1-The Age of the Data Product

1.1-What Is a Data Product?

The traditional answer to this question is usually "any application that combines data and algorithms." But frankly, if you're writing software and you're not combining data with algorithms, then what are you doing? After all, data is the currency of programming! More specifically, we might say that a data product is the combination of data with statistical algorithms that are used for inference or prediction. Many data scientists are also statisticians, and statistical methodologies are central to data science.

Armed with this definition, you could cite Amazon recommendations as an example of a data product. Amazon examines items you've purchased, and based on similar purchase behavior of other users, makes recommendations. In this case, order history data is combined with recommendation algorithms to make predictions about what you might purchase in the future. You might also cite Facebook's "People You May Know" feature because this product "shows you people based on mutual friends, work and education information ... [and] many other factors"—essentially using the combination of social network data with graph algorithms to infer members of communities.

Indeed, defining data products as simply the combination of data with statistical algorithms seems to limit data products to single software instances (e.g., a web application), which hardly seems a revolutionary economic force. Although we might point to Google or others as large-scale economic forces, the combination of a web crawler gathering a massive HTML corpus with the PageRank algorithm alone does not create a data economy. We know what an important role search plays in economic activity, so something must be missing from this first definition.

Mike Loukides defines a data product as follows:

> ***A data application acquires its value from the data itself, and creates more data as a result. It's not just an application with data; it's a data product.***

This is the revolution. A data product is an economic engine. It derives value from data and then produces more data, more value, in return. The data that it creates may fuel the generating product or it might lead to the creation of other data products that derive their value from that generated data. This is precisely what has led to the surplus of information and the resulting information revolution.

We describe data products as systems that *learn from data,* are *self-adapting,* and are *broadly applicable.* Under this definition, the Nest thermostat is a data product. It derives its value from sensor data, adapts how it schedules heating and cooling, and causes new sensor observations to be collected that validate the adaptation. Autonomous vehicles such as those being produced by Stanford's Autonomous Driving Team also fall into this category. The team's machine vision and pilot behavior simulation are the result of algorithms, so when the vehicle is in motion, it produces more data in the form of navigation and sensor data that can be used to improve the driving platform.

Data products are self-adapting, broadly applicable economic engines that derive their value from data and generate more data by influencing human behavior or by making inferences or predictions upon new data. Data products are not merely web applications and are rapidly becoming an essential component of almost every single domain of economic activity of the modern world. Because they are able to discover individual patterns in human activity, they drive decisions, whose resulting actions and influences are also recorded as new data.

1.2-Building Data Products at Scale with Hadoop

Josh Wills provides us with the following definition:

> ***Data Scientist: Person who is better at statistics than any software engineer and better at software engineering than any statistician.***

Certainly, this fits in well with the idea that a data product is simply the combination of data with statistical algorithms. Both software engineering and statistical knowledge are essential to data science. However, in an economy that demands products that derive their value from data and generate new data in return, we should say instead that as data scientists, it is our job to build data products.

Harlan Harris provides more detail about the incarnation of data products: they are built at the intersection of data, domain knowledge, software engineering, and analytics. Because data products are systems, they require an engineering skill set, usually in software, in order to build them. They are powered by data, so having data is a necessary requirement. Domain knowledge and analytics are the tools used to build the data engine, usually via experimentation, hence the "science" part of data science.

Because of the experimental methodology required, most data scientists will point to this typical analytical workflow:

> *Ingestion --->wrangling--->modeling--->reporting and visualization.*

Yet this so-called *data science pipeline* is completely human-powered, augmented by the use of scripting languages like R and Python. Human knowledge and analytical skill are required at every step of the pipeline, which is intended to produce unique, non-generalizable results. Although this pipeline is a good starting place as a statistical and analytical framework, it does not meet the requirements of building data products, especially when the data from which value is being derived is too big for humans to deal with on a single laptop. As data becomes bigger, faster, and more variable, tools for automatically deriving insights without human intervention become far more important.

1.2.1-Leveraging Large Datasets

Humans have an excellent ability to see large-scale patterns—the metaphorical forests and clearings though the trees. The cognitive process of making sense of data involves high-level overviews of data, zooming into specified levels of detail, and moving back out again. More data can be both tightly tuned patterns and signals just as much as it can be noise and distractions.

Statistical methodologies give us the means to deal with simultaneously noisy and meaningful data, either by describing the data through aggregations and indices or inferentially by directly modeling the data. These techniques help us understand data at the cost of computational granularity—for example, rare events that might be interesting signals tend to be smoothed out of our models. Statistical techniques that attempt to take into account rare events leverage a computer's power to track multiple data points simultaneously, but require more computing resources.

