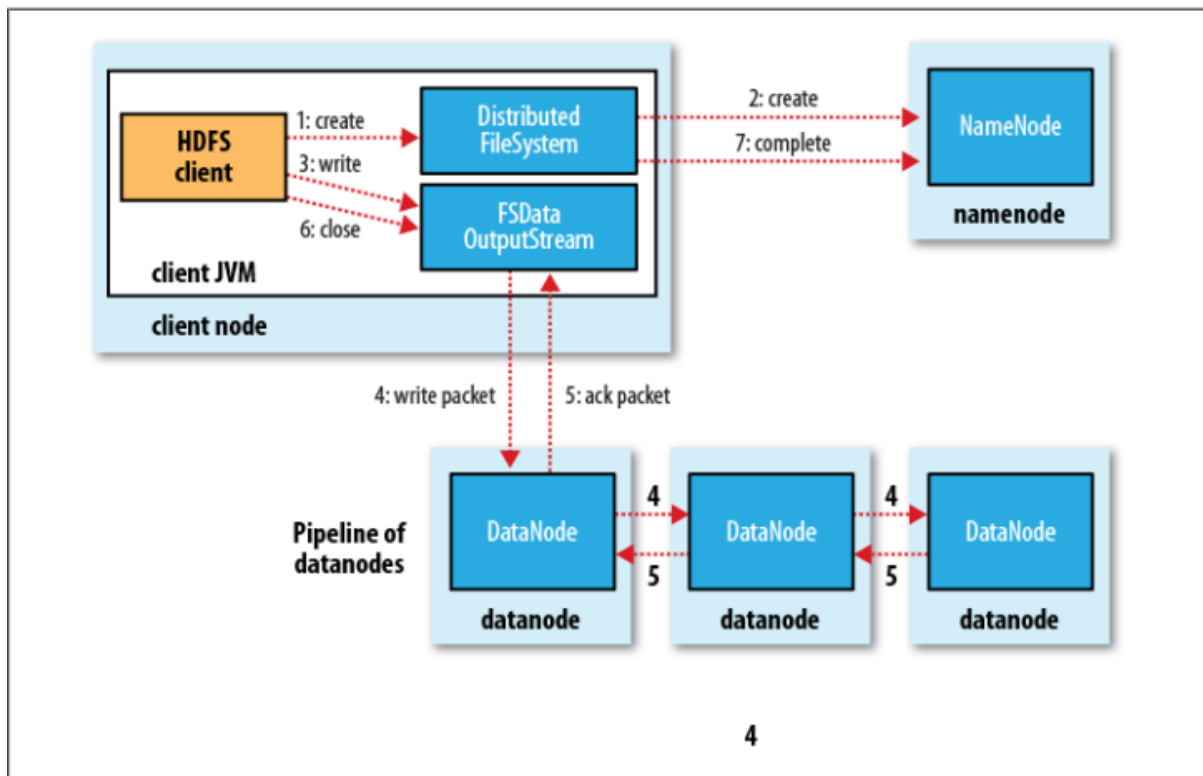


Figure 3-4. A client writing data to HDFS

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 a 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 the remainder of the block's data is written to the two 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.replication.min** replicas (which default 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 acknowledgements 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 **Data Streamer** asking for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

Coherency Model

A coherency model for a filesystem describes the data visibility of reads and writes for a file. After creating a file, it is visible in the filesystem namespace:

```
Path p = new Path("p");
fs.create(p);
assertThat(fs.exists(p), is(true));
```

However, any content written to the file is not guaranteed to be visible, even if the stream is flushed. So the file appears to have a length of zero:

```
Path p = new Path("p");
OutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();
assertThat(fs.getFileStatus(p).getLen(), is(0L));
```

Once more than a block's worth of data has been written, the first block will be visible to new readers. This is true of subsequent blocks, too: it is always the current block being written that is not visible to other readers.

HDFS provides a method for forcing all buffers to be synchronized to the datanodes via the **sync()** method on **FSDDataOutputStream**. After a successful return from **sync()**, 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");
FSDDataOutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();
out.sync();
assertThat(fs.getFileStatus(p).getLen(), is(((long) "content".length())));
```

This behavior 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 **sync()**, 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())));
```

Consequences for application design

This coherency model has implications for the way you design applications. With no calls to **sync()**, you should be prepared to lose up to a block of data in the event of client or system failure. For many applications, this is unacceptable, so you should call **sync()** at suitable points, such as after writing a certain number of records or number of bytes. Though the **sync()** operation is designed to not unduly tax HDFS, it does have some overhead, so there is a trade-off between data robustness and throughput. What constitutes an acceptable trade-off is application-dependent, and suitable values can be selected after measuring your application's performance with different **sync()** frequencies.

Parallel Copying with **distcp**

The HDFS access patterns that we have seen so far focus on single-threaded access. It's possible to act on a collection of files but for efficient parallel processing of these files, you would have to write a program yourself. Hadoop comes with a useful program called **distcp** for copying large amounts of data to and from Hadoop filesystems in parallel.

The canonical use case for **distcp** is for transferring data between two HDFS clusters. If the clusters are running identical versions of Hadoop, the **hdfs** scheme is appropriate:

```
% hadoop distcp hdfs://namenode1/foo hdfs://namenode2/bar
```

This will copy the **/foo** directory (and its contents) from the first cluster to the **/bar** directory on the second cluster, so the second cluster ends up with the directory structure **/bar/foo**. If **/bar** doesn't exist, it will be created first. You can specify multiple source paths, and all will be copied to the destination. Source paths must be absolute.

By default, **distcp** will skip files that already exist in the destination, but they can be overwritten by supplying the **-overwrite** option. You can also update only the files that have changed using the **-update** option.

There are more options to control the behavior of **distcp**, including ones to preserve file attributes, ignore failures, and limit the number of files or total data copied. Run it with no options to see the usage instructions.

distcp is implemented as a MapReduce job where the work of copying is done by the maps that run in parallel across the cluster. There are no reducers. Each file is copied by a single map, and **distcp** tries to give each map approximately the same amount of data by bucketing files into roughly equal allocations.

The number of maps is decided as follows. Because it's a good idea to get each map to copy a reasonable amount of data to minimize overheads in task setup, each map copies at least 256 MB

(unless the total size of the input is less, in which case one map handles it all). For example, 1 GB of files will be given four map tasks. When the data size is very large, it becomes necessary to limit the number of maps in order to limit bandwidth and cluster utilization. By default, the maximum number of maps is 20 per (tasktracker) cluster node. For example, copying 1,000 GB of files to a 100-node cluster will allocate 2,000 maps (20 per node), so each will copy 512 MB on average. This can be reduced by specifying the **-m** argument to **distcp**. For example, **-m 1000** would allocate 1,000 maps, each copying 1 GB on average.

When you try to use **distcp** between two HDFS clusters that are running different versions, the copy will fail if you use the **hdfs** protocol because the RPC systems are incompatible. To remedy this, you can use the read-only HTTP-based HFTP filesystem to read from the source. The job must run on the destination cluster so that the HDFS RPC versions are compatible. To repeat the previous example using HFTP:

```
% hadoop distcp hftp://namenode1:50070/foo hdfs://namenode2/bar
```

Note that you need to specify the namenode's web port in the source URI. This is determined by the **dfs.http.address** property, which defaults to 50070.

Using the newer **webhdfs** protocol (which replaces **hftp**), it is possible to use HTTP for both the source and destination clusters without hitting any wire incompatibility problems.

```
% hadoop distcp webhdfs://namenode1:50070/foo webhdfs://namenode2:50070/bar
```

Another variant is to use an HDFS HTTP proxy as the **distcp** source or destination, which has the advantage of being able to set firewall and bandwidth controls.

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. Going back to the 1,000 GB 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.

However, it's not always possible to prevent a cluster from becoming unbalanced. Perhaps you want to limit the number of maps so that some of the nodes can be used by other jobs. In this case, you can use the **balancer** tool to subsequently even out the block distribution across the cluster.

Hadoop Archives

A Hadoop Archive is created from a collection of files using the **archive** tool. The tool runs a MapReduce job to process the input files in parallel, so to run it, you need a running MapReduce cluster to use it. Here are some files in HDFS that we would like to archive:

```
% hadoop fs -lsr /my/files
-rw-r--r--          1 tom  supergroup      1 2009-04-09 19:13 /my/files/a
drwxr-xr-x          - tom  supergroup      0 2009-04-09 19:13 /my/files/dir
-rw-r--r--          1 tom  supergroup      1 2009-04-09 19:13 /my/files/dir/b
```

Now we can run the archive command:

```
% hadoop archive -archiveName files.har /my/files /my
```

The first option is the name of the archive, here **files.har**. HAR files always have a **.har** extension, which is mandatory for reasons we shall see later. Next come the files to put in the archive. Here we are archiving only one source tree, the files in **/my/files** in HDFS, but the tool accepts multiple source trees. The final argument is the output directory for the HAR file. Let's see what the archive has created:

```
% hadoop fs -ls /my
```

```
Found 2 items
```

```
drwxr-xr-x - tom supergroup 0 2009-04-09 19:13 /my/f
drwxr-xr-x - tom supergroup 0 2009-04-09 19:13 /my/files.har
```

```
% hadoop fs -ls /my/files.har
```

```
Found 3 items
```

```
-rw-r--r-- 10 tom supergroup 165 2009-04-09 19:13 /my/files.har/_index
-rw-r--r-- 10 tom supergroup 23 2009-04-09 19:13 /my/files.har/_masterindex
-rw-r--r-- 1 tom supergroup 2 2009-04-09 19:13 /my/files.har/part-0
```

