Learning Cascading

Cascading is open source software that is used to create and execute complex data processing workflows on big data clusters. The book starts by explaining how Cascading relates to core big data technologies such as Hadoop MapReduce. Having instilled an understanding of the technology, the book provides a comprehensive introduction to the Cascading paradigm and its components using code examples. You will not only learn more advanced Cascading features, you will also write code to utilize them. Furthermore, you will gain in-depth knowledge of how to efficiently optimize a Cascading application. To deepen your knowledge and experience with Cascading, you will work through a real-life case study using Natural Language Processing to perform text analysis and search on large volumes of unstructured text. Throughout the book, you will receive expert advice on how to use the portions of the product that are undocumented or have limited documentation. By the end of the book, you will be able to build practical Cascading applications.

Who this book is written for

This book is intended for software developers, system architects and analysts, big data project managers, and data scientists who wish to deploy big data solutions using the Cascading framework. You must have a basic understanding of the big data paradigm and should be familiar with Java development techniques.

What you will learn from this book

- Familiarize yourself with tuples, pipes, taps, and flows and build your first Cascading application
- Discover how to design, develop, and use custom operations
- Design, develop, use, and reuse code with subassemblies and Cascades
- Acquire the skills you need to integrate Cascading with external systems
- Gain expertise in testing, QA, and performance tuning to run an efficient and successful Cascading project
- Explore project management methodologies and steps to develop workable solutions
- Discover the future of big data frameworks and understand how Cascading can help your software to evolve with it
- Uncover sources of additional information and other tools that can make development tasks a lot easier

Build reliable, robust, and high-performance big data applications using the Cascading application development efficiently

Foreword by Chris K. Wensel, CTO, Concurrent, Inc.

In this package, you will find:

- The author’s biography
- A preview chapter from the book, Chapter 1 'The Big Data Core Technology Stack'
- A synopsis of the book’s content
- More information on Learning Cascading

About the Authors

**Michael Covert**, CEO, Analytics Inside LLC, has significant experience in a variety of business and technical roles. Michael is a mathematician and computer scientist and is involved in machine learning, deep learning, predictive analytics, graph theory, and big data. He earned a bachelor's of science degree in mathematics with honors and distinction from The Ohio State University. He also attended it as a PhD student, specializing in machine learning and high-performance computing. Michael is a Cloudera Hadoop Certified Developer.

Michael served as the vice president of performance management in Whittman-Hart, Inc., based in Chicago, and as the chief operating officer of Infinis, Inc., a business intelligence consulting company based in Columbus, Ohio. Infinis merged with Whittman-Hart in 2005. Prior to working at Infinis, Michael was the vice president of product development and chief technology officer at Alta Analytics, and the producer of data mining and visualization software. In addition to this, he has served in technology management roles for Claremont Technology Group, Inc., where he was the director of advanced technology.
Victoria Loewengart, COO, Analytics Inside LLC, is an innovative software systems architect with a proven record of bringing emerging technologies to clients through discovery, design, and integration. Additionally, Victoria spent a large part of her career developing software technologies that extract information from unstructured text. Victoria has published numerous articles on topics ranging from text analytics to intelligence analysis and cyber security. Her book *An Introduction to Hacking & Crimeware: A Pocket Guide* was published by IT Governance, UK, in January 2012. Victoria earned a bachelor's degree in computer science from Purdue University and a master's degree in intelligence studies from the American Military University.
Learning Cascading

Big data is the new "must have" of this century. Suddenly, everyone wants to manage huge amounts of data and find patterns in it, which they did not see before. The problem, however, is that big data is not one, but a whole slew of technologies, which work together to produce the desired outcome. New technologies are emerging, and existing ones are evolving very quickly. The design and development of big data systems is very complex, with an incredibly steep learning curve and not a whole lot of prior experience and best practices to rely on. This is where Cascading comes in. Cascading sits on top of the core big data frameworks and makes design and development of big data applications intuitive and fun! Cascading significantly simplifies and streamlines application development, job creation, and job scheduling. Cascading is an open source software, and many large organizations prefer to use it to manage their big data systems over more complicated solutions, such as MapReduce.

We discovered Cascading after our applications started to get to the level of complexity when pure MapReduce turned into a blood-pressure-raising nightmare. Needless to say, we fell in love with Cascading. Now, we train other developers on it, and evangelize it every chance we get. So, this is how our book came about. Our vision was to provide a book to the Cascading user community that will help them accelerate the development of complex, workflow-based Cascading applications, while still keeping their sanity intact so that they can enjoy life.

This book will teach you how to quickly develop practical Cascading applications, starting with the basics and gradually progressing into more complex topics. We start with a look "under the hood", how Cascading relates to core big data technologies, such as Hadoop MapReduce, and future emerging technologies, such as Tez, Spark, Storm, and others. Having gained an understanding of underlying technologies, we follow with a comprehensive introduction to the Cascading paradigm and components using well-tested code examples that go beyond the ones in the open domain that exist today. Throughout this book, you will receive expert advice on how to use the portions of a product that are undocumented or have limited documentation. To deepen your knowledge and experience with Cascading, you will work with a real-life case study using Natural Language Processing to perform text analysis and search large volumes of unstructured text. We conclude with a look to the future, and how Cascading will soon run on additional big data fabrics, such as Spark and Tez.

Cascading has rapidly gained popularity, and obtaining development skills in this product is a very marketable feature for a big data professional. With in-depth instructions and hands-on practical approaches, *Learning Cascading* will ensure your mastery of Cascading.
What This Book Covers

Chapter 1, The Big Data Core Technology Stack, introduces you to the concepts of big data and the core technologies that comprise it. This knowledge is essential as the foundation of learning Cascading.