As our ability to collect data has grown, so has the need for wider generalization. Smart grids, quantified self, mobile technology, sensors, and connected homes require the application of personalized statistical inference. Scale comes not just from the amount of data, but from the number of facets that exploration requires —a forest view for individual trees.

What makes Hadoop different is partly the economics of data processing and partly the fact that Hadoop is a platform. However, what really makes Hadoop special is its timing—it was released right at the moment when technology needed a solution to do data analytics at scale, not just for population-level statistics, but also for individual generalizability and insight.

1.2.2-Hadoop for Data Products

Hadoop comes from big companies with big data challenges like Google, Facebook, and Yahoo; however, the reason Hadoop is important and the reason that you have picked up this book is because data challenges are no longer experienced only by the tech giants. Commercial and governmental entities from large to small: enterprises to startups, federal agencies to cities, and even individuals. Computing resources are also becoming ubiquitous and cheap—like the days of the PC when garage hackers innovated using available electronics, now small clusters of 10-20 nodes are being put together by startups to innovate in data exploration. Cloud computing resources such as Amazon EC2 and Google Compute Engine mean that data scientists have unprecedented on-demand, instant access to large-scale clusters for relatively little money and no data center management. Hadoop has made big data computing democratic and accessible, as illustrated by the following examples.

In 2011, Lady Gaga released her album *Born This Way,* an event that was broadcast by approximately 1.3 trillion social media impressions from "likes" to tweets to images and videos. Troy Carter, Lady Gaga's manager, immediately saw an opportunity to bring fans together, and in a massive data mining effort, managed to aggregate the millions of followers on Twitter and Facebook to a smaller, Lady Gaga-specific social network, LittleMonsters.com. The success of the site led to the foundation of Backplane (now Place), a tool for the generation and management of smaller, community- driven social networks.

The quantified self-movement has grown in popularity, and companies have been striving to make technological wearables, personal data collection, and even genetic sequencing widely available to consumers. Connected homes and mobile devices, along with other personal sensors, are generating huge amounts of individual data, which among other things sparks concern about privacy. In 2015, researchers in the United Kingdom created the *Hub of All Things (HAT*)—a personalized data collection that deals with the question "who owns your data?" and provides a technical solution to the aggregation of personal data.

Large-scale, individual data analytics have traditionally been the realm of social networks like Facebook and Twitter, but thanks to Place, large social networks are now the provenance of individual brands or artists. Cities deal with unique data challenges, but whereas the generalization of a typical city could suffice for many analytics, new data challenges are arising that must be explored on a per-city basis (what is the affect of industry, shipping, or weather on the performance of an acoustic sensor network?). How do technologies provide value to consumers utilizing their personal health records without aggregation to others because of privacy issues? Can we make personal data mining for medical diagnosis secure?

In order to answer these questions on a routine and meaningful (individual) basis, a data product is required. Applications like Place, ShotSpotter, quantified self products, and HAT derive their value from data and generate new data by providing an application platform and decision-making resources for people to act upon. The value they provide is clear, but traditional software development workflows are not up to the challenges of dealing with massive datasets that are generated from trillions of likes and millions of microphones, or the avalanche of personal data that we generate on a daily basis. Big data workflows and Hadoop have made these applications possible and personalized.

1.3-The Data Science Pipeline and the Hadoop Ecosystem

The data science pipeline is a pedagogical model for teaching the workflow required for thorough statistical analyses of data, as shown in Figure 1-1. In each phase, an analyst transforms an initial dataset, augmenting or ingesting it from a variety of data sources, wrangling it into a normal form that can be computed upon, either with descriptive or inferential statistical methods, before producing a result via visualization or reporting mechanisms. These analytical procedures are usually designed to answer specific questions, or to investigate the relationship of data to some business practice for validation or decision making.

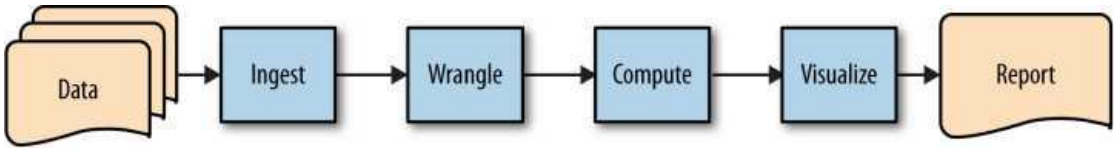

Figure 1-1. The data science pipeline

Although it may come as a surprise, original discussions about the application of data science revolved around the creation of meaningful information visualization, primarily because this workflow is intended to produce something that allows humans to make decisions. By aggregating, describing, and modeling large datasets, humans are better able to make judgments based on patterns rather than individual data points. Data visualizations are nascent data products—they generate their value from data, then allow humans to take action based on what they learn, creating new data from those actions.