The directory listing shows what a HAR file is made of: two index files and a collection of part files. The part files contain the contents of a number of the original files concatenated together, and the indexes make it possible to look up the part file that an archived file is contained in, as well as its offset and length. All these details are hidden from the application, however, which uses the **har** URI scheme to interact with HAR files, using a HAR filesystem that is layered on top of the underlying filesystem (HDFS in this case). The following command recursively lists the files in the archive:

```
% hadoop fs -lsr har:///my/files.har
```

```
drw-r--r-- - tom supergroup 0 2009-04-09 19:13 /my/files.har/my
drw-r--r-- - tom supergroup 0 2009-04-09 19:13 /my/files.har/my/files
-rw-r--r-- 10 tom supergroup 1 2009-04-09 19:13 /my/files.har/my/files/a
drw-r--r-- - tom supergroup 0 2009-04-09 19:13 /my/files.har/my/files/dir
-rw-r--r-- 10 tom supergroup 1 2009-04-09 19:13 /my/files.har/my/files/dir/b
```

This is quite straightforward when the filesystem that the HAR file is on is the default filesystem. On the other hand, if you want to refer to a HAR file on a different filesystem, you need to use a different form of the path URI. These two commands have the same effect, for example:

```
% hadoop fs -lsr har:///my/files.har/my/files/dir
```

```
% hadoop fs -lsr har://hdfs-localhost:8020/my/files.har/my/files/dir
```

Notice in the second form that the scheme is still **har** to signify a HAR filesystem, but the authority is **hdfs** to specify the underlying filesystem's scheme, followed by a dash and the HDFS host (localhost) and port (8020). We can now see why HAR files must have a **.har** extension. The HAR filesystem translates the **har** URI into a URI for the underlying filesystem by looking at the authority and path up to and including the component with the **.har** extension. In this case, it is **hdfs://localhost:8020/my/files.har**. The remaining part of the path is the path of the file in the archive: **/my/files/dir**.

To delete a HAR file, you need to use the recursive form of delete because from the underlying filesystem's point of view, the HAR file is a directory:

```
% hadoop fs -rmr /my/files.har
```

```
*****
```




BMS INSTITUTE OF TECHNOLOGY & MANAGEMENT

Yelahanka, Bangalore-64

Department of MCA

Big Data Analytics – Lecture Notes

Module 5

Note: Parts of the notes in grey colour are just to understand the structure and not for strictly memorizing to present in exam.

What is MapReduce?

MapReduce is a programming model for data processing. Hadoop can run MapReduce programs written in various languages. Same program can be expressed in Java, Ruby, Python, and C++. MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal.

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 it is semi-structured and record-oriented.

Data Format

The data we will use is from the National Climatic Data Center. The data is stored using a line-oriented ASCII format, in which each line is a record. The format supports a rich set of meteorological elements, many of which are optional or with variable data lengths. For simplicity, we focus on the basic elements, such as temperature, which are always present and are of fixed width.

Example 2-1 shows a sample line with some of the salient fields highlighted. The line has been split into multiple lines to show each field; in the real file, fields are packed into one line with no delimiters.

Example 2-1. Format of a National Climate Data Center record

```
0057  
332130 # USAF weather station identifier  
999999 # WBAN weather station identifier
```

```

19500101 # observation date
0300     # observation time
4
+51317   # latitude (degrees x 1000)
+028783  # longitude (degrees x 1000)
FM-12
+0171    # elevation (meters)
99999
V020
320      # wind direction (degrees)
1        # quality code
N
0072
1
00450    # sky ceiling height (meters)
1        # quality code
C
N
010000   # visibility distance (meters)
1        # quality code
N
9
-0128    # air temperature (degrees Celsius x 10)
1        # quality code
-0139    # dew point temperature (degrees Celsius x 10)
1        # quality code
10268    # atmospheric pressure (hectopascals x 10)
1        # quality code

```

Datafiles are organized by date and weather station. There is a directory for each year from 1901 to 2001, each containing a gzipped file for each weather station with its readings for that year. For example, here are the first entries for 1990:

```

% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz

```

Since there are tens of thousands of weather stations, the whole dataset is made up of a large number of relatively small files. It's generally easier and more efficient to process a smaller number of relatively large files, so the data was preprocessed so that each year's readings were concatenated into a single file.

Analyzing the Data with Unix Tools

What's the highest recorded global temperature for each year in the dataset? We will answer this first without using Hadoop, as this information will provide a performance baseline and a useful means to check our results. The classic tool for processing line-oriented data is **awk**. If we write a script, the script loops through the compressed year files, first printing the year, and then processing each file using **awk**. The **awk** script extracts two fields from the data: the air temperature and the quality code. Next, a test is applied to see whether the temperature is valid (the value 9999 signifies a missing value in the NCDC dataset) and whether the quality code indicates that the reading is not suspect or erroneous. If the reading is OK, the value is compared with the maximum value seen so far, which is updated if a new maximum is found. The execution comes to an end after all the lines in the file have been processed, and it prints the maximum value.

The temperature values in the source file are scaled by a factor of 10, so this works out as a maximum temperature of 31.7°C for 1901. The complete run for the century will take 42 minutes in one run on a single EC2 High-CPU Extra Large Instance.

To speed up the processing, we need to run parts of the program in parallel. In theory, this is straightforward: we could process different years in different processes, using all the available hardware threads on a machine. There are a few problems with this, however.

First, dividing the work into equal-size pieces isn't always easy or obvious. In this case, the file size for different years varies widely, so some processes will finish much earlier than others. Even if they pick up further work, the whole run is dominated by the longest file. A better approach, although one that requires more work, is to split the input into fixed-size chunks and assign each chunk to a process.

Second, combining the results from independent processes may need further processing. In this case, the result for each year is independent of other years and may be combined by concatenating all the results and sorting by year. If using the fixed-size chunk approach, the combination is more delicate. For this example, data for a particular year will typically be split into several chunks, each processed independently. We'll end up with the maximum temperature for each chunk, so the final step is to look for the highest of these maximums for each year.

Third, you are still limited by the processing capacity of a single machine. If the best time you can achieve is 20 minutes with the number of processors you have, then that's it. You can't make it go faster. Also, some datasets grow beyond the capacity of a single machine. When we start using multiple machines, a whole host of other factors come into play, mainly falling into the category of **coordination and reliability**. Who runs the overall job? How do we deal with failed processes?

So, although it's feasible to parallelize the processing, in practice it's messy. Using a framework like Hadoop to take care of these issues is a great help.

Analyzing the Data with Hadoop

To take advantage of the parallel processing that Hadoop provides, we need to express our query as a MapReduce job. After some local, small-scale testing, we will be able to run it on a cluster of machines.

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.

The input to our map phase is the raw NCDC data. We choose a text input format that gives us each line in the dataset as a text value. The key is the offset of the beginning of the line from the beginning of the file, but as we have no need for this, we ignore it. Our map function is simple. We pull out the year and the air temperature because these are the only fields we are interested in. In this case, the map function is just a data preparation phase, setting up the data in such a way that the reducer function can do its work on it: finding the maximum temperature for each year. The map function is also a good place to drop bad records: here we filter out temperatures that are missing, suspect, or erroneous.

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+9999999999...
0043011990999991950051512004...9999999N9+00221+9999999999...
0043011990999991950051518004...9999999N9-00111+9999999999...
0043012650999991949032412004...0500001N9+01111+9999999999...
0043012650999991949032418004...0500001N9+00781+9999999999...
```

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

```
(0, 0067011990999991950051507004...9999999N9+00001+9999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+9999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+9999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+9999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+9999999999...)
```

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)

This is the final output: the maximum global temperature recorded in each year.

The whole data flow is illustrated in Figure 2-1. At the bottom of the diagram is a Unix pipeline, which mimics the whole MapReduce flow.

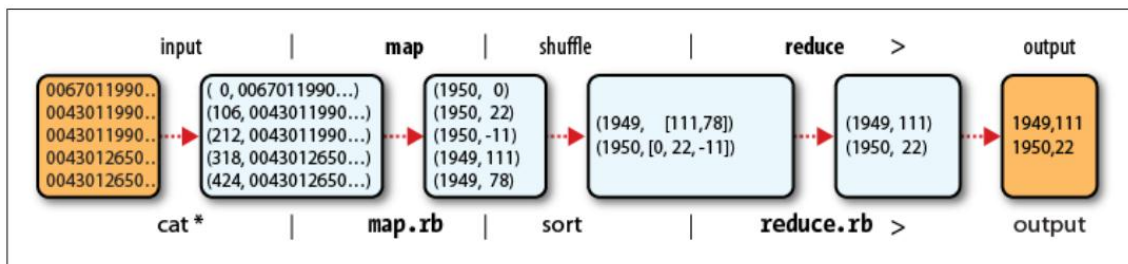


Figure 2-1. MapReduce logical data flow

Java MapReduce

Having run through how the MapReduce program works, the next step is to express it in code. We need three things: a map function, a reduce function, and some code to run the job. The map function is represented by the **Mapper** class, which declares an abstract **map()** method. Example 2-3 shows the implementation of our map method.

Example 2-3. Mapper for the maximum temperature example

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {
    private static final int MISSING = 9999;
    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        String line = value.toString();
        String year = line.substring(15, 19);
        int airTemperature;
        if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
            airTemperature = Integer.parseInt(line.substring(88, 92));
        } else {
            airTemperature = Integer.parseInt(line.substring(87, 92));
        }
    }
}
```

```

}
String quality = line.substring(92, 93);
if (airTemperature != MISSING && quality.matches("[01459]")) {
    context.write(new Text(year), new IntWritable(airTemperature));
}
}
}
}

```

The **Mapper** class is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function. For the present example, the input key is a long integer offset, the input value is a line of text, the output key is a year, and the output value is an air temperature (an integer). Rather than use built-in Java types, Hadoop provides its own set of basic types that are optimized for network serialization. These are found in the **org.apache.hadoop.io** package. Here we use **LongWritable**, which corresponds to a Java **Long**, **Text** (like Java String), and **IntWritable** (like Java Integer).