Chapter 2, Cascading Basics in Detail, we will begin our exploration of Cascading. We will look at its intended purpose, how it is structured, how to develop programs using it base classes, and how it then manages the execution of these programs.

Chapter 3, Understanding Custom Operations, is based on the newly acquired Cascading knowledge. We will learn more advanced Cascading features that will truly unleash the power of the Cascading API. We will concentrate on an in-depth discussion about operations and their five primary types.

Chapter 4, Creating Custom Operations, will show you how to actually write a custom operation for each type of operation. This is done after we've completely explored and understood the anatomy of operations in the previous chapter.

Chapter 5, Code Reuse and Integration, introduces you to learning how to assemble reusable code, integrating Cascading with external systems, and using the Cascading instrumentation effectively to track and control execution. Specifically, we will discuss how Cascading fits into the overall processing infrastructure.

Chapter 6, Testing a Cascading Application, provides in-depth information on how to efficiently test a Cascading application using many techniques.

Chapter 7, Optimizing the Performance of a Cascading Application, provides in-depth information on how to efficiently optimize a Cascading application and configure the underlying framework (Hadoop) for maximum performance.

Chapter 8, Creating a Real-world Application in Cascading, takes everything we learned and creates a real-world application in Cascading. We will use a practical case study, which could be easily adapted to a functional component of a larger system within your organization.

Chapter 9, Planning for Future Growth, provides information that will allow you to continue to advance your skills.

Appendix, Downloadable Software, provides information that will allow you to download all the code from the book.
This chapter will introduce the reader to the concepts of big data and the core technologies that comprise it. This knowledge is essential as the foundation to learn Cascading. Cascading provides a comprehensive framework that definitely eases many of the verbose and mundane tasks associated with writing Hadoop, and in the future, many other types of big data jobs. However, as with all complex subjects, an understanding of how Hadoop works is required to gain a full understanding of how to best use Cascading.

Reviewing Hadoop

Hadoop is very complex and has many low-level details and nuances that require a significant amount of documentation and explanation to cover. This chapter is a high-level overview of the Hadoop system and is not intended to be sufficient for anything other than laying the groundwork that you will need to begin Cascading programming. For a deeper dive into Hadoop, we recommend *Hadoop Explained* by Aravind Shenoy.

Hadoop is designed to provide massively parallel (often referred to as "embarrassingly parallel"), fault-tolerant, and high-performance computing. Hadoop is intended to solve one specific problem — how to reduce the duration of time that is required to read a very large amount of data from disk sequentially. We see that with most physical (mechanical spinning) disks, the amount of time to read a block of data is in the order of 2 ms-10 ms, depending on the characteristics of the disk. If we do a simple calculation using a disk that can read 25 MBps, we see that reading 1 TB of data sequentially will require 40 seconds. However, if this data were spread across multiple disks, say 20 of them, we could read this data parallelly in 2 seconds! This is the key element of Hadoop — using parallel I/O to improve performance.
We must also note that disk performance improvements have lagged far behind compute improvements. While disk performance (measured in megabytes per second) has only moderately increased, compute performance (such as instructions per second or other benchmark metrics) has improved by orders of magnitude, growing at an average rate of 50 percent annually since 1986, and a modern day processor now runs 1000 times faster than one of a 1986 vintage. This disparity between I/O and compute provides the basis for the invention of Hadoop. Hadoop provides compute parallelism through its MapReduce (MR) subsystem. It provides I/O parallelism through its Hadoop Distributed File System (HDFS) subsystem.

Hadoop itself is a very large system comprised of over approximately 2.4 million lines of code as of the current version. There is much to learn, and much of this knowledge is used daily. As a result, the learning curve is a steep one. One must become quite proficient at library structures, performing I/O in a cluster, accessing shared objects, Hadoop serialization, and a large number of specific classes that one must write to interact with execution and I/O subsystems. For instance, as we shall see in detail later in this chapter, just to do a multfield sort in MapReduce requires that five separate Java classes be written.

Some required knowledge is arcane, such as how to effectively process time ranges where files have been partitioned by date range, or how to only read portions of a file. After initially working with MapReduce, one can find that an enormous amount of code has been written, and that without careful planning, reusing the code is difficult to obtain. One typically ends up with many, many packages, hundreds of classes, and a significant amount of code overlap (this includes classes with minor variations that are rewritten using copied code).
Hadoop is almost universally deployed on Linux operating systems. However, it can be made to run on OS/X, many flavors of Unix, BSD, and now Microsoft Windows as well. It is also very amenable to Cloud deployment and can support a variety of Cloud-based file systems. Hadoop requires Java 1.6 or above. Versions that exist now support Java 1.7 and 1.8 as well.

Hadoop is inherently a batch processing system. Work is packaged into a job. The job is submitted and is run at the discretion of the Hadoop scheduler. The amount of time between starting the job and having results returned is not under the control of the issuer, and, in general, is difficult to use for any sort of processing where there is a desire to bound the time of execution. For instance, it is very difficult to connect a web server user request to a Hadoop job and then to wait for it to return results to achieve some service-level objective.

Hadoop itself is made more difficult to understand largely because of its history of growth and change. It now has three full versions, each with significantly different semantics. Initially, the first release of Hadoop used the class package prefix of mapred. It was superseded by a newer release that used the class package prefix of mapreduce. Later, we will see that the newest version, YARN, is now significantly different.

