Chapter 9

Exploring the World of Hadoop

In This Chapter

- Discovering Hadoop and why it's so important
- Exploring the Hadoop Distributed File System
- Digging into Hadoop MapReduce
- Putting Hadoop to work

When you need to process big data sources, traditional approaches fall short. The volume, velocity, and variety of big data will bring most technologies to their knees, so new technologies had to be created to address this new challenge. MapReduce is one of those new technologies, but it is just an algorithm, a recipe for how to make sense of all the data. To get the most from MapReduce, you need more than just an algorithm. You need a collection of products and technologies designed to handle the challenges presented by big data.

Explaining Hadoop

Search engine innovators like Yahoo! and Google needed to find a way to make sense of the massive amounts of data that their engines were collecting. These companies needed to both understand what information they were gathering and how they could monetize that data to support their business model. Hadoop was developed because it represented the most pragmatic way to allow companies to manage huge volumes of data easily. Hadoop allowed big problems to be broken down into smaller elements so that analysis could be done quickly and cost-effectively.

By breaking the big data problem into small pieces that could be processed in parallel, you can process the information and regroup the small pieces to present results.

Hadoop (`http://hadoop.apache.org`) was originally built by a Yahoo! engineer named Doug Cutting and is now an open source project managed by the Apache Software Foundation. It is made available under the Apache License v2.0. Along with other projects that we examine in Chapter 10, Hadoop is a fundamental building block in our desire to capture and process big data. Hadoop is designed to parallelize data processing across computing nodes to speed computations and hide latency. At its core, Hadoop has two primary components:

- **Hadoop Distributed File System:** A reliable, high-bandwidth, low-cost, data storage cluster that facilitates the management of related files across machines.
- **MapReduce engine:** A high-performance parallel/distributed data-processing implementation of the MapReduce algorithm.

Hadoop is designed to process huge amounts of structured and unstructured data (terabytes to petabytes) and is implemented on racks of commodity servers as a Hadoop cluster. Servers can be added or removed from the cluster dynamically because Hadoop is designed to be "self-healing." In other words, Hadoop is able to detect changes, including failures, and adjust to those changes and continue to operate without interruption.

We now take a closer look at the Hadoop Distributed File System (HDFS) and MapReduce as implemented in Hadoop.

Understanding the Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System is a versatile, resilient, clustered approach to managing files in a big data environment. HDFS is not the final destination for files. Rather, it is a data service that offers a unique set of capabilities needed when data volumes and velocity are high. Because the data is written once and then read many times thereafter, rather than the constant read-writes of other file systems, HDFS is an excellent choice for supporting big data analysis. The service includes a "NameNode" and multiple "data nodes" running on a commodity hardware cluster and provides the highest levels of performance when the entire cluster is in the same physical rack in the data center. In essence, the NameNode keeps track of where data is physically stored. Figure 9-1 depicts the basic architecture of HDFS.