However, this human-powered model is not a scalable solution in the face of exponential growth in the volume and velocity of data that many organizations are now grappling with. At even a small fraction of this scale, manual methods of data preparation and mining are simply unable to deliver meaningful insights in a timely manner.

In addition to the limitations of scale, the human-centric and one-way design of this workflow precludes the ability to efficiently design self-adapting systems that are able to learn. Machine learning algorithms have become widely available beyond academia, and fit the definition of data products very well. These types of algorithms derive their value from data as models are fit to existing datasets, then generate new data in return by making predictions about new observations.

To create a framework that allows the construction of scalable, automated solutions to interpret data and generate insights, we must revise the data science pipeline into a framework that incorporates a feedback loop for machine learning methods.

1.3.1-Big Data Workflows

With the goals of scalability and automation in mind, we can refactor the human- driven data science pipeline into an iterative model with four primary phases: *ingestion, staging, computation,* and *workflow management* (illustrated in Figure 1-2). Like the data science pipeline, this model in its simplest form takes raw data and converts it into insights. By converting the ingestion, staging, and computation steps into an automated workflow, this step ultimately produces a reusable data product as the output. The workflow management step also introduces a feedback flow mechanism, where the output from one job execution can be automatically fed in as the data input for the next iteration, and thus provides the necessary self-adapting framework for machine learning applications.

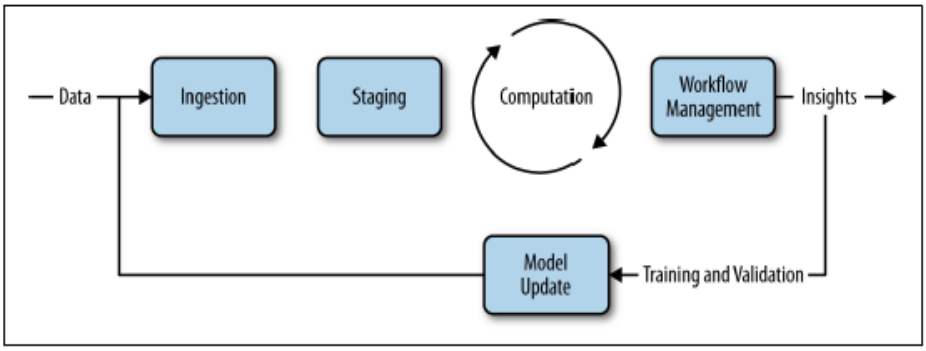

Figure 1-2. The big data pipeline

The ***ingestion*** phase is both the initialization of a model as well as an application interaction between users and the model. During initialization, users specify locations for data sources or annotate data (another form of ingestion). During interaction, users consume the predictions of the model and provide feedback that is used to reinforce the model.

The ***staging*** phase is where transformations are applied to data to make it consumable and stored so that it can be made available for processing. Staging is responsible for normalization and standardization of data, as well as data management in some computational data store.

The ***computation*** phase is the heavy-lifting phase with the primary responsibility of mining the data for insights, performing aggregations or reports, or building machine learning models for recommendations, clustering, or classification.

The ***workflow management*** phase performs abstraction, orchestration, and automation tasks that enable the workflow steps to be operationalized for production. The end result of this step should be an application, job, or script that can be run on- demand in an automated fashion.

Hadoop has specifically evolved into an ecosystem of tools that operationalize some part of this pipeline. For example, Sqoop and Kafka are designed for ingestion, allowing the import of relational databases into Hadoop or distributed message queues for on-demand processing. In Hadoop, data warehouses such as Hive and HBase provide data management opportunities at scale.

2-An Operating System for Big Data

2.1-Basic Concepts

In order to perform computation at scale, Hadoop distributes an analytical computation that involves a massive dataset to many machines that each simultaneously operate on their own individual chunk of data. More specifically, a distributed system must meet the following requirements:

- ***Fault tolerance:*** *If a component fails, it should not result in the failure of the entire system. The system should gracefully degrade into a lower performing state. If a failed component recovers, it should be able to rejoin the system.*
- ***Recoverability:*** *In the event of failure, no data should be lost.*
- ***Consistency:*** *The failure of one job or task should not affect the final result.*
- ***Scalability:*** *Adding load (more data, more computation) leads to a decline in performance, not failure; increasing resources should result in a proportional increase in capacity.*