Hadoop is controlled by several XML configuration files. On a Linux system, these files generally reside within /etc/hadoop/conf, but the location of these files is actually dependent on the installation (and vendor, if you choose this route). Three primary files are used to control Hadoop:

- **core-site.xml**: It contains basic parameters, such as server port numbers, location of servers, and so on
- **hdfs-site.xml**: It contains information that controls the Hadoop Distributed File System, such as locations on local disk where data is to be stored
- **mapred-site.xml**: It contains information about the MapReduce execution framework, such as the number of threads to use

A typical configuration file looks similar to this:

```xml
<?xml version="1.0"?>
<configuration>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/tmp/hadoop-${user.name}</value>
  </property>
  <property>
    <name>fs.default.name</name>
  </property>
</configuration>
```
Additionally, several shell scripts are needed. A key script, such as hadoop-env.sh, is needed in instances where environment variables are set. Some important environment variables are:

- **JAVA_HOME**: It sets the location of the default Java implementation
- **HADOOP_HOME**: It sets the location of the Hadoop implementation
- **HADOOP_CONF_DIR**: It specifies where the Hadoop configuration files are located

Often, the hadoop-env.sh script is placed into /etc/profile.d, so that it will be executed automatically at login. However, sometimes, they are placed in the user's local shell startup script, so this may be different for you.

### Hadoop architecture

At its most basic level, Hadoop is a system that processes records represented as key-value pairs. Every record has an associated **key** field. This key need not be unique and is accompanied by a **value** field. This value field may be composite, so that it can be decomposed into multiple subfields. While this may sound limiting at first, especially when one is used to using a relational database where multiple indexes are available, we will see that this mechanism is sufficient to do most of the processing that one may require. We will also see that this simplified approach is fundamental to the performance gains that are required to process big data.
In a nutshell, Hadoop MapReduce processing can be visualized in the following diagram:

![Diagram showing Hadoop processing flow](image)

Figure 1.1 – Hadoop processing flow

While the flow does look simplistic, in this diagram, we can see some underlying similarities to how a relational database processes standard SQL queries. The **Read** phase looks a lot like `SELECT *`, retrieving all the records. The **Map** process is then capable of applying the WHERE criteria. The **Sort and Group** step applies both to `GROUP BY` and an `ORDER BY`. The **Reduce** process can then apply processing logic, such as `COUNT`, `MIN`, `MAX`, and so on. Lastly, the **Write** phase persists this data back to the disk.

One of the most interesting concepts behind Hadoop is that its data is **schema-on-read**. This means that while the data does have some sort of format, this format is not imposed until the data is read. This is in sharp contrast to the philosophy of a **relational database management system (RDBMS)**, where the schema is defined long before any data can be placed in it. Schema-on-read facilitates faster development times, since no lead times are required to define the schema, prepare the **data definition language (DDL)** — the `CREATE TABLE` statement), execute it, load the data, and correct records with errors. In Hadoop, all data, regardless of type or format, can be copied into its storage and processed.

---

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
What this means is that the interpretation of the input record itself, its key and value, is entirely dependent on the programmer. In one case, the key may be something that we would think of as an actual key, such as a name, SSN, order number, and so on. In other cases, the key may be nothing more than the record number, or maybe even the offset within the input file. This is also true of the value field. It could be a single value, such as a line of text, or it could be somehow parsable into smaller parts through some sort of record offset, or maybe just by performing a `split()` against some sort of delimiter character. Additionally, Hadoop provides many file formats that make handling data easier by supplying metadata to parse a record. Some examples of these formats are SequenceFile, Avro, and others. With these types of file structures, the actual format of the record is encoded into the dataset itself, and when the record is read, it is returned in a decomposed, record-like format, where fields can be retrieved by offsets or even by name.

The Hadoop architecture is relatively complex. It consists of two major subsystems—one that manages data and files, and the other that manages execution. The Hadoop system itself does much of the work for you, but the developer is still required to write a lot of code to perform the work.

![Figure 1.2 – Hadoop logical architecture](image-url)
The basic tasks that Hadoop performs are as follows:

- Hadoop manages files in a familiar directory structure. However, these files are replicated in a cluster for fault tolerance and performance. By default, each file of data is replicated three times. This is a tunable parameter that can be set at the file, directory, or global level.

- Hadoop treats a directory of files as a single file. The files are concatenated together and then split into large blocks for consumption by executing tasks.
  - The block size is a file-level parameter. The default size is 64 MB, but sometimes this size is even further increased. The rationale here is to provide each task that runs in the cluster a sufficiently large amount of data so that the ratio of task startup time (consisting mainly of transferring executable code to the target system and then starting the Java Virtual Machine) to program execution time is small.
  - **Parallel** tasks are created in the cluster, and each is given data to process. These processes run simultaneously, and the purpose here is to improve execution speed by processing these blocks in parallel. Hadoop tasks are divided into mapper tasks and reducer tasks.
    - **Mapper** tasks process records from the input that is supplied to it. Each mapper task is assigned one block of data to process, and it is handed one record of data from its assigned block. The mapper implements three basic method calls. Only map() actually needs to be defined.

```java
void setup(Context c)
throws IOException, InterruptedException
void map(<KEYIN> key, <VALUEIN> value,
        Context context)
throws IOException, InterruptedException
void cleanup(Context c)
throws IOException, InterruptedException
```

Mappers process one record at a time, so setup() is performed before any data is sent to the mapper. Then, map() is called repeatedly for each record. The context object passed to the map() call, and contains methods to output data that is the result of its processing. Finally, after all records are processed, cleanup() is called once to allow the user any final processing that may be required.
Note that in Hadoop, only mappers are required. The number of mappers used will be
\((\text{total-bytes-processed}/\text{block-size})\).

Additionally, note that Hadoop methods should be able to throw IllegalAccessException if an I/O error occurs, and also an InterruptedException if they are somehow stopped by any one of many types of cluster-wide interruptions that could occur.

Reducer tasks receive data that has been outputted by the mappers. Each mapper task processes records from its assigned block and then outputs a result. The result is sent to one reduce task for final processing. Note that, as shown in Figure 1.1, each record may be sent to a different reducer. We will discuss this in a minute.