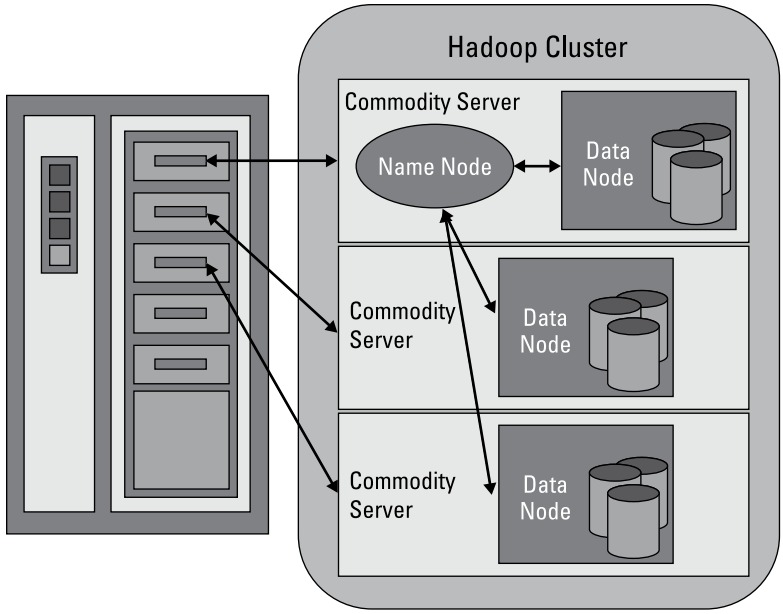

Figure 9-1: How a Hadoop cluster is mapped to hardware.

NameNodes

HDFS works by breaking large files into smaller pieces called *blocks.* The blocks are stored on data nodes, and it is the responsibility of the NameNode to know what blocks on which data nodes make up the complete file. The NameNode also acts as a "traffic cop," managing all access to the files, including reads, writes, creates, deletes, and replication of data blocks on the data nodes. The complete collection of all the files in the cluster is sometimes referred to as the file system *namespace.* It is the NameNode's job to manage this namespace.

Even though a strong relationship exists between the NameNode and the data nodes, they operate in a "loosely coupled" fashion. This allows the cluster elements to behave dynamically, adding (or subtracting) servers as the demand increases (or decreases). In a typical configuration, you find one NameNode and possibly a data node running on one physical server in the rack. Other servers run data nodes only.

Data nodes are not very smart, but the NameNode is. The data nodes constantly ask the NameNode whether there is anything for them to do. This continuous behavior also tells the NameNode what data nodes are out there and how busy they are. The data nodes also communicate among themselves

so that they can cooperate during normal file system operations. This is necessary because blocks for one file are likely to be stored on multiple data nodes. Since the NameNode is so critical for correct operation of the cluster, it can and should be replicated to guard against a single point failure.

Data nodes

Data nodes are not smart, but they are resilient. Within the HDFS cluster, data blocks are replicated across multiple data nodes and access is managed by the NameNode. The replication mechanism is designed for optimal efficiency when all the nodes of the cluster are collected into a rack. In fact, the NameNode uses a "rack ID" to keep track of the data nodes in the cluster. HDFS clusters are sometimes referred to as being "rack-aware." Data nodes also provide "heartbeat" messages to detect and ensure connectivity between the NameNode and the data nodes. When a heartbeat is no longer present, the NameNode unmaps the data node from the cluster and keeps on operating as though nothing happened. When the heartbeat returns (or a new heartbeat appears), it is added to the cluster transparently with respect to the user or application.

As with all file systems, data integrity is a key feature. HDFS supports a number of capabilities designed to provide data integrity. As you might expect, when files are broken into blocks and then distributed across different servers in the cluster, any variation in the operation of any element could affect data integrity. HDFS uses transaction logs and checksum validation to ensure integrity across the cluster.

Transaction logs are a very common practice in file system and database design. They keep track of every operation and are effective in auditing or rebuilding of the file system should something untoward occur.

Checksum validations are used to guarantee the contents of files in HDFS. When a client requests a file, it can verify the contents by examining its checksum. If the checksum matches, the file operation can continue. If not, an error is reported. Checksum files are hidden to help avoid tampering.

Data nodes use local disks in the commodity server for persistence. All the data blocks are stored locally, primarily for performance reasons. Data blocks are replicated across several data nodes, so the failure of one server may not necessarily corrupt a file. The degree of replication, the number of data nodes, and the HDFS namespace are established when the cluster is implemented. Because HDFS is dynamic, all parameters can be adjusted during the operation of the cluster.