The **map()** method is passed a key and a value. We convert the **Text** value containing the line of input into a Java String, then use its **substring()** method to extract the columns we are interested in.

The **map()** method also provides an instance of **Context** to write the output to. In this case, we write the year as a **Text** object (since we are just using it as a key), and the temperature is wrapped in an **IntWritable**. We write an output record only if the temperature is present and the quality code indicates the temperature reading is OK.

The reduce function is similarly defined using a **Reducer**, as illustrated in Example 2-4.

Example 2-4. Reducer for the maximum temperature example

```

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

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

        int maxValue = Integer.MIN_VALUE;
        for (IntWritable value : values) {
            maxValue = Math.max(maxValue, value.get());
        }
        context.write(key, new IntWritable(maxValue));
    }
}

```

Again, four formal type parameters are used to specify the input and output types, this time for

the **reduce** function. The input types of the reduce function must match the output types of the map function: **Text** and **IntWritable**. And in this case, the output types of the reduce function are **Text** and **IntWritable**, for a year and its maximum temperature, which we find by iterating through the temperatures and comparing each with a record of the highest found so far.

The third piece of code runs the MapReduce job (Example 2-5).

Example 2-5. Application to find the maximum temperature in the weather dataset

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MaxTemperature {

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: MaxTemperature <input path> <output path>");
            System.exit(-1);
        }

        Job job = new Job();
        job.setJarByClass(MaxTemperature.class);
        job.setJobName("Max temperature");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(MaxTemperatureMapper.class);
        job.setReducerClass(MaxTemperatureReducer.class);

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

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

A Job object forms the specification of the job and gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster).

Having constructed a Job object, we specify the input and output paths. An input path is specified by calling the static **addInputPath()** method on **FileInputFormat**, and it can be a single file, a directory (in which case the input forms all the files in that directory), or a file pattern. As the name suggests, **addInputPath()** can be called more than once to use input from multiple paths.

The output path (of which there is only one) is specified by the static **setOutput Path()** method on **FileOutputFormat**. It specifies a directory where the output files from the reducer

functions are written. The directory shouldn't exist before running the job because Hadoop will complain and not run the job. This precaution is to prevent data loss.

Next, we specify the map and reduce types to use via the **setMapperClass()** and **setReducerClass()** methods.

The **setOutputKeyClass()** and **setOutputValueClass()** methods control the output types for the map and the reduce functions, which are often the same, as they are in our case. If they are different, the map output types can be set using the methods **setMapOutputKeyClass()** and **setMapOutputValueClass()**.

The input types are controlled via the input format, which we have not explicitly set because we are using the default **TextInputFormat**.

After setting the classes that define the map and reduce functions, we are ready to run the job. The **waitForCompletion()** method on Job submits the job and waits for it to finish. The method's Boolean argument is a verbose flag, so in this case the job writes information about its progress to the console.

The return value of the **waitForCompletion()** method is a Boolean indicating success (true) or failure (false), which we translate into the program's exit code of 0 or 1.

Scaling Out

You've seen how MapReduce works for small inputs; now it's time to look at the data flow for large inputs. For simplicity, the examples so far have used files on the local filesystem. To scale out, we need to store the data in a distributed filesystem, typically HDFS. To allow Hadoop to move the MapReduce computation to each machine hosting a part of the data, the process is as follows:

Data Flow

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

There are two types of nodes that control the job execution process: a **jobtracker** and a number of **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.

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, 64 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.

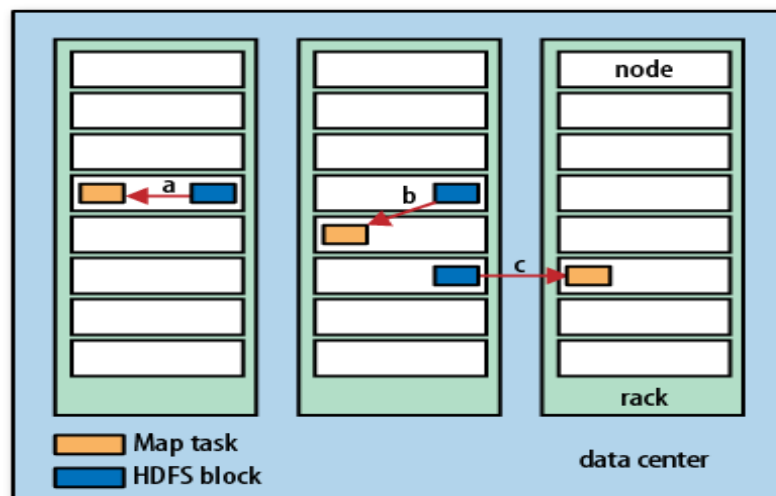


Figure 2-2. Data-local (a), rack-local (b), and off-rack (c) map tasks

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 because 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. 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. 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.

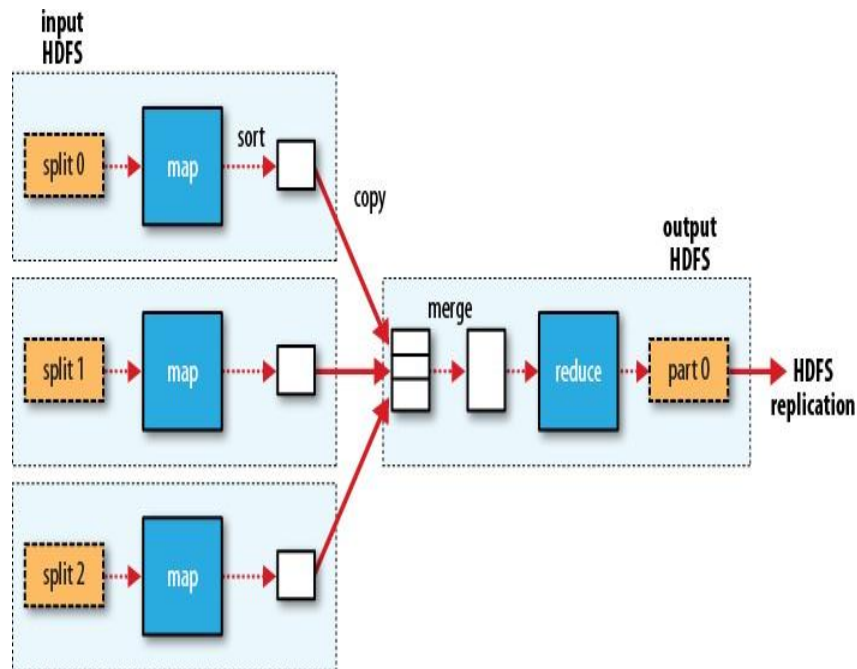


Figure 2-3. MapReduce data flow with a single reduce task

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

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.

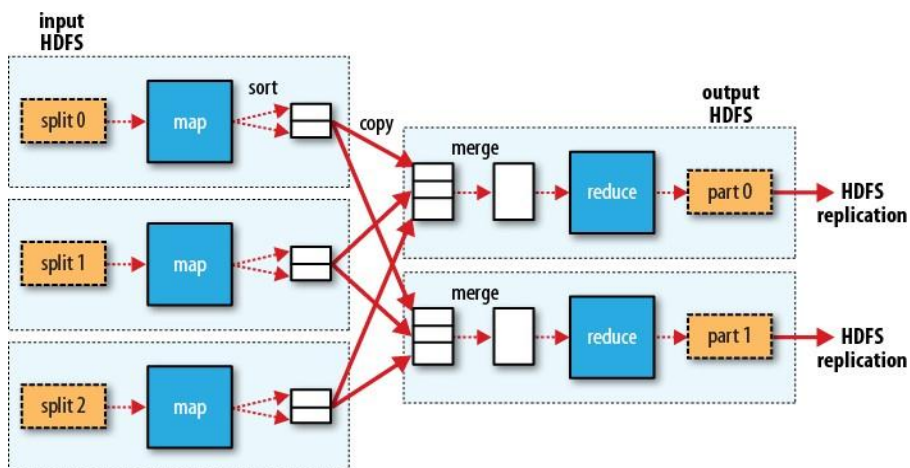


Figure 2-4. MapReduce data flow with multiple reduce tasks

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. In this case, the only off-node data transfer is when the map tasks write to HDFS. This is shown in Fig. 2-5.