Unlike what we saw with mappers, the number of reducers to use is specified by the programmer. This allows total control of parallelism during the final stage of aggregation. The assumption here is that the developer has some idea of the data itself and is, therefore, able to determine the degree of parallelism that is needed. It is important to note that the reducer can become a major bottleneck. For instance, if only one reducer is used, all the data generated by all the mappers will flow through it sequentially. Additionally, if one reducer receives significantly more data than others, a bottleneck will also occur.

Similar to a mapper, a reducer has the following method calls:

\begin{verbatim}
void setup(Context c)
throws IOException, InterruptedException

void reduce(<INKEY> key, Iterable<INVALUE> values, Context context)
throws IOException, InterruptedException

void cleanup(Context c)
throws IOException, InterruptedException
\end{verbatim}

Reducer tasks receive data sent by the mappers that have been grouped by the Hadoop framework. This a record that consists of the grouping key followed by an Iterable object, containing every processed record that pertains to this key. Hadoop does this grouping for you through a Shuffle and Sort process that occurs after the mapper has sent records to the reducer(s).
Now, how does a mapper know which reducer should receive the record? This decision is made by a **partitioner**. A partitioner analyzes each record and determines the reducer that should process it. A partitioner has one method call that must be implemented:

```java
void configure(Context c)
int getPartition(<KEY> key, <VALUE> value,
    int numReduceTasks)
```

Note that the partitioner passed a parameter telling it how many reducers it has to use. It must then return a value between 0 and `numReduceTasks`, and 1 to its caller (that is, the Hadoop framework). Hadoop will then handle the routing of this record to this reducer. The default partitioner simply looks at the key of the record and uses a modulo function, based on the number of reducers that have been defined to route it.

```java
return key.hashCode() % numReduceTasks;
```

Hadoop allows for another component to be defined as well, and this is the **combiner**. When data is sent from the mappers to the reducers, a tremendous amount of data can be generated and sent across the wire. In some cases, bandwidth could be conserved if some preprocessing were to occur on the mapper side. This is precisely what a combiner does. It is a sort of "mini-reducer" that runs on the mapper side. It allows records to be combined (hence its name), and a consolidated record to be sent to the reducer. Care must be taken when using it, since mathematical operations that it performs should be both commutative and associative.

- Hadoop manages all these tasks, allowing them to run, pass data to each other. Data flows forward from the mappers to the reducers. In order for the reducers to receive the input that they expect, Hadoop performs sorting and aggregation/consolidation to create the composite record.
- Job execution is monitored and managed. Progress is reported back from the job so that it can be tracked for completion.
During job execution, various errors or transient conditions may occur that hinder the job's progress. In these cases, Hadoop will attempt to keep the job running and may take corrective actions:

- Failing jobs may be restarted.
- Slow running jobs may be restarted as well.
- The same tasks may be run multiple times using what is called **speculative execution**. When this occurs, Hadoop starts multiple copies to see which one will finish first. This typically occurs when Hadoop determines that some sort of performance delay is occurring on one of the tasks. In this case, the task finishing first is used and the loser is unceremoniously terminated.

Most typically today, Hadoop configurations consist of a set of physical or virtual nodes, which are complete standalone systems running some version of the Linux operating system, an installed version of the Java JDK, and are all networked together with high speed Ethernet (such as InfiniBand) connectivity. Nodes are then divided into the following types:

- **Head nodes**: These are the controlling systems that contain the servers required to submit jobs, manage data, monitor a system, provide error recovery, failovers, and software distribution.
- **Slave nodes**: These are the nodes that do the actual work. They contain local disk storage (usually a lot of it), and run programs that perform the work required. They report the status of the work performed back to several of the head nodes.
- **Boundary nodes**: These are the nodes where users submit units of work to a cluster. Typically, these nodes are not part of an actual cluster, but have networked access to it. These nodes are also sometimes referred to as gateway nodes.
Figure 1.3 shows a typical Hadoop cluster. The user sits at a boundary node system and creates/submits jobs to the cluster. The JobTracker server receives a request and places it into its Job Queue. Later, the JobTracker will schedule the job for execution and distribute it to one or more Task Tracker servers to start the Java Virtual Machine (JVM) and execute the program. Task Trackers are configured with slots, which represent the number of job components that they are allowed to start.

In the above diagram, the following points about the cluster should be noted.

- The JobTracker and NameNode reside on a head node that manages the execution of the jobs.
- Every other node shown is a slave node.
- Typically, many more head nodes are used than the single one shown in the preceding figure.
- Note that there are asymmetric nodes that have differing numbers of slots (and maybe even different hardware configurations, such as number of processor cores, and so on). While this is possible, it is discouraged, since it will lead to another level of diagnostics when analyzing performance delays.
There are some key points to be made from this diagram:

- Hadoop replicates its data in the cluster. The default is that every block of data is written three times (the replicated block is shown in red). This aids in redundancy, performance, and scheduling. This is a definable parameter and is controlled globally by specifying `dfs.replication` in the Hadoop configuration files.

- Blocks of data are very large with the default size being 64 MB. This size is controllable and is typically much larger, usually 128 MB and beyond. Often, this data is compressed, and the method of compression is also controllable. Remember that we have a massive amount of compute at our disposal, and we are trying to address the I/O latency to improve performance.

- Hadoop is "rack aware." A **rack** is a hardware unit that contains multiple servers, typically, in a single container (a rack). HDFS, when replicating data will attempt to place blocks on different racks, if possible. The rationale is that should an entire rack fail, no data will be lost, and I/O across servers that reside in the same rack is generally faster than I/O across different racks, because of rack-level network optimization (that is, high speed backplanes).

  Note that the definition of a rack is a manual task. A rack number must be defined for a node. It is not automatically detected.