Under the covers of HDFS

Big data brings the big challenges of volume, velocity, and variety. As covered in the previous sections, HDFS addresses these challenges by breaking files into a related collection of smaller blocks. These blocks are distributed among the data nodes in the HDFS cluster and are managed by the NameNode. Block sizes are configurable and are usually 128 megabytes (MB) or 256MB, meaning that a 1GB file consumes eight 128MB blocks for its basic storage needs. HDFS is resilient, so these blocks are replicated throughout the cluster in case of a server failure. How does HDFS keep track of all these pieces? The short answer is file system *metadata.*

Metadata is defined as "data about data." Software designers have been using metadata for decades under several names like data dictionary, metadata directory, and more recently, tags. Think of HDFS metadata as a template for providing a detailed description of the following:

- When the file was created, accessed, modified, deleted, and so on
- Where the blocks of the file are stored in the cluster
- Who has the rights to view or modify the file
- How many files are stored on the cluster
- How many data nodes exist in the cluster
- The location of the transaction log for the cluster

HDFS metadata is stored in the NameNode, and while the cluster is operating, all the metadata is loaded into the physical memory of the NameNode server. As you might expect, the larger the cluster, the larger the metadata footprint. For best performance, the NameNode server should have lots of physical memory and, ideally, lots of solid-state disks. The more the merrier, from a performance point of view.

As we cover earlier in the chapter, the data nodes are very simplistic. They are servers that contain the blocks for a given set of files. It is reasonable to think of data nodes as "block servers" because that is their primary function. What exactly does a block server do? Check out the following list:

- Stores (and retrieves) the data blocks in the local file system of the server. HDFS is available on many different operating systems and behaves the same whether on Windows, Mac OS, or Linux.
- Stores the metadata of a block in the local file system based on the metadata template in the NameNode.
- Performs periodic validations of file checksums.

- Sends regular reports to the NameNode about what blocks are available for file operations.
- Provides metadata and data to clients on demand. HDFS supports direct access to the data nodes from client application programs.
- Forwards data to other data nodes based on a "pipelining" model.

Block placement on the data nodes is critical to data replication and support for data pipelining. HDFS keeps one replica of every block locally. It then places a second replica on a different rack to guard against a complete rack failure. It also sends a third replica to the same remote rack, but to a different server in the rack. Finally, it can send additional replicas to random locations in local or remote clusters. HDFS is serious about data replication and resiliency. Fortunately, client applications do not need to worry about where all the blocks are located. In fact, clients are directed to the nearest replica to ensure highest performance.

HDFS supports the capability to create data pipelines. A *pipeline* is a connection between multiple data nodes that exists to support the movement of data across the servers. A client application writes a block to the first data node in the pipeline. The data node takes over and forwards the data to the next node in the pipeline; this continues until all the data, and all the data replicas, are written to disk. At this point, the client repeats the process by writing the next block in the file. As you see later in this chapter, this is an important feature for Hadoop MapReduce.

With all these files and blocks and servers, you might wonder how things are kept in balance. Without any intervention, it is possible for one data node to become congested while another might be nearly empty. HDFS has a "rebalancer" service that's designed to address these possibilities. The goal is to balance the data nodes based on how full each set of local disks might be. The rebalancer runs while the cluster is active and can be throttled to avoid congestion of network traffic. After all, HDFS needs to manage the files and blocks first and then worry about how balanced the cluster needs to be.

The rebalancer is effective, but it does not have a great deal of built-in intelligence. For example, you can't create access or load patterns and have the rebalancer optimize for those conditions. Nor will it identify data "hot spots" and correct for them. Perhaps these features will be offered in future versions of HDFS.

Hadoop MapReduce

To fully understand the capabilities of Hadoop MapReduce, we need to differentiate between MapReduce (the algorithm) and an implementation of MapReduce. Hadoop MapReduce is an implementation of the algorithm

developed and maintained by the Apache Hadoop project. It is helpful to think about this implementation as a MapReduce engine, because that is exactly how it works. You provide input (fuel), the engine converts the input into output quickly and efficiently, and you get the answers you need. You are using Hadoop to solve business problems, so it is necessary for you to understand how and why it works. So, we take a look at the Hadoop implementation of MapReduce in more detail.