Hadoop addresses these requirements through several abstract concepts, as defined in the following list:

- Data is distributed immediately when added to the cluster and stored on multiple nodes. Nodes prefer to process data that is stored locally in order to minimize traffic across the network.
- Data is stored in blocks of a fixed size (usually 128 MB) and each block is duplicated multiple times across the system to provide redundancy and data safety.
- A computation is usually referred to as a job; jobs are broken into tasks where each individual node performs the task on a single block of data.
- Jobs are written at a high level without concern for network programming, time, or low- level infrastructure, allowing developers to focus on the data and computation rather than distributed programming details.
- The amount of network traffic between nodes should be minimized transparently by the system. Each task should be independent and nodes should not have to communicate with each other during processing to ensure that there are no interprocess dependencies that could lead to deadlock.
- Jobs are fault tolerant usually through task redundancy, such that if a single node or task fails, the final computation is not incorrect or incomplete.
- Master programs allocate work to worker nodes such that many worker nodes can operate in parallel, each on their own portion of the larger dataset.

These basic concepts, while implemented slightly differently for various Hadoop systems, drive the core architecture and together ensure that the requirements for *fault tolerance, recoverability, consistency,* and *scalability* are met. Unlike data warehouses, however, Hadoop is able to run on more economical, commercial off-the-shelf hardware. As such, Hadoop has been leveraged primarily to store and compute upon large, heterogeneous datasets stored in "lakes" rather than warehouses, and relied upon for rapid analysis and prototyping of data products.

2.2-Hadoop Architecture

Hadoop is composed of two primary components that implement the basic concepts of distributed storage and computation:

- HDFS and YARN. HDFS is the Hadoop Distributed File System, responsible for managing data stored on disks across the cluster.
- YARN acts as a cluster resource manager, allocating computational assets (processing availability and memory on worker nodes) to applications that wish to perform a distributed computation.

The architectural stack is shown in Figure 2-1

HDFS and YARN work in concert to minimize the amount of network traffic in the cluster primarily by ensuring that data is local to the required computation. Duplication of both data and tasks ensures fault tolerance, recoverability, and consistency. Moreover, the cluster is centrally managed to provide scalability and to abstract low- level clustering programming details. Together, HDFS and YARN are a platform upon which big data applications are built; perhaps more than just a platform, they provide an operating system for big data.

Like any good operating system, HDFS and YARN are flexible. Alternatively, data storage systems can be built directly on top of HDFS to provide more features than a simple file system. For example, HBase is a columnar data store built on top of HDFS and is one the most advanced analytical applications that leverage distributed storage. However, YARN now allows richer abstractions of the cluster utility, making new data processing

applications for machine learning, graph analysis, SQL-like querying of data, or even streaming data services faster and more easily implemented.