- Hadoop jobs are Java or Java-based programs. They are packaged in Java JAR files (Java Archives). The JAR files are delivered to the data. The rationale here is simple: it is less costly to move the relatively smaller JAR file to where the data lives, rather than transfer it through the network to an executing program. It is often said that "moving computation is cheaper than moving data."

- Being rack aware, the **JobTracker** can be smarter about the node it uses to dispatch a task. Its first choice is to pick a node with a free slot where the data resides locally. Its second choice is to use a node with a free slot where data exists on the same rack. Its third and final choice is to use any node that has a free slot.
HDFS – the Hadoop Distributed File System

Hadoop comes with its own form of file storage called the Hadoop distributed file system (HDFS). HDFS is designed to do several things:

1. Provide a namespace that can control, read, write, update, and delete actions performed on files using a POSIX style of file system. A typical HDFS file locator (a URI) for a file named file.tsv, owned by a user named mcovert, is of the hdfs://users/mcovert/data/file.tsv form.

2. Provide redundancy so that losing a small section of data will not break the cluster.

3. Provide high speed and parallel access to data, thereby feeding the execution framework with data as quickly as possible.

4. Provide utilities (the balancer) that can rectify any imbalances that may exist in the cluster. For instance, if a node fails, typically all of its blocks will be lost, but the balancer will assign new nodes where these missing blocks can be copied from the surviving two nodes.

HDFS is implemented in the form of several server processes that handle I/O requests for data stored in the cluster. These server processes are explained in the next section.

The NameNode

The NameNode is where information about the data in the cluster is stored. It represents a catalog of all the files and directories that are managed by the cluster. The NameNode is a very complex server process. It is memory-intensive, since it caches most file metadata. Also, note that a file that resides in HDFS is spread across data nodes in the cluster, and also that each block of the file is replicated, so that the data catalog tends to be very large for even a moderate sized system.

The NameNode itself is a Java server. Hadoop provides an application programming interface (API) to access data and to perform various file management tasks. When a program runs, it is assigned a block of data to process, and the NameNode is queried to find the location of the server (see DataNode in the DataNodes section that follows) where the data resides, and also to obtain various metadata about the file and the data block. When data is written back into HDFS, the NameNode is notified, so that it can record the block metadata into its catalog, and to subsequently handle replication as well. Later, when we discuss the DataNode, we will complete this understanding.
The secondary NameNode

The secondary NameNode is where a backup copy of NameNode data is stored. This server provides recoverability in case of catastrophic failures. While the mechanics of catastrophic recovery are beyond the intentions of this discussion, there are two modes of recovery that can be configured. A basic mode of recovery can occur when the NameNode fails (for instance, due to a hard drive error), and then the redundant copy of the NameNode metadata is used to restart the failed NameNode. In this case, the cluster itself fails, and all running jobs are lost and must be rerun. A more sophisticated capability exists, called High Availability (HA), where the secondary NameNode assumes control of HDFS storage. In this case, the cluster will continue to run, and active jobs generally can be completed without requiring a restart.

DataNodes

Every node in the cluster that is assigned the task of handling data runs a DataNode. The DataNode performs local I/O, which means that when a block of data belonging to a file in the cluster has been requested, the NameNode finds it, and then assigns the DataNode that owns the data block to deliver it to the requesting program.

The DataNode is largely unaware of the activities of the NameNode and is only responsible for storing data locally. It is not stored by the actual file name though. The DataNode stores blocks as files that are spread across directories. This prevents a directory from becoming overloaded and makes the time required to open files occur faster. The DataNode is only responsible for passing this information back to the NameNode so that it can be stored in the catalog.

MapReduce execution framework

MapReduce jobs are orchestrated by two primary server types—the JobTracker and the TaskTracker. There is one JobTracker, and it uses one or more TaskTrackers running on slave nodes where it distributes the work (the mappers and reducers).

The JobTracker

The JobTracker is the manager of all the jobs that are submitted to Hadoop and it performs many tasks. It queues jobs into an input queue, and then determines the order they are allowed to run in. Hadoop starts with a very basic approach here, which is to run jobs in the order in which they are received, or first come first served (FCFS). Clearly, this can be very inefficient, so Hadoop allows this to be customized by providing several other types of schedulers that take into account system capacity, assigned job priority, assigned job class, and so on.
The TaskTracker
The TaskTracker is the manager of all the tasks that are started by the JobTracker. A TaskTracker is responsible for a lot of things:

- It physically manages the JVM container that will run the task.
  - Note that typically, a JVM requires a few seconds to initialize. Therefore, Hadoop provides a, `mapred.job.reuse.jvm.num.tasks` parameter, that can be used to reuse an existing JVM. Its default value is 1.

  [YARN does not support the reuse of JVM.]

- It starts the mapper or reducer that it has been assigned.
- It handles I/O mapping for its assigned task.
- It reports progress back to the JobTracker.

The TaskTracker also provides a "heartbeat" back to the JobTracker. This is a message sent by default every 3 seconds. It is configurable through the `dfs.heartbeat.interval` parameter.