Hadoop MapReduce includes several stages, each with an important set of operations helping to get to your goal of getting the answers you need from big data. The process starts with a user request to run a MapReduce program and continues until the results are written back to the HDFS. Figure 9-2 illustrates how MapReduce performs its tasks.

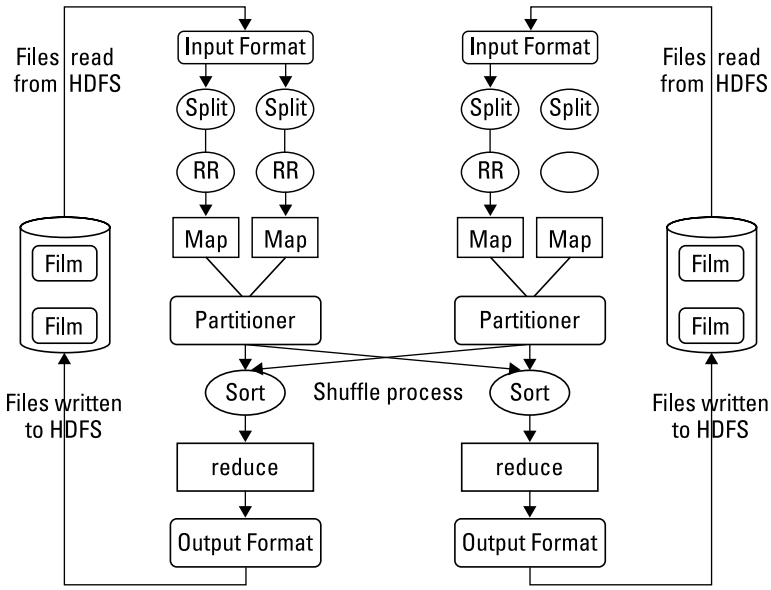

Figure 9-2: Workflow and data movement in a small Hadoop cluster.

HDFS and MapReduce perform their work on nodes in a cluster hosted on racks of commodity servers. To simplify the discussion, the diagram shows only two nodes.

Getting the data ready

When a client requests a MapReduce program to run, the first step is to locate and read the input file containing the raw data. The file format is completely arbitrary, but the data must be converted to something the program can process. This is the function of InputFormat and RecordReader (RR).

InputFormat decides how the file is going to be broken into smaller pieces for processing using a function called InputSplit. It then assigns a RecordReader to transform the raw data for processing by the map. If you read the discussion of map in Chapter 8, you know it requires two inputs: a key and a value. Several types of RecordReaders are supplied with Hadoop, offering a wide variety of conversion options. This feature is one of the ways that Hadoop manages the huge variety of data types found in big data problems.

Let the mapping begin

Your data is now in a form acceptable to map. For each input pair, a distinct instance of map is called to process the data. But what does it do with the processed output, and how can you keep track of them? Map has two additional capabilities to address the questions. Because map and reduce need to work together to process your data, the program needs to collect the output from the independent mappers and pass it to the reducers. This task is performed by an OutputCollector. A Reporter function also provides information gathered from map tasks so that you know when or if the map tasks are complete.

All this work is being performed on multiple nodes in the Hadoop cluster simultaneously. You may have cases where the output from certain mapping processes needs to be accumulated before the reducers can begin. Or, some of the intermediate results may need to be processed before reduction. In addition, some of this output may be on a node different from the node where the reducers for that specific output will run. The gathering and shuffling of intermediate results are performed by a partitioner and a sort. The map tasks will deliver the results to a specific partition as inputs to the reduce tasks. After all the map tasks are complete, the intermediate results are gathered in the partition and a shuffling occurs, sorting the output for optimal processing by reduce.