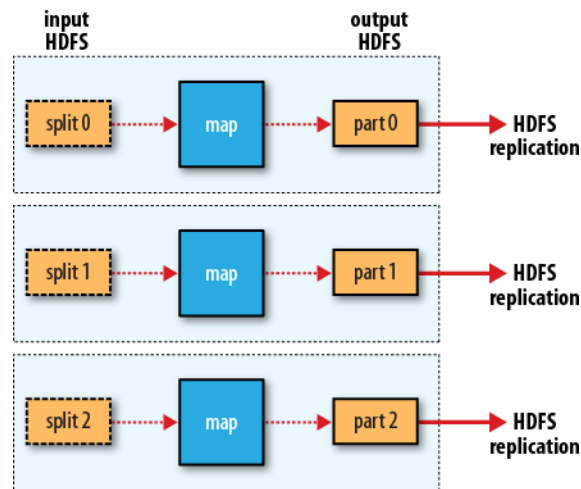


Figure 2-5. MapReduce data flow with no reduce tasks

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])

and the reduce would produce the same output as before. More succinctly, we may express the function calls on the temperature values in this case as follows:

$max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25$

Not all functions possess this property. For example, if we were calculating mean temperatures, we couldn't use the mean as our combiner function, because:

$mean(0, 20, 10, 25, 15) = 14$

but:

$mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15$

The combiner function doesn't replace the reduce function. 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

Going back to the Java MapReduce program, the combiner function is defined using the Reducer class, and for this application, it is the same implementation as the reducer function in **MaxTemperatureReducer**. The only change we need to make is to set the combiner class on the **Job**.

Example 2-7. Application to find the maximum temperature, using a combiner function for efficiency

```
public class MaxTemperatureWithCombiner {

    public static void main(String[] args) throws
        Exception { if (args.length != 2) {

        System.err.println("Usage: MaxTemperatureWithCombiner <input path> " +
            "<output path>");

        System.exit(-1);

        }

        Job job = new Job();
        job.setJarByClass(MaxTemperatureWithCombiner.class);
        job.setJobName("Max temperature");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(MaxTemperatureMapper.class);
        job.setCombinerClass(MaxTemperatureReducer.class);
        job.setReducerClass(MaxTemperatureReducer.class);
```

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

Running a Distributed MapReduce Job

The same program will run, without alteration, on a full dataset. This is the point of MapReduce: it scales to the size of your data and the size of your hardware. Here's one data point: on a 10-node EC2 cluster running High-CPU Extra Large Instances, the program took six minutes to run.

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.

Streaming is naturally suited for text processing. Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output. A map output key-value pair is written as a single tab-delimited line. Input to the reduce function is in the same format—a tab-separated key-value pair—passed over standard input. The reduce function reads lines from standard input, which the framework guarantees are sorted by key, and writes its results to standard output.

Hadoop Pipes

Hadoop Pipes is the name of the C++ interface to Hadoop MapReduce. Unlike Streaming, which uses standard input and output to communicate with the map and reduce code, Pipes uses sockets as the channel over which the tasktracker communicates with the process running the C++ map or reduce function. JNI is not used.

We'll rewrite this chapter's temperature example in C++, and then we'll see how to run it using Pipes. Example 2-12 shows the source code for the map and reduce functions in C++.

Example 2-12. Maximum temperature in C++

```
#include <algorithm>
#include <limits>
#include <stdint.h>
#include <string>

#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"
```

```

class MaxTemperatureMapper : public HadoopPipes::Mapper {
public:
    MaxTemperatureMapper(HadoopPipes::TaskContext& context) {
    }
    void map(HadoopPipes::MapContext& context) { std::string line = context.getInputValue();
        std::string year = line.substr(15, 4); std::string airTemperature = line.substr(87, 5);
        std::string q = line.substr(92, 1);
        if (airTemperature != "+9999" &&
            (q == "0" || q == "1" || q == "4" || q == "5" || q == "9")) {
            context.emit(year, airTemperature);
        }
    }
};

class MapTemperatureReducer : public HadoopPipes::Reducer {
public:
    MapTemperatureReducer(HadoopPipes::TaskContext& context) {
    }
    void reduce(HadoopPipes::ReduceContext& context) {
        int maxValue = INT_MIN;
        while (context.nextValue()) {

            maxValue = std::max(maxValue, HadoopUtils::toInt(context.getInputValue()));
        }
        context.emit(context.getInputKey(), HadoopUtils::toString(maxValue));
    }
};

int main(int argc, char *argv[]) {
    return HadoopPipes::runTask(HadoopPipes::TemplateFactory<MaxTemperatureMapper,
        MapTemperatureReducer>());
}

```

The application links against the Hadoop C++ library, which is a thin wrapper for communicating with the tasktracker child process. The map and reduce functions are defined by extending the Mapper and Reducer classes defined in the **HadoopPipes** namespace and providing implementations of the **map()** and **reduce()** methods in each case. These methods take a context object (of type **MapContext** or **ReduceContext**), which provides the means for reading input and writing output, as well as accessing job configuration information via the **JobConf** class. The processing in this example is very similar to the Java equivalent.

Unlike the Java interface, keys and values in the C++ interface are byte buffers represented as Standard Template Library (STL) strings. This makes the interface simpler, although it does put a slightly greater burden on the application developer, who has to convert to and from richer domain-level types. This is evident in **MapTemperatureReducer**, where we have to convert the input value into an integer (using a convenience method in **HadoopUtils**) and then the maximum value back into a string before it's written out. In some cases, we can skip the conversion, such as in **MaxTemperatureMapper**, where the **airTemperature** value is never converted to an integer because it is never processed as a number in the **map()** method.

The **main()** method is the application entry point. It calls **HadoopPipes::runTask**, which connects to the Java parent process and marshals data to and from the **Mapper** or **Reducer**. The **runTask()** method is passed a Factory so that it can create instances of the **Mapper** or **Reducer**. Which one it creates is controlled by the Java parent over the socket connection. There are overloaded template factory methods for setting a combiner, partitioner, record

reader, or recordwriter.

Compiling and Running

Now we can compile and link our program using the makefile in Example 2-13.

Example 2-13. Makefile for C++ MapReduce program

```
CC = g++
CPPFLAGS = -m32 -I$(HADOOP_INSTALL)/c++/$(PLATFORM)/include

max_temperature: max_temperature.cpp
    $(CC) $(CPPFLAGS) $< -Wall -L$(HADOOP_INSTALL)/c++/$(PLATFORM)/lib -lhadooppipes \
    -lhadooputils -lpthread -g -O2 -o $@
```

The makefile expects a couple of environment variables to be set. Apart from HADOOP_INSTALL, you need to define PLATFORM, which specifies the operating system, architecture, and data model (e.g., 32- or 64-bit). It is run on a 32-bit Linux system with the following:

```
% export PLATFORM=Linux-i386-32
% make
```

On successful completion, you'll find the **max_temperature** executable in the current directory.

To run a Pipes job, we need to run Hadoop in *pseudodistributed* mode (where all the daemons run on the local machine). Pipes doesn't run in standalone (local) mode, because it relies on Hadoop's distributed cache mechanism, which works only when HDFS is running.

With the Hadoop daemons now running, the first step is to copy the executable to HDFS so that it can be picked up by tasktrackers when they launch map and reduce tasks:

```
% hadoop fs -put max_temperature bin/max_temperature
```

The sample data also needs to be copied from the local filesystem into HDFS:

```
% hadoop fs -put input/ncdc/sample.txt sample.txt
```

Now we can run the job. For this, we use the Hadoop pipes command, passing the Uniform Resource Identifier (URI) of the executable in HDFS using the -program argument:

```
% hadoop pipes \
  -D hadoop.pipes.java.recordreader=true \
  -D hadoop.pipes.java.recordwriter=true \
  -input sample.txt \
  -output output \
  -program bin/max_temperature
```

We specify two properties using the **-D** option: **hadoop.pipes.java.recordreader** and **hadoop.pipes.java.recordwriter**, setting both to true to say that we have not specified a C++ record reader or writer, but that we want to use the default Java ones (which are for text input and output). Pipes also allows you to set a Java mapper, reducer, combiner, or partitioner. In fact, you can have a mixture of Java or C++ classes within any one job.

The result is the same as the other versions of the same program that we ran previously.

Developing a MapReduceApplication

In this chapter, we look at the practical aspects of developing a MapReduce application in Hadoop.

Writing a program in MapReduce follows a certain pattern. You start by writing your map and reduce functions, ideally with unit tests to make sure they do what you expect. Then you write a driver program to run a job, which can run from your IDE using a small subset of the data to check that it is working. If it fails, you can use your IDE's debugger to find the source of the problem. With this information, you can expand your unit tests to cover this case and improve your mapper or reducer as appropriate to handle such input correctly.

When the program runs as expected against the small dataset, you are ready to unleash it on a cluster. Running against the full dataset is likely to expose some more issues, which you can fix as before, by expanding your tests and mapper or reducer to handle the new cases. Debugging failing programs in the cluster is a challenge, so we look at some common techniques to make it easier.

After the program is working, you may wish to do some tuning, first by running through some standard checks for making MapReduce programs faster and then by doing task profiling. Profiling distributed programs is not easy, but Hadoop has hooks to aid the process.

Before we start writing a MapReduce program, we need to set up and configure the development environment. And to do that, we need to learn a bit about how Hadoop does configuration.

The Configuration API

Components in Hadoop are configured using Hadoop's own configuration API. An instance of the **Configuration** class (found in the **org.apache.hadoop.conf** package) represents a collection of configuration *properties* and their values. Each property is named by a **String**, and the type of a value may be one of several types, including Java primitives such as **boolean**, **int**, **long**, and **float**, other useful types such as **String**, **Class**, and **java.io.File**, and collections of **Strings**.

Configurations read their properties from *resources*—XML files with a simple structure for defining name-value pairs. See Example 5-1.