Hadoop jobs
Every job that executes on the Hadoop cluster is represented by a Job object. Through this object, the job also has access to a Configuration object that can be used to pass information to its tasks as they execute. This object is essentially a HashMap of keys and values, where the key represents the parameter name, and the value represents the value of the parameter. In a job, you create this object, set values on it, and then use it to first create the Job object, which is then submitted to the cluster:

```java
import org.apache.hadoop.conf.Configuration;

Configuration conf = new Configuration();
conf.set("parm", "value");
// Set other configuration items
Job job = Job.getInstance(conf);
// Set up your job
job.setJarByClass(MRJob.class);
job.setJobName("MRJob");
```
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.setMapperClass(MRMapper.class);
job.setReducerClass(MRReducer.class);
job.setPartitionerClass(MRPartitioner.class);
job.setSortComparatorClass(MRComparator.class);
job.setMapOutputKeyClass(org.apache.hadoop.io.Text.class);
job.setMapOutputValueClass(org.apache.hadoop.io.Text.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
job.setNumReduceTasks(1);
int return_code = (job.waitForCompletion(true)) ? 1 : 0;
return return_code;

**Distributed cache**

One important thing to remember is that when your job is run and it submits your work, it will run on a boundary machine, but the components of your job (its mappers and reducers) will run on nodes within the cluster. Therefore, local files that can be accessed at the time the job is submitted are not automatically available to the actual running job. In these cases, in order for your job to run on the Hadoop cluster, there needs to be some way for these external files to be copied to the cluster. This is true when:

- Additional JAR files are required to be in `CLASSPATH`
- Various other files are needed, containing parameters, lookup data, and so on

Clearly, it would be possible to just copy these files to every node, but this quickly becomes unwieldy. The files could also be copied to HDFS, but this makes versioning difficult since a using job must know what file name to use. To solve this problem, Hadoop provides a facility called **distributed cache**, which allows these files to be automatically copied to wherever the task will run.

The distributed cache component allows one to specify various files that should be copied to nodes that will run our tasks. This includes JAR files, regular files, and compressed files (such as ZIP files that contain multiple files organized into directories). In order to use the `DistributedCache`, the following lines of code can be used:

```java
URI parms = new URI("parms.txt#parms");
DistributedCache.addCacheFile(parms, job);
DistributedCache.addCacheArchive(new URI("map.zip", job);
DistributedCache.addFileToClassPath(new Path("lib.jar"), job);
```
Note, the movement of `parms.txt` into a distributed cache and the strange `#parms` syntax at the end. This causes Hadoop to copy this file to each node where processes will run, and then to symbolically link it to a simple file name of `parms`, which can then be opened and read typically in a `setup()` method. This is powerful because it frees our tasks from needing to know the real file name. We could easily send a `newparms.txt` file call by only changing our job code, and then the underlying mappers and reducers would never need to know this.

When dealing with multiple JAR files, there is another technique, which is often used, called a **FAT JAR**. This is a single JAR file where all the required JAR files are unpacked and repacked. This can even include the required Cascading JAR files. While unwieldy, this is still a very useful technique, and it can save you debugging time when you get the dreaded `java.lang.ClassNotFound` exception.

**Counters**

Hadoop also provides instrumentation: the ability to define counters that can be used for debugging, performance assessment, and general visibility into the workload being processed. Here, we must note that jobs that run in a cluster are largely out of your control. They cannot use typical development techniques, such as debugging breakpoints. Additionally, since there are a lot of them running simultaneously, even using the ubiquitous `System.out.println` function is problematic, since so many output consoles are being captured. Counters are easy to use, as shown in the following example:

```java
public enum COUNTERS {
    RECORDS_WITH_ERRORS
}
```

Then in a mapper, reducer, and so on:

```java
context.getCounter(RECORDS_WITH_ERRORS).increment(1L);
```

And later in our job:

```java
System.out.printf("Errors: %d\n",
    counters.findCounter(COUNTERS.RECORDS_WITH_ERRORS).getValue());
```

**YARN – MapReduce version 2**

As can be seen in the architecture, the JobTracker is a single point of failure. Additionally, it performs a tremendous amount of work. It handles job submission, job scheduling (including finding and allocating slots where the parallel parts of each job run), and also tracking the progress of the job itself. It became a major bottleneck over time. As a result, version 2 of Hadoop now splits job scheduling and job execution into separate components.
In YARN, jobs are submitted to the **Resource Manager**. This server tracks all active nodes and their resources (such as CPU, memory, disk, and so on). When a job is to be dispatched, the job itself gives an indication of the resources it would like to use. This is a negotiation though, and the job is given what can be made available to it, in the form of a set of nodes where its tasks can run and some other guidance for what it can use. Then, the Resource Manager starts an Application Master. This process runs on a slave node and manages the entire job. It dispatches the mappers and the reducers. It also receives progress notifications. Each mapper and reducer runs inside a JVM container.

So, we can see that the old JobTracker is now split into Application Master and Node Manager. The **Application Master** offloads the management of the running job. This reduces the points of failure, since the failure of an Application Master will only kill its tasks.

One other significant aspect of YARN is that it does much more than just run MapReduce. In fact, YARN was designed to run arbitrary frameworks. For instance, both Spark and Tez can run on YARN.

What we are seeing here is the emergence of an application stack. YARN now forms the basis for resource management and scheduling/dispatching functions. Frameworks, such as MapReduce, Tez, Spark, and others, provide an application execution framework. After this, application development frameworks, such as Cascading, run within the application execution environment. It is this separation of concerns that is driving innovation, reducing the complexity of development, and providing upward compatibility by freeing the tight coupling that the original MapReduce imposed.

![YARN architecture](image-url)
Make no mistake, YARN is more complicated. However, frameworks are also emerging to aid developers. Additionally, existing MapReduce jobs can run without change on YARN. So, the addition of YARN, has provided a major enhancement to Hadoop. It allows scalability far beyond what version 1 could achieve. It has all but eliminated the single points of failure, and it now provides consideration for the resources that are being requested by the job. Also, YARN is being adopted rapidly. First, adopters chose it to build infrastructure that required persistent servers and better scalability. For instance, Spark and Tez (see the Beyond MapReduce section that follows) can now run on top of YARN. Given that YARN can seamlessly support legacy MapReduce applications, its adoption is now occurring at the application level.