Reduce and combine

For each output pair, reduce is called to perform its task. In similar fashion to map, reduce gathers its output while all the tasks are processing. Reduce can't begin until all the mapping is done, and it isn't finished until all instances are complete. The output of reduce is also a key and a value. While this is necessary for reduce to do its work, it may not be the most effective output format for your application. Hadoop provides an OutputFormat feature, and it works very much like InputFormat. OutputFormat takes the key-value pair and organizes the output for writing to HDFS. The last task is to actually write the data to HDFS. This is performed by RecordWriter,

and it performs similarly to RecordReader except in reverse. It takes the OutputFormat data and writes it to HDFS in the form necessary for the requirements of the application program.

The coordination of all these activities was managed in earlier versions of Hadoop by a job scheduler. This scheduler was rudimentary, and as the mix of jobs changed and grew, it was clear that a different approach was necessary. The primary deficiency in the old scheduler was the lack of resource management. The latest version of Hadoop has this new capability, and we look at it more closely in Chapter 10.

Hadoop MapReduce is the heart of the Hadoop system. It provides all the capabilities you need to break big data into manageable chunks, process the data in parallel on your distributed cluster, and then make the data available for user consumption or additional processing. And it does all this work in a highly resilient, fault-tolerant manner. This is just the beginning. The Hadoop ecosystem is a large, growing set of tools and technologies designed specifically for cutting your big data problems down to size.

Chapter 10

The Hadoop Foundation and Ecosystem

In This Chapter

- Why the Hadoop ecosystem is foundational for big data
- Managing resources and applications with Hadoop YARN
- Storing big data with HBase
- Mining big data with Hive
- Interacting with the Hadoop ecosystem

As Chapter 9 explains, Hadoop MapReduce and Hadoop Distributed File System (HDFS) are powerful technologies designed to address big data challenges. That's the good news. The bad news is that you really need to be a programmer or data scientist to be able to get the most out of these elemental components. Enter the Hadoop ecosystem. For several years and for the foreseeable future, open source as well as commercial developers all over the world have been building and testing tools to increase the adoption and usability of Hadoop. Many are working on bits of the ecosystem and offering their enhancements back to the Apache project. This constant flow of fixes and improvements helps to drive the entire ecosystem forward in a controlled and secure manner.

In this chapter, you take a look at the various technologies that make up the Hadoop ecosystem.

Building a Big Data Foundation with the Hadoop Ecosystem

Trying to tackle big data challenges without a toolbox filled with technology and services is like trying to empty the ocean with a spoon. As core components, Hadoop MapReduce and HDFS are constantly being improved and

provide great starting points, but you need something more. The Hadoop ecosystem provides an ever-expanding collection of tools and technologies specifically created to smooth the development, deployment, and support of big data solutions. Before we look at the key components of the ecosystem, let's take a moment to discuss the Hadoop ecosystem and the role it plays on the big data stage.

No building is stable without a foundation. While important, stability is not the only important criterion in a building. Each part of the building must support its overall purpose. The walls, floors, stairs, electrical, plumbing, and roof need to complement each other while relying on the foundation for support and integration. It is the same with the Hadoop ecosystem. The foundation is MapReduce and HDFS. They provide the basic structure and integration services needed to support the core requirements of big data solutions. The remainder of the ecosystem provides the components you need to build and manage purpose-driven big data applications for the real world.

In the absence of the ecosystem it would be incumbent on developers, database administrators, system and network managers, and others to identify and agree on a set of technologies to build and deploy big data solutions. This is often the case when businesses want to adapt new and emerging technology trends. The chore of cobbling together technologies in a new market is daunting. That is why the Hadoop ecosystem is so fundamental to the success of big data. It is the most comprehensive collection of tools and technologies available today to target big data challenges. The ecosystem facilitate the creation of new opportunities for the widespread adoption of big data by businesses and organizations.