Example 5-1. A simple configuration file, configuration-1.xml

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>color</name>
    <value>yellow</value>
    <description>Color</description>
  </property>

  <property>
    <name>size</name>
    <value>10</value>
```



```

    <description>Size</description>
  </property>

  <property>
    <name>weight</name>
    <value>heavy</value>
    <final>true</final>
    <description>Weight</description>
  </property>

  <property>
    <name>size-weight</name>
    <value>${size},${weight}</value>
    <description>Size and weight</description>
  </property>
</configuration>

```

Assuming this configuration file is in a file called *configuration-1.xml*, we can access its properties using a piece of code like this:

```

Configuration conf = new Configuration();
conf.addResource("configuration-1.xml");
assertThat(conf.get("color"), is("yellow"));
assertThat(conf.getInt("size", 0), is(10));
assertThat(conf.get("breadth", "wide"), is("wide"));

```

There are a couple of things to note: type information is not stored in the XML file; instead, properties can be interpreted as a given type when they are read. Also, the **get()** methods allow you to specify a default value, which is used if the property is not defined in the XML file, as in the case of **breadth** here.

Combining Resources

Things get interesting when more than one resource is used to define a configuration. This is used in Hadoop to separate out the default properties for the system, defined internally in a file called *core-default.xml*, from the site-specific overrides in *core-site.xml*. The file in Example 5-2 defines the size and weight properties.

Example 5-2. A second configuration file, *configuration-2.xml*

```

<?xml version="1.0"?>
<configuration>
  <property>
    <name>size</name>
    <value>12</value>
  </property>

  <property>
    <name>weight</name>
    <value>light</value>
  </property>
</configuration>

```

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"));
```

Attempting to override final properties usually indicates a configuration error, so this results in a warning message being logged to aid diagnosis. Administrators mark properties as final in the daemon's site files that they don't want users to change in their client-side configuration files or job submission parameters.

Variable Expansion

Configuration properties can be defined in terms of other properties, or system properties. For example, the property **size-weight** in the first configuration file is defined as **`\${size},\${weight}**, and these properties are expanded using the values found in the configuration:

```
assertThat(conf.get("size-weight"), is("12,heavy"));
```

System properties take priority over properties defined in resource files:

```
System.setProperty("size", "14");
assertThat(conf.get("size-weight"), is("14,heavy"));
```

This feature is useful for overriding properties on the command line by using **-Dproperty=value** JVM arguments.

Note that although configuration properties can be defined in terms of system properties, unless system properties are redefined using configuration properties, they are *not* accessible through the configuration API. Hence:

```
System.setProperty("length", "2");
assertThat(conf.get("length"), is((String) null));
```

Setting Up the Development Environment

The first step is to create a project so you can build MapReduce programs and run them in local (standalone) mode from the command line or within your IDE. The Maven POM in Example 5-3 shows the dependencies needed for building and testing Map- Reduce programs.

Example 5-3. A Maven POM for building and testing a MapReduce application

```
<project>
```

```
<modelVersion>4.0.0</modelVersion>
<groupId>com.hadoopbook</groupId>
<artifactId>hadoop-book-mr-dev</artifactId>
<version>3.0</version>
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
</properties>
<dependencies>
  <!-- Hadoop main artifact -->
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-core</artifactId>
    <version>1.0.0</version>
  </dependency>
  <!-- Unit test artifacts -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.10</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-all</artifactId>
    <version>1.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.mrunit</groupId>
    <artifactId>mrunit</artifactId>
    <version>0.8.0-incubating</version>
    <scope>test</scope>
  </dependency>
  <!-- Hadoop test artifacts for running mini clusters -->
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-test</artifactId>
    <version>1.0.0</version>
    <scope>test</scope>
  </dependency>
  <!-- Missing dependency for running mini clusters -->
  <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-core</artifactId>
    <version>1.8</version>
    <scope>test</scope>
  </dependency>
</dependencies>
<build>
  <finalName>hadoop-examples</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
```

```

        <source>1.6</source>
        <target>1.6</target>
    </configuration>
</plugin>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.4</version>
    <configuration>
        <outputDirectory>${basedir}</outputDirectory>
    </configuration>
</plugin>
</plugins>
</build>
</project>

```

The dependencies section is the interesting part of the POM. For building MapReduce jobs you only need to have the **hadoop-core** dependency, which contains all the Hadoop classes. For running unit tests we use **junit**, as well as a couple of helper libraries: **hamcrest-all** provides useful matchers for writing test assertions, and **mrunit** is used for writing MapReduce tests. The **hadoop-test** library contains the “mini-” clusters that are useful for testing with Hadoop clusters running in a single JVM.

Many IDEs can read Maven POMs directly, so you can just point them at the directory containing the *pom.xml* file and start writing code. Alternatively, you can use Maven to generate configuration files for your IDE. For example, the following creates Eclipse configuration files so you can import the project into Eclipse:

```
% mvn eclipse:eclipse -DdownloadSources=true -DdownloadJavadocs=true
```

Managing Configuration

When developing Hadoop applications, it is common to switch between running the application locally and running it on a cluster. In fact, you may have several clusters you work with, or you may have a local “pseudodistributed” cluster that you like to test on (a pseudodistributed cluster is one whose daemons all run on the local machine).

One way to accommodate these variations is to have Hadoop configuration files containing the connection settings for each cluster you run against and specify which one you are using when you run Hadoop applications or tools. As a matter of best practice, it’s recommended to keep these files outside Hadoop’s installation directory tree, as this makes it easy to switch between Hadoop versions without duplicating or losing settings.

Let us assume the existence of a directory called *conf* that contains three configuration files: *hadoop-local.xml*, *hadoop-localhost.xml*, and *hadoop-cluster.xml*. Note that there is nothing special about the names of these files; they are just convenient ways to package up some configuration settings.

The *hadoop-local.xml* file contains the default Hadoop configuration for the default filesystem and the jobtracker:

```

<?xml version="1.0"?>
<configuration>

  <property>
    <name>fs.default.name</name>
    <value>file:///</value>
  </property>

  <property>
    <name>mapred.job.tracker</name>
    <value>local</value>
  </property>

</configuration>

```

The settings in ***hadoop-localhost.xml*** point to a namenode and a jobtracker both running on localhost:

```

<?xml version="1.0"?>
<configuration>

  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost/</value>
  </property>

  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:8021</value>
  </property>

</configuration>

```

Finally, ***hadoop-cluster.xml*** contains details of the cluster's namenode and jobtracker 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.default.name</name>
    <value>hdfs://namenode/</value>
  </property>

  <property>
    <name>mapred.job.tracker</name>
    <value>jobtracker:8021</value>
  </property>

</configuration>

```

You can add other configuration properties to these files as needed. For example, if you wanted to set your Hadoop username for a particular cluster, you could do it in the appropriate file.

GenericOptionsParser, Tool, and ToolRunner

Hadoop comes with a few helper classes for making it easier to run jobs from the command line. **GenericOptionsParser** is a class that interprets common Hadoop command-line options and sets them on a Configuration object for your application to use as desired. You don't usually use **GenericOptionsParser** directly, as it's more convenient to implement the **Tool** interface and run your application with the **ToolRunner**, which uses **GenericOptionsParser** internally:

```
public interface Tool extends Configurable {
    int run(String [] args) throws Exception;
}
```

[Example 5-4](#) shows a very simple implementation of **Tool** that prints the keys and values of all the properties in the **Tool's** Configuration object.

Example 5-4. An example Tool implementation for printing the properties in a Configuration

```
public class ConfigurationPrinter extends Configured implements Tool {

    static {
        Configuration.addDefaultResource("hdfs-default.xml");
        Configuration.addDefaultResource("hdfs-site.xml");
        Configuration.addDefaultResource("mapred-default.xml");
        Configuration.addDefaultResource("mapred-site.xml");
    }

    @Override
    public int run(String[] args) throws Exception {
        Configuration conf = getConf();

        for (Entry<String, String> entry: conf) {
            System.out.printf("%s=%s\n", entry.getKey(), entry.getValue());
        }

        return 0;
    }

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

We make **ConfigurationPrinter** a subclass of **Configured**, which is an implementation of the **Configurable** interface. All implementations of **Tool** need to implement **Configurable** (since **Tool** extends it), and subclassing **Configured** is often the easiest way to achieve this. The **run()** method obtains the **Configuration** using **Configurable's getConf()** method and then iterates over it, printing each property to standard output.

The static block makes sure that the HDFS and MapReduce configurations are picked up in addition to the core ones (which **Configuration** knows about already).

ConfigurationPrinter's main() method does not invoke its own **run()** method directly. Instead, we call **ToolRunner's** static **run()** method, which takes care of creating a **Configuration** object for the **Tool** before calling its **run()** method. **ToolRunner** also uses a **GenericOptionsParser** to pick up any standard options specified on the command line and to set them on the **Configuration** instance. We can see the effect of picking up the properties specified in *conf/hadoop-localhost.xml* by running the following command:

```
% mvn compile
% export HADOOP_CLASSPATH=target/classes/
% hadoop ConfigurationPrinter -conf conf/hadoop-localhost.xml\
|grep mapred.job.tracker=
mapred.job.tracker=localhost:8021
```

GenericOptionsParser also allows you to set individual properties. For example:

```
% hadoop ConfigurationPrinter -D color=yellow | grep color
color=yellow
```

The **-D** option is used to set the configuration property with key **color** to the value **yellow**. Options specified with **-D** take priority over properties from the configuration files. This is very useful because you can put defaults into configuration files and then override them with the **-D** option as needed. A common example of this is setting the number of reducers for a MapReduce job via **-D mapred.reduce.tasks=*n***. This will override the number of reducers set on the cluster or set in any client-side configuration files.