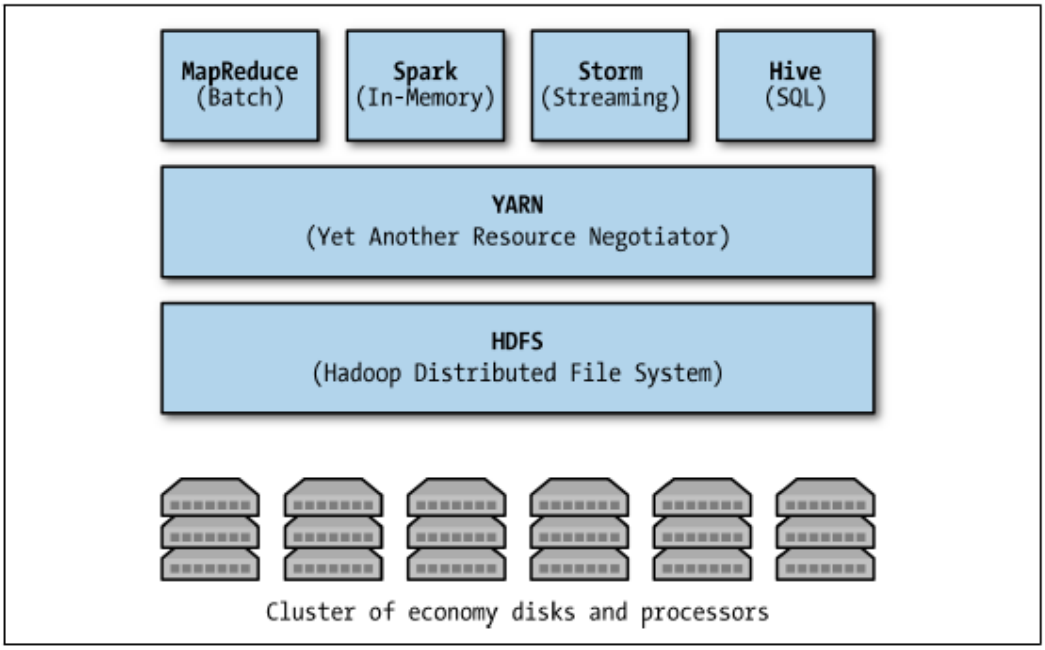

Figure 2-1. Hadoop is made up of HDFS and YARN

2.2.1-A Hadoop Cluster

Hadoop is actually the name of the software that runs on a cluster—namely, the distributed file system, HDFS, and the cluster resource manager, YARN, which are collectively composed of six types of background services running on a group of machines.

HDFS and YARN expose an application programming interface (API) that abstracts developers from low-level cluster administration details. A set of machines that is running HDFS and YARN is known as a cluster, and the individual machines are called nodes. A cluster can have a single node, or many thousands of nodes, but all clusters scale horizontally, meaning as you add more nodes, the cluster increases in both capacity and performance in a linear fashion.

YARN and HDFS are implemented by several daemon processes—that is, software that runs in the background and does not require user input. Hadoop processes are services, meaning they run all the time on a cluster node and accept input and deliver output through the network, similar to how an HTTP server works. Each of these processes runs inside of its own Java Virtual Machine (JVM) so each daemon has its own system resource allocation and is managed independently by the operating system. Each node in the cluster is identified by the type of process or processes that it runs:

- ***Master nodes:*** These nodes run coordinating services for Hadoop workers and are usually the entry points for user access to the cluster. Without masters, coordination would fall apart, and distributed storage or computation would not be possible.
- ***Worker nodes:*** These nodes are the majority of the computers in the cluster. Worker nodes run services that accept tasks from master nodes—either to store or retrieve data or to run a particular application.

Both HDFS and YARN have multiple master services responsible for coordinating worker services that run on each worker node.

For HDFS, the master and worker services are as follows:

- ***NameNode (Master):*** Stores the directory tree of the file system, file metadata, and the locations of each file in the cluster. Clients wanting to access HDFS must first locate the appropriate storage nodes by requesting information from the NameNode.
- ***Secondary NameNode (Master):*** Performs housekeeping tasks and checkpointing on behalf of the NameNode. Despite its name, it is not a backup NameNode.
- ***DataNode (Worker):*** Stores and manages HDFS blocks on the local disk. Reports health and status of individual data stores back to the NameNode.

At a high level, when data is accessed from HDFS, a client application must first make a request to the NameNode to locate the data on disk. The NameNode will reply with a list of DataNodes that store the data, and the client must then directly request each block of data from the DataNode. Note that the NameNode does not store data, nor does it pass data from DataNode to client, instead acting like a traffic cop, pointing clients to the correct DataNodes.

YARN has multiple master services and a worker service as follows:

- ***ResourceManager (Master):*** Allocates and monitors available cluster resources (e.g., physical assets like memory and processor cores) to applications as well as handling scheduling of jobs on the cluster.
- ***ApplicationMaster (Master):*** Coordinates a particular application being run on the cluster as scheduled by the ResourceManager.
- ***NodeManager (Worker):*** Runs and manages processing tasks on an individual node as well as reports the health and status of tasks as they're running.

Similar to how HDFS works, clients that wish to execute a job must first request resources from the ResourceManager, which assigns an application-specific ApplicationMaster for the duration of the job. The ApplicationMaster tracks the execution of the job, while the ResourceManager tracks the status of the nodes, and each individual NodeManager creates containers and executes tasks within them.