Managing Resources and Applications with Hadoop YARN

Job scheduling and tracking are integral parts of Hadoop MapReduce. The early versions of Hadoop supported a rudimentary job and task tracking system, but as the mix of work supported by Hadoop changed, the scheduler could not keep up. In particular, the old scheduler could not manage non-MapReduce jobs, and it was incapable of optimizing cluster utilization. So a new capability was designed to address these shortcomings and offer more flexibility, efficiency, and performance.

Yet Another Resource Negotiator (YARN) is a core Hadoop service providing two major services:

- Global resource management (ResourceManager)
- Per-application management (ApplicationMaster)

The ResourceManager is a master service and control NodeManager in each of the nodes of a Hadoop cluster. Included in the ResourceManager is Scheduler, whose sole task is to allocate system resources to specific running applications (tasks), but it does not monitor or track the application's status. All the required system information is stored in a Resource Container. It contains detailed CPU, disk, network, and other important resource attributes necessary for running applications on the node and in the cluster.

Each node has a NodeManager slaved to the global ResourceManager in the cluster. The NodeManager monitors the application's usage of CPU, disk, network, and memory and reports back to the ResourceManager. For each application running on the node there is a corresponding ApplicationMaster. If more resources are necessary to support the running application, the ApplicationMaster notifies the NodeManager and the NodeManager negotiates with the ResourceManager (Scheduler) for the additional capacity on behalf of the application. The NodeManager is also responsible for tracking job status and progress within its node.

Storing Big Data with HBase

HBase is a distributed, nonrelational (columnar) database that utilizes HDFS as its persistence store. It is modeled after Google BigTable and is capable of hosting very large tables (billions of columns/rows) because it is layered on Hadoop clusters of commodity hardware. HBase provides random, real-time read/write access to big data. HBase is highly configurable, providing a great deal of flexibility to address huge amounts of data efficiently. Now take a look at how HBase can help address your big data challenges.

HBase is a columnar database, so all data is stored into tables with rows and columns similar to relational database management systems (RDBMSs). The intersection of a row and a column is called a cell. One important difference between HBase tables and RDBMS tables is versioning. Each cell value includes a "version" attribute, which is nothing more than a timestamp uniquely identifying the cell. Versioning tracks changes in the cell and makes it possible to retrieve any version of the contents should it become necessary. HBase stores the data in cells in decreasing order (using the timestamp), so a read will always find the most recent values first.

Columns in HBase belong to a column family. The column family name is used as a prefix to identify members of its family. For example, *fruits:apple* and *fruits:banana* are members of the *fruits* column family. HBase implementations are tuned at the column family level, so it is important to be mindful of how you are going to access the data and how big you expect the columns to be.

The rows in HBase tables also have a key associated with them. The structure of the key is very flexible. It can be a computed value, a string, or even another data structure. The key is used to control access to the cells in the row, and they are stored in order from low value to high value.

All of these features together make up the schema. The schema is defined and created before any data can be stored. Even so, tables can be altered and new column families can be added after the database is up and running. This extensibility is extremely useful when dealing with big data because you don't always know about the variety of your data streams.

Mining Big Data with Hive

Hive is a batch-oriented, data-warehousing layer built on the core elements of Hadoop (HDFS and MapReduce). It provides users who know SQL with a simple SQL-lite implementation called HiveQL without sacrificing access via mappers and reducers. With Hive, you can get the best of both worlds: SQL-like access to structured data and sophisticated big data analysis with MapReduce.

Unlike most data warehouses, Hive is not designed for quick responses to queries. In fact, queries can take several minutes or even hours depending on the complexity. As a result, Hive is best used for data mining and deeper analytics that do not require real-time behaviors. Because it relies on the Hadoop foundation, it is very extensible, scalable, and resilient, something that the average data warehouse is not.