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 (<http://incubator.apache.org/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.

Mapper

The test for the mapper is shown in Example 5-5.

Example 5-5. Unit test for MaxTemperatureMapper

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;
import org.junit.*;

public class MaxTemperatureMapperTest {

    @Test
    public void processesValidRecord() throws IOException, InterruptedException {
        Text value = new Text("00430119909999991950051518004+68750+023550FM-12+0382" +
            // Year ^^^^

```

```

        "99999V0203201N00261220001CN9999999N9-00111+9999999999");
        // Temperature ^^^^^
    new MapDriver<LongWritable, Text, Text, IntWritable>()
        .withMapper(new MaxTemperatureMapper())
        .withInputValue(value)
        .withOutput(new Text("1950"), new IntWritable(-11))
        .runTest();
    }
}

```

The idea of the test is very simple: pass a weather record as input to the mapper, and check that the output is the year and temperature reading.

Since we are testing the mapper, we use MRUnit's **MapDriver**, which we configure with the mapper under test (**MaxTemperatureMapper**), the input value, and the expected output key (a **Text** object representing the year, 1950) and expected output value (an **IntWritable** representing the temperature, -1.1°C), before finally calling the **runTest()** method to execute the test. If the expected output values are not emitted by the mapper, MRUnit will fail the test. Notice that we didn't set the input key because our mapper ignores it.

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())
        .withInputKey(new Text("1950"))
        .withInputValues(Arrays.asList(new IntWritable(10), new IntWritable(5)))
        .withOutput(new Text("1950"), new IntWritable(10))
        .runTest();
}

```

We construct a list of some **IntWritable** values and then verify that **MaxTemperatureReducer** picks the largest. The code in Example 5-7 is for an implementation of **MaxTemperatureReducer** that passes the test.

Example 5-7. Reducer for the maximum temperature example

```

public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

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

        int maxValue = Integer.MIN_VALUE;
        for (IntWritable value : values) {
            maxValue = Math.max(maxValue, value.get());
        }
        context.write(key, new IntWritable(maxValue));
    }
}

```


Running Locally on Test Data

Now that we have the mapper and reducer working on controlled inputs, the next step is to write a job driver and run it on some test data on a development machine.

Running a Job in a Local Job Runner

Using the **Tool** interface introduced earlier in the chapter, it's easy to write a driver to run our **MapReduce** job for finding the maximum temperature by year.

Example 5-8. Application to find the maximum temperature

```
public class MaxTemperatureDriver extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.printf("Usage: %s [generic options] <input> <output>\n",
                getClass().getSimpleName());
            ToolRunner.printGenericCommandUsage(System.err);
            return -1;
        }

        Job job = new Job(getConf(), "Max temperature");
        job.setJarByClass(getClass());

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(MaxTemperatureMapper.class);
        job.setCombinerClass(MaxTemperatureReducer.class);
        job.setReducerClass(MaxTemperatureReducer.class);

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

        return job.waitForCompletion(true) ? 0 : 1;
    }

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

MaxTemperatureDriver implements the **Tool** interface, so we get the benefit of being able to set the options that **GenericOptionsParser** supports. The **run()** method constructs a **Job** object based on the tool's configuration, which it uses to launch a job. Among the possible job configuration parameters, we set the input and output file paths, the mapper, reducer, and combiner classes, and the output types (the input types are determined by the input format, which defaults to **TextInputFormat** and has **LongWritable** keys and **Text** values). It's also a good idea to set a name for the job (Max temperature) so that you can pick it out in the job list during execution and after it has completed. By default, the name is the name of the JAR file, which normally is not particularly descriptive.

Now we can run this application against some local files. Hadoop comes with a local job runner, a cut-down version of the MapReduce execution engine for running Map- Reduce jobs in a single

JVM. It's designed for testing and is very convenient for use in an IDE, since you can run it in a debugger to step through the code in your mapper and reducer.

Testing the Driver

Apart from the flexible configuration options offered by making your application implement **Tool**, you also make it more testable because it allows you to inject an arbitrary **Configuration**. You can take advantage of this to write a test that uses a local job runner to run a **job** against known input data, which checks that the output is as expected.

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 5-11 gives an idea of how to do this.

Example 5-11. A test for `MaxTemperatureDriver` that uses a local, in-process job runner

```
@Test
public void test() throws Exception {
    Configuration conf = new Configuration();
    conf.set("fs.default.name", "file:///");
    conf.set("mapred.job.tracker", "local");

    Path input = new Path("input/ncdc/micro");
    Path output = new Path("output");
    FileSystem fs = FileSystem.getLocal(conf);
    fs.delete(output, true); // delete old output

    MaxTemperatureDriver driver = new MaxTemperatureDriver();
    driver.setConf(conf);

    int exitCode = driver.run(new String[] {
        input.toString(), output.toString() });
    assertThat(exitCode, is(0));

    checkOutput(conf, output);
}
```

The test explicitly sets **fs.default.name** and **mapred.job.tracker** so it uses the local filesystem and the local job runner. It then runs the **MaxTemperatureDriver** via its **Tool** interface against a small amount of known data. At the end of the test, the **checkOutput()** method is called to compare the actual output with the expected output, line by line.

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 and MapReduce machinery.

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 MapReduce 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, run a MapReduce job, and confirm the output is as expected.

Tests like this serve as regression tests, and are a useful repository of input edge cases and their expected results. As you encounter more test cases, you can simply add them to the input file and update the file of expected output accordingly.

Running on a Cluster

Now that we are happy with the program running on a small test dataset, we are ready to try it on the full dataset on a Hadoop cluster.

Packaging a Job

The local job runner uses a single JVM to run a job, so as long as all the classes that your job needs are on its classpath, then things will just work.

In a distributed setting, things are a little more complex. For a start, a job's classes must be packaged into a **job JAR file** to send to the cluster. Hadoop will find the job JAR automatically by searching for the JAR on the driver's classpath that contains the class set in the **setJarByClass()** method (on **JobConf** or **Job**). Alternatively, if you want to set an explicit JAR file by its file path, you can use the **setJar()** method.

Creating a job JAR file is conveniently achieved using a build tool such as Ant or Maven. The following Maven command, for example, will create a JAR file called **hadoop-examples.jar** in the project directory containing all of the compiled classes:

```
% mvn package -DskipTests
```

If you have a single job per JAR, you can specify the main class to run in the JAR file's manifest. If the main class is not in the manifest, it must be specified on the command line.

Any dependent JAR files can be packaged in a **lib** subdirectory in the job JAR file. Similarly, resource files can be packaged in a **classes** subdirectory.

The client classpath

The user's client-side classpath set by **hadoop jar <jar>** is made up of:

- The job JAR file
- Any JAR files in the **lib** directory of the job JAR file, and the **classes** directory (if present)
- The **classpath** defined by **HADOOP_CLASSPATH**, if set

This explains why you have to set **HADOOP_CLASSPATH** to point to dependent classes and libraries if you are running using the local job runner without a job JAR (hadoop **CLASSNAME**).

The task classpath

On a cluster (and this includes pseudodistributed mode), map and reduce tasks run in separate JVMs, and their classpaths are *not* controlled by **HADOOP_CLASSPATH**. **HADOOP_CLASSPATH** is a client-side setting and only sets the classpath for the driver JVM, which submits the job.

Instead, the user's task classpath is comprised of the following:

- The job JAR file
- Any JAR files contained in the **lib** directory of the job JAR file, and the **classes** directory (if present)
- Any files added to the distributed cache, using the **-libjars** option, or the

addFileToClassPath() method on **Job**.

Packaging dependencies

Given these different ways of controlling what is on the client and task classpaths, there are corresponding options for including library dependencies for a job.

- Unpack the libraries and repackage them in the job JAR.
- Package the libraries in the *lib* directory of the job JAR.
- Keep the libraries separate from the job JAR, and add them to the client classpath via **HADOOP_CLASSPATH** and to the task classpath via **-libjars**.

The last option, using the distributed cache, is simplest from a build point of view because dependencies don't need rebundling in the job JAR. Also, the distributed cache can mean fewer transfers of JAR files around the cluster, since files may be cached on a node between tasks.

Task classpath precedence

User JAR files are added to the end of both the client classpath and the task classpath, which in some cases can cause a dependency conflict with Hadoop's built-in libraries if Hadoop uses a different, incompatible version of a library that your code uses. Sometimes you need to be able to control the task classpath order so that your classes are picked up first. On the client side, you can force Hadoop to put the user classpath first in the search order by setting the **HADOOP_USER_CLASSPATH_FIRST** environment variable to true. For the task classpath, you can set **mapreduce.task.classpath.first** to true. Note that by setting these options you change the class loading for Hadoop framework dependencies (but only in your job), which could potentially cause the job submission or task to fail, so use these options with caution.

Launching a Job

To launch the job, we need to run the driver, specifying the cluster that we want to run the job on with the **-conf** option:

```
% unset HADOOP_CLASSPATH
% hadoop jar hadoop-examples.jar v3.MaxTemperatureDriver \
  -conf conf/hadoop-cluster.xml input/ncdc/all max-temp
```

The **waitForCompletion()** method on **Job** launches the job and polls for progress, writing a line summarizing the map and reduce's progress whenever either changes. Here's the output:

```
09/04/11 08:15:52 INFO mapred.FileInputFormat: Total input paths to process : 101
09/04/11 08:15:53 INFO mapred.JobClient: Running job: job_200904110811_0002
09/04/11 08:15:54 INFO mapred.JobClient: map 0% reduce 0%
09/04/11 08:16:06 INFO mapred.JobClient: map 28% reduce 0%
09/04/11 08:16:07 INFO mapred.JobClient: map 30% reduce 0%
...
09/04/11 08:21:36 INFO mapred.JobClient: map 100% reduce 100%
09/04/11 08:21:38 INFO mapred.JobClient: Job complete: job_200904110811_0002
09/04/11 08:21:38 INFO mapred.JobClient: Counters: 19
09/04/11 08:21:38 INFO mapred.JobClient: Job Counters
09/04/11 08:21:38 INFO mapred.JobClient: Launched reduce tasks=32
09/04/11 08:21:38 INFO mapred.JobClient: Rack-local map tasks=82
09/04/11 08:21:38 INFO mapred.JobClient: Launched map tasks=127
```

```

09/04/11 08:21:38 INFO mapred.JobClient: Data-local map tasks=45
09/04/11 08:21:38 INFO mapred.JobClient: FileSystemCounters
09/04/11 08:21:38 INFO mapred.JobClient: FILE_BYTES_READ=12667214
09/04/11 08:21:38 INFO mapred.JobClient: HDFS_BYTES_READ=33485841275
09/04/11 08:21:38 INFO mapred.JobClient: FILE_BYTES_WRITTEN=989397
09/04/11 08:21:38 INFO mapred.JobClient: HDFS_BYTES_WRITTEN=904
09/04/11 08:21:38 INFO mapred.JobClient: Map-Reduce Framework
09/04/11 08:21:38 INFO mapred.JobClient: Reduce input groups=100
09/04/11 08:21:38 INFO mapred.JobClient: Combine output records=4489
09/04/11 08:21:38 INFO mapred.JobClient: Map input records=1209901509
09/04/11 08:21:38 INFO mapred.JobClient: Reduce shuffle bytes=19140
09/04/11 08:21:38 INFO mapred.JobClient: Reduce output records=100
09/04/11 08:21:38 INFO mapred.JobClient: Spilled Records=9481
09/04/11 08:21:38 INFO mapred.JobClient: Map output bytes=10282306995
09/04/11 08:21:38 INFO mapred.JobClient: Map input bytes=274600205558
09/04/11 08:21:38 INFO mapred.JobClient: Combine input records=1142482941
09/04/11 08:21:38 INFO mapred.JobClient: Map output records=1142478555
09/04/11 08:21:38 INFO mapred.JobClient: Reduce input records=103

```

The output includes more useful information. Before the job starts, its ID is printed; this is needed whenever you want to refer to the job—in logfiles, for example—or when interrogating it via the `hadoop job` command. When the job is complete, its statistics (known as counters) are printed out. These are very useful for confirming that the job did what you expected. For example, for this job we can see that around 275 GB of input data was analyzed (Map input bytes), read from around 34 GB of compressed files on HDFS (HDFS_BYTES_READ). The input was broken into 101 gzipped files of reasonable size, so there was no problem with not being able to split them.

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://jobtracker-host:50030/>.

The jobtracker page

A screenshot of the home page is shown in Figure 5-1. The first section of the page gives details of the Hadoop installation, such as the version number and when it was compiled, and the current state of the jobtracker (in this case, running) and when it was started.

Next is a summary of the cluster, which has measures of cluster capacity and utilization. This shows the number of maps and reduces currently running on the cluster, the total number of job submissions, the number of tasktracker nodes currently available, and the cluster's capacity, in terms of the number of map and reduce slots available across the cluster ("Map Task Capacity" and "Reduce Task Capacity") and the number of available slots per node, on average. The number of tasktrackers that have been blacklisted by the jobtracker is listed as well.

Below the summary, there is a section about the job scheduler that is running (here, the default). You can also see job queues.

Further down, we see sections for running, (successfully) completed, and failed jobs. Each of these sections has a table of jobs, with a row per job that shows the job's ID, owner, name (as set in the `Job` constructor or `setJobName()` method, both of which internally set the `mapred.job.name` property), and progress information.

Finally, at the foot of the page, there are links to the jobtracker's logs and the jobtracker's history, which contains information on all the jobs that the jobtracker has run. The main view displays only 100 jobs before consigning them to the history page. Note also that the job history is persistent, so you can find jobs here from previous runs of the jobtracker.

ip-10-250-110-47 Hadoop Map/Reduce Administration [Quick Links](#)

State: RUNNING
Started: Sat Apr 11 08:11:53 EDT 2009
Version: 0.20.0, r763504
Compiled: Thu Apr 9 05:18:40 UTC 2009 by ndaley
Identifier: 200904110811

Cluster Summary (Heap Size is 53.75 MB/888.94 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
53	30	2	11	88	88	16.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0002	NORMAL	root	Max temperature	47.52%	101	48	15.25%	30	0	NA

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0001	NORMAL	gonzo	word count	100.00%	14	14	100.00%	30	30	NA

Failed Jobs

[none](#)

Local Logs

[Log directory](#), [Job Tracker History](#)
 Hadoop, 2009.

Figure 5-1. Screenshot of the jobtracker page

The job page

Clicking on a job ID brings you to a page for the job, illustrated in Figure 5-2. At the top of the page is a summary of the job, with basic information such as job owner and name and how long the job has been running for. The job file is the consolidated configuration file for the job, containing all the properties and their values that were in effect during the job run. If you are unsure of what a particular property was set to, you can click through to inspect the file.

While the job is running, you can monitor its progress on this page, which periodically updates itself. Below the summary is a table that shows the map progress and the reduce progress. "Num Tasks" shows the total number of map and reduce tasks for this job (a row for each). The other columns then show the state of these tasks: "Pending" (waiting to run), "Running," "Complete" (successfully run), or "Killed" (tasks that have failed; this column would be more accurately labeled

“Failed”). The final column shows the total number of failed and killed task attempts for all the map or reduce tasks for the job.

Farther down the page, you can find completion graphs for each task that show their progress graphically. The reduce completion graph is divided into the three phases of the reduce task: copy (when the map outputs are being transferred to the reduce’s tasktracker), sort (when the reduce inputs are being merged), and reduce (when the reduce function is being run to produce the final output).

In the middle of the page is a table of job counters. These are dynamically updated during the job run and provide another useful window into the job’s progress and general health.

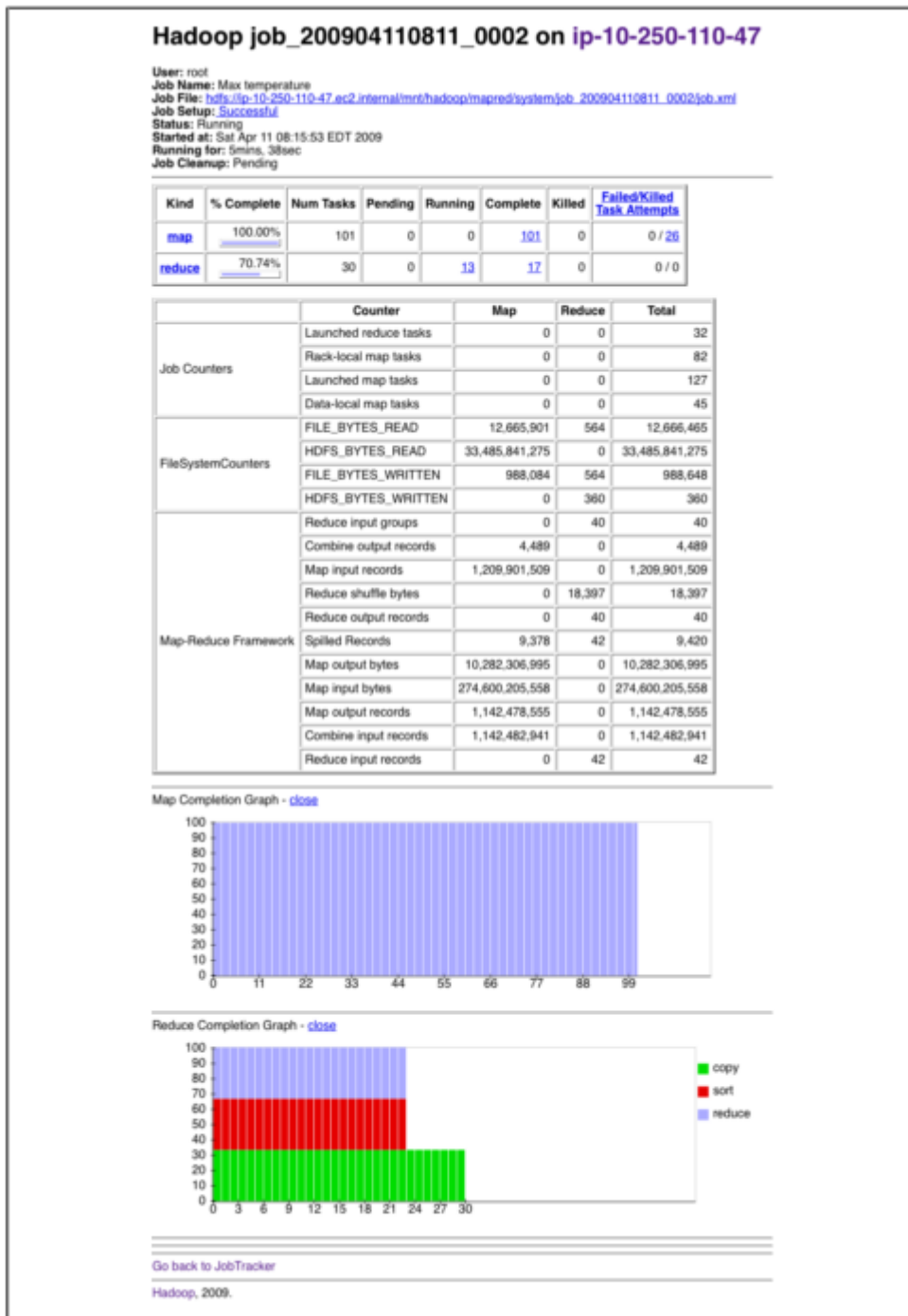


Figure 5-2. Screenshot of the job page

Retrieving the Results

Once the job is finished, there are various ways to retrieve the results. Each reducer produces one output file, so there are 30 part files named *part-r-00000* to *part-r-00029* in the *max-temp* directory.

This job produces a very small amount of output, so it is convenient to copy it from HDFS to our development machine. The **-getmerge** option to the **hadoop fs** command is useful here, as it gets all the files in the directory specified in the source pattern and merges them into a single file on the local filesystem:

```
% hadoop fs -getmerge max-temp max-temp-local
% sort max-temp-local | tail
1991 607
1992 605
1993 567
1994 568
1995 567
1996 561
1997 565
1998 568
1999 568
2000 558
```

We sorted the output, as the reduce output partitions are unordered. Doing a bit of postprocessing of data from MapReduce is very common, as is feeding it into analysis tools such as R, a spreadsheet, or even a relational database.

Another way of retrieving the output if it is small is to use the **-cat** option to print the output files to the console:

```
% hadoop fs -cat max-temp/*
```

On closer inspection, we see that some of the results don't look reasonable. For instance, the maximum temperature for 1951 (not shown here) is 590°C! How do we find out what's causing this? Is it corrupt input data or a bug in the program?

Debugging a Job

The way of debugging programs is via print statements, and this is certainly possible in Hadoop. However, there are complications to consider: with programs running on tens, hundreds, or thousands of nodes, how do we find and examine the output of the debug statements, which may be scattered across these nodes? For this particular case, where we are looking for an unusual case, we can use a debug statement to log to standard error, in conjunction with a message to update the task's status message to prompt us to look in the error log. The web UI makes this easy.

We also create a custom counter to count the total number of records with unreasonable temperatures in the whole dataset. This gives us valuable information about how to deal with the condition.

We add our debugging to the mapper, as opposed to the reducer, as we want to find out what the source data causing the anomalous output looks like:


```

public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

    enum Temperature {
        OVER_100
    }

    private NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        parser.parse(value);
        if (parser.isValidTemperature()) {
            int airTemperature = parser.getAirTemperature();
            if (airTemperature > 1000) {
                System.err.println("Temperature over 100 degrees for input: " + value);
                context.setStatus("Detected possibly corrupt record: see logs.");
                context.getCounter(Temperature.OVER_100).increment(1);
            }
            context.write(new Text(parser.getYear()), new IntWritable(airTemperature));
        }
    }
}

```

If the temperature is over 100°C (represented by 1000, because temperatures are in tenths of a degree), we print a line to standard error with the suspect line, as well as update the map's status message using the **setStatus()** method on **Context** directing us to look in the log. We also increment a counter, which in Java is represented by a field of an **enum** type. In this program, we have defined a single field **OVER_100** as a way to count the number of records with a temperature of over 100°C.

With this modification, we recompile the code, re-create the JAR file, then rerun the job, and while it's running, go to the tasks page.

The tasks page

The job page has a number of links for viewing the tasks in a job in more detail. For example, by clicking on the "map" link, you are brought to a page that lists information for all of the map tasks on one page. You can also see just the completed tasks. The screenshot in Figure 5-3 shows a portion of this page for the job run with our debugging statements. Each row in the table is a task, and it provides such information as the start and end times for each task, any errors reported back from the tasktracker, and a link to view the counters for an individual task.

The "Status" column can be helpful for debugging because it shows a task's latest status message. Before a task starts, it shows its status as "initializing," and then once it starts reading records, it shows the split information for the split it is reading as a filename with a byte offset and length. You can see the status we set for debugging for task **task_200904110811_0003_m_000044**, so let's click through to the logs page to find the associated debug message. (Notice that there is an extra counter for this task because our user counter has a nonzero count for this task.)

The task details page

From the tasks page, you can click on any task to get more information about it. The task details page, shown in Figure 5-4, shows each task attempt. In this case, there was one task attempt, which completed successfully. The table provides further useful data, such as the node the task attempt ran on and links to task logfiles and counters.

The “Actions” column contains links for killing a task attempt. By default, this is disabled, making the web UI a read-only interface. Set **webinterface.private.actions** to true to enable the actions links.

Hadoop map task list for [job 200904110811_0003](#) on [ip-10-250-110-47](#)

Completed Tasks



Task	Complete	Status	Start Time	Finish Time	Errors	Counters
task_200904110811_0003_m_000043	100.00% 	hdfs://ip-10-250-110-47.ec2.internal/user/root/input/ncdc/all/1949.gz:0+220338475	11-Apr-2009 09:00:06	11-Apr-2009 09:01:25 (1mins, 18sec)		10
task_200904110811_0003_m_000044	100.00% 	Detected possibly corrupt record: see logs.	11-Apr-2009 09:00:06	11-Apr-2009 09:01:28 (1mins, 21sec)		11
task_200904110811_0003_m_000045	100.00% 	hdfs://ip-10-250-110-47.ec2.internal/user/root/input/ncdc/all/1970.gz:0+208374610	11-Apr-2009 09:00:06	11-Apr-2009 09:01:28 (1mins, 21sec)		10

Figure 5-3. Screenshot of the tasks page

Job [job 200904110811_0003](#)

All Task Attempts

Task Attempts	Machine	Status	Progress	Start Time	Finish Time	Errors	Task Logs	Counters	Actions
attempt_200904110811_0003_m_000044_0	/default-rack/ip-10-250-163-143.ec2.internal	SUCCEEDED	100.00% 	11-Apr-2009 09:00:06	11-Apr-2009 09:01:25 (1mins, 19sec)		Last 4KB Last 8KB All	11	

Input Split Locations

- [/default-rack/10.250.202.127](#)
- [/default-rack/10.250.123.223](#)
- [/default-rack/10.250.115.79](#)

[Go back to the job](#)
[Go back to JobTracker](#)

Hadoop, 2009.

Figure 5-4. Screenshot of the task details page

Hadoop Logs

Hadoop produces logs in various places, and for various audiences. These are summarized in Table 5-2.

Table 5-2. Types of Hadoop logs

Logs	Primary audience	Description
System daemon logs	Administrators	Each Hadoop daemon produces a logfile (using log4j) and another file that combines standard out and error. Written in the directory defined by the HADOOP_LOG_DIR environment variable.
HDFS audit logs	Administrators	A log of all HDFS requests, turned off by default. Written to the namenode's log, although this is configurable.
MapReduce job history logs	Users	A log of the events (such as task completion) that occur in the course of running a job. Saved centrally on the jobtracker and in the job's output directory in a <i>_logs/history</i> subdirectory.
MapReduce task logs	Users	Each tasktracker child process produces a logfile using log4j (called <i>syslog</i>), a file for data sent to standard out (<i>stdout</i>), and a file for standard error (<i>stderr</i>). Written in the <i>userlogs</i> subdirectory of the directory defined by the HADOOP_LOG_DIR environment variable.

As we have seen in the previous section, MapReduce task logs are accessible through the web UI, which is the most convenient way to view them. You can also find the logfiles on the local filesystem of the tasktracker that ran the task attempt, located in a directory named by the task attempt. If task JVM reuse is enabled, each logfile accumulates the logs for the entire JVM run, so multiple task attempts will be found in each logfile. The web UI hides this by showing only the portion that is relevant for the task attempt being viewed.

It is straightforward to write to these logfiles. Anything written to standard output or standard error is directed to the relevant logfile.

Remote Debugging

When a task fails and there is not enough information logged to diagnose the error, you may want to resort to running a debugger for that task. This is hard to arrange when running the job on a cluster, as you don't know which node is going to process which part of the input, so you can't set up your debugger ahead of the failure. However, there are a few other options available:

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.

Use IsolationRunner

Older versions of Hadoop provided a special task runner called **IsolationRunner** that could rerun failed tasks in situ on the cluster.

In some cases it's useful to keep the intermediate files for a failed task attempt for later inspection, particularly if supplementary dump or profile files are created in the task's working directory. You can set **keep.failed.task.files** to true to keep a failed task's files.

You can keep the intermediate files for successful tasks, too, which may be handy if you want to examine a task that isn't failing. In this case, set the property **keep.task.files.pattern** to a regular expression that matches the IDs of the tasks you want to keep.

To examine the intermediate files, log into the node that the task failed on and look for the directory for that task attempt. It will be under one of the local MapReduce directories, as set by the **mapred.local.dir** property. If this property is a comma-separated list of directories (to spread load across the physical disks on a machine), you may need to look in all of the directories before you find the directory for that particular task attempt. The task attempt directory is in the following location:

mapred.local.dir/taskTracker/jobcache/job-ID/task-attempt-ID