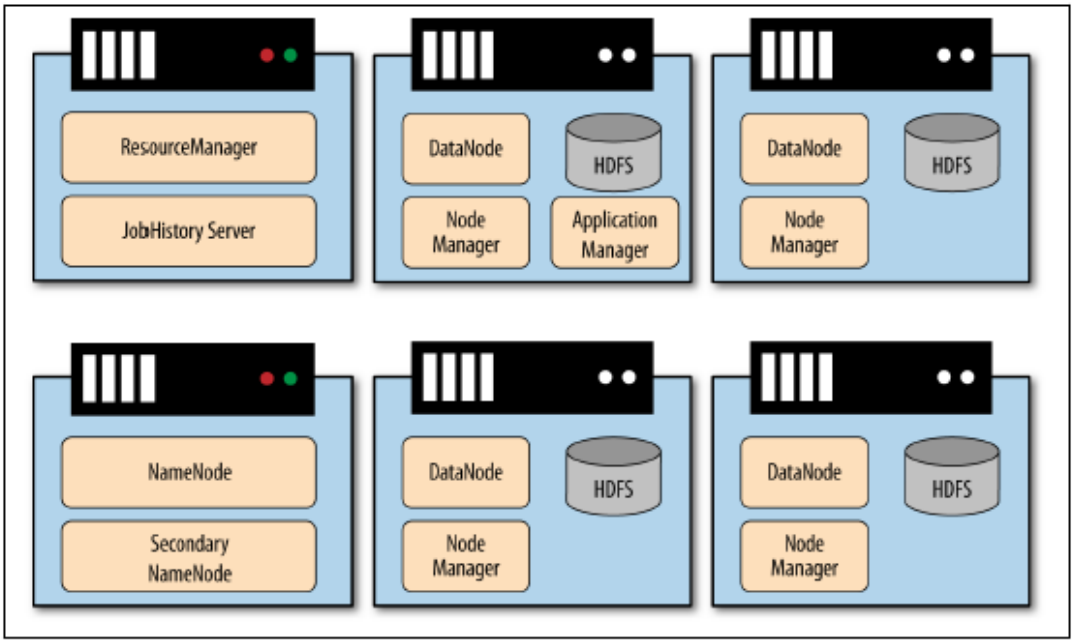

Figure 2-2. A small Hadoop cluster with two master nodes and four workers nodes that implements all six primary Hadoop services

Master processes are so important that they usually are run on their own node so they don't compete for resources and present a bottleneck. However, in smaller clusters, the master daemons may all run on a single node. An example deployment of a small Hadoop cluster with six nodes, two master and four worker, is shown in Figure 2-2. Note that in larger clusters the NameNode and the Secondary NameNode will reside on separate machines so they do not compete for resources. The size of the cluster should be relative to the size of the expected computation or data storage because clusters scale horizontally. Typically a cluster of 20-30 worker nodes and a single master is sufficient to run several jobs simultaneously on datasets in the tens of terabytes. For more significant deployments of hundreds of nodes, each master requires its own machine; and in even larger clusters of thousands of nodes, multiple masters are utilized for coordination.

2.2.2-HDFS

HDFS provides redundant storage for big data by storing that data across a cluster of cheap, unreliable computers, thus extending the amount of available storage capacity that a single machine alone might have. However, because of the networked nature of a distributed file system, HDFS is more complex than traditional file systems. In order to minimize that complexity, HDFS is based off of the centralized storage architecture.

In principle, HDFS is a software layer on top of a native file system such as ext4 or xfs, and in fact Hadoop generalizes the storage layer and can interact with local file systems and other storage types like Amazon S3. HDFS is designed for storing very large files with streaming data access, and as such, it comes with a few caveats:

- HDFS performs best with a modest number of very large files—for example, millions of large files (100 MB or more) rather than billions of smaller files that might occupy the same volume.
- HDFS implements the WORM pattern—write once, read many. No random writes or appends to files are allowed.
- HDFS is optimized for large, streaming reading of files, not random reading or selection.

Therefore, HDFS is best suited for storing raw input data to computation, intermediary results between computational stages, and final results for the entire job. It is not a good fit as a data backend for applications that require updates in real-time, interactive data analysis, or record-based transactional support. Instead, by writing data only once and reading many times, HDFS users tend to create large stores of heterogeneous data to aid in a variety of different computations and analytics. These stores are sometimes called "data lakes".

Blocks

HDFS files are split into blocks, usually of either 64 MB or 128 MB. The block size is the minimum amount of data that can be read or written to in HDFS, similar to the block size on a single disk file system. To achieve the best performance, Hadoop prefers big files that are broken up into smaller chunks, if only through the combination of many smaller files into a bigger file format. However, if many small files are stored on HDFS, it will not reduce the total available disk space by 128 MB per file.

Blocks allow very large files to be split across and distributed to many machines at run time. Different blocks from the same file will be stored on different machines to provide for more efficient distributed processing. In fact, there is a one-to-one connection between a task and a block of data.

Additionally, blocks will be replicated across the DataNodes. By default, the replication is three-fold, but this is also configurable at runtime. Therefore, each block exists on three different machines and three different disks, and if even two nodes fail, the data will not be lost.

Data management

The master NameNode keeps track of what blocks make up a file and where those blocks are located. The NameNode communicates with the DataNodes, the processes that actually hold the blocks in the cluster. Metadata associated with each file is stored in the memory of the NameNode master for quick lookups, and if the NameNode stops or fails, the entire cluster will become inaccessible!