A simple MapReduce job
Let's take a look at a simple MapReduce job. Let's look at a simple task. We are going to compute a weighted average of prices for some products. We will have data that represents price changes over time for some set of products. This is how the data looks:

```
product-id,date,price
1,1/1/2014,10.00
1,6/1/2014,11.00
1,11/26/2014,9.99
1,12/31/2014,11.99
```

Product number 1 has gone through several markups and markdowns. We seek to determine the average price over the low date and high date within the data for each product. In order to compute our average, we will need records that arrive at our reducer and are sorted both by the product ID date. For this, we will use the following classes:

- **MRJob**: This is a program that submits a job to the Hadoop cluster for execution
- **MRMapper**: This is a mapper that handles each data split and formats the data, in such a way that reducers can perform counting
- **MRReducer**: This is a reducer that performs a summary
- **MRPartitioner**: This is a class that ensures that all identical keys get routed to the same reducer
- **MRComparator**: This is a class that compares the correct portions of the composite keys so that they can be sorted correctly
Let's start by looking at a job that will submit work to Hadoop. This job is provided with command-line arguments that specify input and output directories, as well as any other parameters that may be required:

```java
public class MRJob extends Configured implements Tool {
    /**
     * @param args  input_directory_name (required, HDFS)
     *              output_directory_name (required, HDFS)
     */
    public static void main(String[] args) throws Exception {
        int rc = ToolRunner.run(new MRJob(), args);
        System.exit(rc);
    }
    @Override
    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        Job job = Job.getInstance(conf);
        job.setJarByClass(MRJob.class);
        job.setJobName("MRJob");
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(MRMapper.class);
        job.setReducerClass(MRReducer.class);
        job.setPartitionerClass(MRPartitioner.class);
        job.setMapOutputKeyClass(org.apache.hadoop.io.Text.class);
        job.setMapOutputValueClass(org.apache.hadoop.io.Text.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.setNumReduceTasks(8);
        int rc = (job.waitForCompletion(true) ? 1 : 0);
        return rc;
    }
}
```
We can make a few notes here now. The job itself looks much like the standard Java program by implementing a static \texttt{main()} method. It is passed \texttt{String args[]}, containing the parsed command line. Only two parameters are specified: the input directory and the output directory. Both are assumed to be HDFS by default. However, this class implements the \texttt{Tool} interface, and uses a static \texttt{ToolRunner} call to start execution. Use this form to save yourself a lot of trouble. \texttt{ToolRunner} implements a lot of good functionalities, and specifically, it is responsible for handling generic command-line arguments. These are specialized parameters that can specify additional JAR files that must be placed into the Java \texttt{CLASSPATH}. Other files and compressed archives should also be copied there, and also various parameters that are placed directly into the Hadoop Configuration container. The actual job submission then occurs in the \texttt{run()} method of this class. We can see that this is where each class is defined and will be used as mappers, reducers, partitioners, and so on.

First, let's look at a mapper. Note the usage of \texttt{LongWritable} as the key. This effectively passes the record offset as the key of the data record being processed.

```java
// Package and import statements omitted
public class MRMapper extends Mapper<LongWritable,Text,Text,Text> {
    private static String INSEP = ",";
    private static String OUTSEP = "#";
    private Map<String, String> recordMap;
    @Override
    void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        // Field 0 is product ID, field 1 is date, field 3 is price
        String[] fields[] = value.toString().split(INSEP);
        // Our KEY is product-ID#date
        // Our VALUE is date,price
        context.write(new Text(fields[0] + OUTSEP + fields[1]),
            new Text(fields[1] + INSEP + fields[2]));
    }
}
```

This input definition and supplying the record offset as the key is the default behavior of the \texttt{FileInputFormat} setting in the job. Different input formats can be used to change this behavior.
Now, let's look at a reducer. It will receive a single product key with an iterator of all the date, price values that are sorted by increasing the date:

```java
// Package and import statements omitted
public class MRReducer extends Reducer<Text, Text, Text, Text> {
    // Allocate our objects once and reuse them!
    Text outKey = new Text();
    Text outText = new Text();
    int sum;
    @Override
    void reduce(Text key, Iterable<Text> values, 
        Context context) throws IOException, InterruptedException {
        // Split our composite key to get just the product ID.
        String[] prodDef = key.toString().split(SEP);
        double average_price;
        // Compute the weighted average of prices based on number of
        // days difference between sorted records
        for (Text v : values) {
            // Computation of weighted average
        }
        outKey.set(prodDef[0]);
        outText.set(new Text(Double.toString(average_price)));
        context.write(outKey, outText);
    }
}
```

Next, let's look at the partitioner. It will route records based just on the product number so that all identical products arrive at the same reducer together to be group:

```java
// Package and import statements omitted
public class MRPartitioner extends Partitioner<Text, Text> implements Configurable {
    @Override
    public int getPartition(Text key, Text value, int numReduceTasks) {
        private static String SEP = "#";
        String[] partStr = key.toString().split(SEP);
        int prodNum = Integer.parseInt(partStr[0]);
        return prodNum % numReduceTasks;
    }
}
```
Lastly, we have the comparator. This compares two composite keys. If the product numbers are different, it performs just like a normal compare. If they are the same, it compares their date fields so that the results will be sorted by the date:

```java
public class MRComparator implements RawComparator<Text> {
    private static String SEP = "#";
    @Override
    public int compare(byte[] b1, int s1, int l1,
                        byte[] b2, int s2, int l2)
    {
        return WritableComparator.compareBytes(b1, s1, l1, b2, s2, l2);
    }

    @Override
    public int compare(Text o1, Text o2) {
        String s1[], s2[];
        s1 = o1.toString().split(SEP);
        s2 = o2.toString().split(SEP);
        int prod1 = Integer.parseInt(s1[0]);
        int prod2 = Integer.parseInt(s2[0]);
        if(prod1 == prod2) {
            return MRUtil.compareDates(s1[1], s2[1]); // See code
        }
        if (prod1 < prod2)
            return -1;
        else
            return 1;
    }
}
```

Whew! This is a lot of code to do something that seems pretty easy. There must be a better way! Cascading provides us with a mechanism that will greatly simplify these tasks.
Beyond MapReduce

With Hadoop, there are really only two primary job types that are supported by the framework: mapper-only jobs and mapper/reducer jobs. This has become limiting as we attempt to maximize performance within a cluster. During many of the Hadoop phases of processing, files are written to disk and then immediately reread. This is inefficient. Also, all job structures usually exist as Map -> Reduce -> Map -> Reduce …. Every job must start with a mapper, even if it’s IdentityMapper that simply reads a record and writes the same record. In this case, the only purpose that the mapper actually serves is to perform the file split.