Hive uses three mechanisms for data organization:

- **Tables:** Hive tables are the same as RDBMS tables consisting of rows and columns. Because Hive is layered on the Hadoop HDFS, tables are mapped to directories in the file system. In addition, Hive supports tables stored in other native file systems.
- **Partitions:** A Hive table can support one or more partitions. These partitions are mapped to subdirectories in the underlying file system and represent the distribution of data throughout the table. For example, if a table is called *autos,* with a key value of *12345* and a maker value *Ford,* the path to the partition would be `/hivewh/autos/kv=12345/Ford`.
- **Buckets:** In turn, data may be divided into buckets. Buckets are stored as files in the partition directory in the underlying file system. The buckets are based on the hash of a column in the table. In the preceding example, you might have a bucket called *Focus,* containing all the attributes of a Ford Focus auto.

Hive metadata is stored externally in the "metastore." The metastore is a relational database containing the detailed descriptions of the Hive schema, including column types, owners, key and value data, table statistics, and so on. The metastore is capable of syncing catalog data with other metadata services in the Hadoop ecosystem.

Hive supports an SQL-like language called HiveQL. HiveQL supports many of the SQL primitives, such as select, join, aggregate, union all, and so on. It also supports multitable queries and inserts by sharing the input data within a single HiveQL statement. HiveQL can be extended to support user-defined aggregation, column transformation, and embedded MapReduce scripts.

Interacting with the Hadoop Ecosystem

Writing programs or using specialty query languages are not the only ways you interact with the Hadoop ecosystem. IT teams that manage infrastructures need to control Hadoop and the big data applications created for it. As big data becomes mainstream, non-technical professionals will want to try to solve business problems with big data. Look at some examples from the Hadoop ecosystem that help these constituencies.

Pig and Pig Latin

The power and flexibility of Hadoop are immediately visible to software developers primarily because the Hadoop ecosystem was built by developers, for developers. However, not everyone is a software developer. Pig was designed to make Hadoop more approachable and usable by nondevelopers. Pig is an interactive, or script-based, execution environment supporting Pig Latin, a language used to express data flows. The Pig Latin language supports the loading and processing of input data with a series of operators that transform the input data and produce the desired output.

The Pig execution environment has two modes:

- **Local mode:** All scripts are run on a single machine. Hadoop MapReduce and HDFS are not required.
- **Hadoop:** Also called MapReduce mode, all scripts are run on a given Hadoop cluster.

Under the covers, Pig creates a set of *map* and *reduce* jobs. The user is absolved from the concerns of writing code, compiling, packaging, submitting, and retrieving the results. In many respects, Pig is analogous to SQL in

the RDBMS world. The Pig Latin language provides an abstract way to get answers from big data by focusing on the data and not the structure of a custom software program. Pig makes prototyping very simple. For example, you can run a Pig script on a small representation of your big data environment to ensure that you are getting the desired results before you commit to processing all the data.

Pig programs can be run in three different ways, all of them compatible with local and Hadoop mode:

- **Script:** Simply a file containing Pig Latin commands, identified by the `.pig` suffix (for example, `file.pig` or `myscript.pig`). The commands are interpreted by Pig and executed in sequential order.
- **Grunt:** Grunt is a command interpreter. You can type Pig Latin on the grunt command line and Grunt will execute the command on your behalf. This is very useful for prototyping and "what if" scenarios.
- **Embedded:** Pig programs can be executed as part of a Java program.

Pig Latin has a very rich syntax. It supports operators for the following operations:

- Loading and storing of data
- Streaming data
- Filtering data
- Grouping and joining data
- Sorting data
- Combining and splitting data

Pig Latin also supports a wide variety of types, expressions, functions, diagnostic operators, macros, and file system commands.

To get more examples, visit the Pig website within Apache.com. It is a rich resource that will provide you with all the details: `http://pig.apache.org`.

Sqoop

Many businesses store information in RDBMSs and other data stores, so they need a way to move data back and forth from these data stores to Hadoop. While it is sometimes necessary to move the data in real time, it is most often necessary to load or unload data in bulk. Sqoop (SQL-to-Hadoop) is a tool that offers the capability to extract data from non-Hadoop data stores,

transform the data into a form usable by Hadoop, and then load the data into HDFS. This process is called ETL, for Extract, Transform, and Load. While getting data into Hadoop is critical for processing using MapReduce, it is also critical to get data out of Hadoop and into an external data source for use in other kinds of application. Sqoop is able to do this as well.