The Secondary NameNode is not a backup to the NameNode, but instead performs housekeeping tasks on behalf of the NameNode, including (and especially) periodically merging a snapshot of the current data space with the edit log to ensure that the edit log doesn't get too large. The edit log is used to ensure data consistency and prevent data loss; if the NameNode fails, this merged record can be used to reconstruct the state of the DataNodes.

When a client application wants access to read a file, it first requests the metadata from the NameNode to locate the blocks that make up the file, as well as the locations of the DataNodes that store the blocks. The application then communicates directly with the DataNodes to read the data. Therefore, the NameNode simply acts like a journal or a lookup table and is not a bottleneck to simultaneous reads.

2.2.3-YARN (Yet another Resource Negotiator)

While the original version of Hadoop (Hadoop 1) popularized MapReduce and made large-scale distributed processing accessible to the masses, it only offered MapReduce on HDFS.

MapReduce can be very efficient for large-scale batch workloads, but it's also quite I/O intensive, and due to the batch-oriented nature of HDFS and MapReduce, faces significant limitations in support for interactive analysis, graph processing, machine learning, and other memory-intensive algorithms. While other distributed processing engines have been developed for these particular use cases, the MapReduce-specific nature of Hadoop 1 made it impossible to repurpose the same cluster for these other distributed workloads.

Hadoop 2 addresses these limitations by introducing YARN, which decouples workload management from resource management so that multiple applications can share a centralized, common resource management service. By providing generalized job and resource management capabilities in YARN, Hadoop is no longer a singularly focused MapReduce framework but a full-fledged multi-application, big data operating system.

YARN provides better resource management in Hadoop, resulting in improved cluster efficiency and application performance. This feature not only improves the MapReduce Data Processing but also enables Hadoop usage in other data processing applications.

2.3-Working with a Distributed File System

When working with HDFS, the file system is in fact a distributed, remote file system. All requests for file system lookups are sent to the Name- Node, which responds very quickly with lookup-type requests. Once you start accessing files, things can slow down quickly, as the various blocks that make up the requested file must be transferred over the network to the client. Because blocks are replicated on HDFS, you'll actually have less disk space available in HDFS than is available from the hardware.

For the most part, interaction with HDFS is performed through a command-line interface. Additionally, there is an HTTP interface to HDFS, as well as a programmatic interface written in Java.

It is assumed that these commands are performed on a client that can connect to a remote Hadoop cluster, or which is running a pseudo-distributed cluster on the localhost. It is also assumed that the hadoop command and other utilities from *$HADOOP_HOME/bin* are on the system path and can be found by the operating system.

2.3.1-Basic File System Operations

All of the usual file system operations are available to the user, such as creating directories; moving, removing, and copying files; listing directories; and modifying permissions of files on the cluster. To see the available commands in the fs shell, type:

hostname $ **hadoop fs -help**

Usage: hadoop fs [generic options]

….

As you can see, many of the familiar commands for interacting with the file system are there, specified as arguments to the hadoop **fs** command as flag arguments in the Java style. Be aware that order can matter when specifying such options.

To get started, let's copy some data from the local file system to the remote (distributed) file system. To do this, use either the **put** or **copyFromLocal** commands. The moveFromLocal command is similar, but the local copy is deleted after a successful transfer to the distributed file system.

For example to copy *shakespeare.txt* file from your local working directory to the distributed file system as follows:

hostname $ **hadoop fs -copyFromLocal shakespeare.txt shakespeare.txt**

This example invokes the Hadoop shell command copyFromLocal with two arguments, <src> and <dst>, both of which are specified as relative paths to a file called *shakespeare.txt.* The command searches your current working directory for the *shakespeare.txt* file and copies it to the */user/ analyst/shakespeare.txt* path on HDFS by first requesting information about that path from the NameNode, then directly communicating with the DataNodes to transfer the file. Because Shakespeare's complete works are less than 64 MB, it is not broken up into blocks. Both relative and absolute paths must be taken into account. The preceding command is shorthand for:

hostname $ **hadoop fs -put /home/analyst/shakespeare.txt \ hdfs://localhost/user/analyst/shakespeare.txt**

Home directory on HDFS is */user/analyst/*. In order to better manage the HDFS file system, create a hierarchical tree of directories just as you would on your local file system:

hostname $ **hadoop fs -mkdir corpora**

To list the contents of the remote home directory, use the ls command:

hostname $ hadoop fs -ls
drwxr-xr-x - analyst analyst 0 2015-05-04 17:58 corpora
-rw-r--r-- 3 analyst analyst 8877968 2015-05-04 17:52 shakespeare.txt

The first column shows the permissions mode of the file. The second column is the replication of the file; by default, the replication is 3. Note that directories are not replicated, so this column is a dash (-) in that case. The user and group follow, then the size of the file in bytes (zero for directories). The last modified date and time is up next, with the name of the file appearing last.