Other job sequences do exist that allow Map -> Map -> … -> Map -> Reduce, but discussion of these is beyond the scope of this book. See the ChainMapper class for more information on this.

As a result, some new systems are seeking to become replacements for Hadoop, or at the very least, to become part of the "plumbing" that Hadoop provides. There are many of these, but two are noteworthy; they're gaining adoption and are proving to provide major performance improvements. We provide a very brief description of these here:

- **Apache Tez** is a new architecture that follows the basic paradigm of MapReduce, but allows for more generic processing than simple mapper and reducer functionalities, and it also provides finer grained control over how data is interchanged and persisted.

- **Apache Spark** is also a new paradigm that is a total replacement for MapReduce. It has its own controlling framework of servers that are persistent, that is, they can remain operational and wait for new data when it becomes available. It provides very generic processors, and includes in-memory computing through its resilient distributed datasets (RDD). Interestingly enough, Spark now can run using YARN as its resource manager. Spark is written in Scala, a very powerful programming language that runs in a JVM.
The Cascading framework

Now let's look very briefly at Cascading. Cascading provides a much higher-level API in addition to MapReduce, and as we shall soon see, many other types of big data "fabrics," such as Tez, Spark, and Storm. Additionally, Cascading provides an abstraction that insulates our code from the underlying fabric and the data source type and protocol. It also gives us an integrated orchestration layer that allows us to build sophisticated sequences of jobs, and it provides rich out-of-the-box functionalities. MapReduce programmers have realized very quickly that much of the code that write is dedicated to some very basic things, such as preparing the sort keys and handling the comparisons for the sort. As we saw previously, MapReduce is verbose! Even such a simple task requires five classes and well over 100 lines of code.

Fundamentally, in this code, little is actually occurring other than key preparation, partitioning, sorting, and counting. Cascading handles this by assigning the input and output sources and destinations, creating the sort keys, performing the sort, and then performing the counting. It accomplished all this in a single class with an astonishing 20 lines of code! We need to know a little bit more about Cascading, so after we gain this understanding in a later chapter, we will look at this code in detail and then return to compare the differences, outlining the efficiency gains.

To a large degree, Cascading hides much of the complexity of MapReduce and of many big data programming complexities in general. Now, to be perfectly clear, Cascading has its own set of complexities. It also provides a standardized approach that has a smaller surface area than all of Hadoop. Cascading is, in fact, a domain-specific language (DSL) for Hadoop that encapsulates map, reduce, partitioning, sorting, and analytical operations in a concise form. This DSL is written in a fluent style, and this makes coding and understanding of the resulting code line much easier.

A fluent interface (also sometimes known as a builder pattern) is one in which each call to a class returns an object (called its context) through which the method operates. This functional style allows for concise code to be written where the resulting lines of code resemble a "scripted language," as shown here:

```java
Company company = new Company("XYZ Corp");
company.setAddress("Company Address")
    .setPhone("123-456-7890")
    .addEmployee("Jim")
    .addEmployee("Sue");
```
The execution graph and flow planner

When Cascading executes its job code, it is really preparing an execution graph. This graph has as its vertices every process that must be performed. These are things, such as reading and transforming records, performing sorts, performing aggregations, and writing results. Its edges are in the form of all of the data exchanges that occur between these processing steps. After this graph is prepared, Cascading plans how it will be executed. The planner is specific to the underlying framework. In the preceding example, we use HadoopFlowConnector to do this. Herein lies the beauty of Cascading. There are other connectors.

LocalFlowConnector can run the job without needing Hadoop at all. It is simply run as a connected set of Java programs. Using this connector, a developer can test their code in isolation. This is very valuable for a developer.

In future, you can see how TezConnector, SparkConnector, and others can be created. So, what we've seen is that we can write one code line and then execute it on differing frameworks. We have magically freed our code from being frozen in place! We've now gain the ability to move to newer, more performant, and more feature-rich big data frameworks without requiring expensive rewrites.

How Cascading produces MapReduce jobs

After the execution graph is produced, the creation of MapReduce jobs is relatively easy. Most everything that we tell Cascading is translated into mappers, reducers, partitioners, and comparators. The lower-level semantics of performing record mapping, multi-field sorts, and so on are handled for us.

Additionally, as we shall soon see, Cascading provides a rich set of high-level functions to do basic ETL work, such as regular expression parsing, data transformation, data validation, error handling, and much more.

Summary

We have now learned what drives the need for big data technology. We have also learned about Hadoop, its components, how it is used, and how it differs from other traditional relational database systems. We have discussed what the future holds for similar technologies, and the problems that exist with Hadoop and MapReduce. Lastly, we took a brief look at Cascading and discussed how it addresses many of these problems, how Cascading can help preserve our investment in development, and how it can help us adapt to rapidly moving technological innovation that is evident in big data frameworks.

In the next chapter, we will take a much closer look at Cascading, its architecture, and how it can be used to solve typical problems in data processing.
Where to buy this book

You can buy Learning Cascading from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.