Like Pig, Sqoop is a command-line interpreter. You type Sqoop commands into the interpreter and they are executed one at a time. Four key features are found in Sqoop:

- **Bulk import:** Sqoop can import individual tables or entire databases into HDFS. The data is stored in the native directories and files in the HDFS file system.
- **Direct input:** Sqoop can import and map SQL (relational) databases directly into Hive and HBase.
- **Data interaction:** Sqoop can generate Java classes so that you can interact with the data programmatically.
- **Data export:** Sqoop can export data directly from HDFS into a relational database using a target table definition based on the specifics of the target database.

Sqoop works by looking at the database you want to import and selecting an appropriate import function for the source data. After it recognizes the input, it then reads the metadata for the table (or database) and creates a class definition of your input requirements. You can force Sqoop to be very selective so that you get just the columns you are looking for before input rather than doing an entire input and then looking for your data. This can save considerable time. The actual import from the external database to HDFS is performed by a MapReduce job created behind the scenes by Sqoop.

Sqoop is another effective tool for nonprogrammers. The other important item to note is the reliance on underlying technologies like HDFS and MapReduce. You see this repeatedly throughout the element of the Hadoop ecosystem.

Zookeeper

Hadoop's greatest technique for addressing big data challenges is its capability to divide and conquer. After the problem has been divided, the conquering relies on the capability to employ distributed and parallel processing techniques across the Hadoop cluster. For some big data problems, the interactive tools are unable to provide the insights or timeliness required to make business decisions. In those cases, you need to create distributed

applications to solve those big data problems. Zookeeper is Hadoop's way of coordinating all the elements of these distributed applications.

Zookeeper as a technology is actually simple, but its features are powerful. Arguably, it would be difficult, if not impossible, to create resilient, fault-tolerant distributed Hadoop applications without it. Some of the capabilities of Zookeeper are as follows:

- **Process synchronization:** Zookeeper coordinates the starting and stopping of multiple nodes in the cluster. This ensures that all processing occurs in the intended order. When an entire process group is complete, then and only then can subsequent processing occur.
- **Configuration management:** Zookeeper can be used to send configuration attributes to any or all nodes in the cluster. When processing is dependent on particular resources being available on all the nodes, Zookeeper ensures the consistency of the configurations.
- **Self-election:** Zookeeper understands the makeup of the cluster and can assign a "leader" role to one of the nodes. This leader/master handles all client requests on behalf of the cluster. Should the leader node fail, another leader will be elected from the remaining nodes.
- **Reliable messaging:** Even though workloads in Zookeeper are loosely coupled, you still have a need for communication between and among the nodes in the cluster specific to the distributed application. Zookeeper offers a publish/subscribe capability that allows the creation of a queue. This queue guarantees message delivery even in the case of a node failure.

Because Zookeeper is managing groups of nodes in service to a single distributed application, it is best implemented *across* racks. This is very different than the requirements for the cluster itself (within racks). The underlying reason is simple: Zookeeper needs to perform, be resilient, and be fault tolerant at a level above the cluster itself. Remember that a Hadoop cluster is already fault tolerant, so it will heal itself. Zookeeper just needs to worry about its own fault tolerance.

The Hadoop ecosystem and the supported commercial distributions are ever-changing. New tools and technologies are introduced, existing technologies are improved, and some technologies are retired by a (hopefully better) replacement. This one of the greatest advantages of open source. Another is the adoption of open source technologies by commercial companies. These companies enhance the products, making them better for everyone by offering support and services at a modest cost. This is how the Hadoop ecosystem has evolved and why it is a good choice for helping to solve your big data challenges.