Other basic file operations like mv, cp, and rm will all work as expected on the remote file system. Use rm -R to recursively remove a directory with all files in it.

To read the contents of a file, use the cat command, then pipe the output to less in order view the contents of the remote file:

hostname $ **hadoop fs -cat shakespeare.txt | less**

Alternatively, you can use the tail command to inspect only the last kilobyte of the file:

hostname $ **hadoop fs -tail shakespeare.txt | less**

It is efficient to **hadoop fs -cat** the file and pipe it to the local shell's head, as the head command terminates the remote stream before the entire file is read. The, **hadoop fs -tail** command seeks to the correct position in the remote file and returns only the required data over the network.

To transfer entire files from the distributed file system to the local file system, use **get** or **copyToLocal**, which are identical commands. Similarly, use the **moveToLocal** command, which also removes the file from the distributed file system. Finally, the **get merge** command merges all files that match a given pattern or directory are copied and merged into a single file on the localhost. If files on the remote system are large, you may want to pipe them to a compression utility:

hostname $ **hadoop fs -get shakespeare.txt ./shakespeare.from-remote.txt**

Table 2-1 demonstrates other useful commands that are provided by hadoop fs.

Table 2-1. Other useful utilities

Command	Output
`hadoop fs -help <cmd>`	Provides information and flags specifically about the *<cmd>* in question.
`hadoop fs -test <path>`	Answer various questions about *<path>* (e.g., exists, is directory, is file, etc.)
`hadoop fs -count <path>`	Count the number of directories, files, and bytes under the paths that match the specified file pattern.
`hadoop fs -du -h <path>`	Show the amount of space, in bytes, used by the files that match the specified file pattern.
`hadoop fs -stat <path>`	Print statistics about the file/directory at *<path>*.
`hadoop fs -text <path>`	Takes a source file and outputs the file in text format. Currently Zip, TextRecordInputStream, and Avro sources are supported.

2.3.2-File Permissions in HDFS

There are three types of permissions: read (r), write (w), and execute (x). These permissions define the access levels for the owner, the group, and any other system users. For directories, the execute permission allows access to the contents of the directory; however, execute permissions are ignored on HDFS for files. Read and write permissions in the context of HDFS specify who can access the data and who can append to the file.

Permissions are expressed during the directory listing command Is. Each mode has 10 slots. The first slot is a d for directories, otherwise a - for files. Each of the following groups of three indicates the rwx permissions for the owner, group, and other users, respectively. There are several HDFS shell commands that will allow you to manage the permissions of files and directories, namely the familiar chmod, chgrp, and chown commands:

hostname $ **hadoop fs -chmod 664 shakespeare.txt**

This command changes the permissions of *shakespeare.txt* to -rw-rw-r--. The 664 is an octal representation of the flags to set for the permission triple. Consider 6 in binary, 110—this means set the read and write flags but not

the execute flag. Completely permissible is 7, 111 in binary and read-only is 4, 100 in binary. The chgrp and chown commands change the group and owner of the files on the distributed file system.

The remote clients can create arbitrary users on the system. These permissions, therefore, should only be used to prevent accidental data loss and to share file system resources between known users, not as a security mechanism.

2.3.3-Other HDFS Interfaces

Programmatic access to HDFS is made available to software developers through a Java API, and any serious data ingestion into a Hadoop cluster should consider utilizing that API. There are also other tools for integrating HDFS with other file systems or network protocols—for example, FTP or Amazon S3.

There are also HTTP interfaces to HDFS, which can be used for routine administration of the cluster file system and programmatic access to HDFS with Python. HTTP access to HDFS comes in two primary interfaces: direct access through the HDFS daemons that serve HTTP requests, or via proxy servers that expose an HTTP interface then directly access HDFS on the client's behalf using the Java API.

The NameNode also supplies direct, read-only access to HDFS over HTTP through an HTTP server that runs on port 50070. If running in pseudo-distributed mode, simply open a browser and navigate to http://127.0.0.1:50070; otherwise, use the host name of the NameNode on your cluster. The NameNode also allows users to browse the file system using a search and navigation utility that is found under the Utilities drop-down tab

DataNodes themselves can be directly browsed for information, accessing the DataNode host on port 50075; all active DataNodes are listed on the NameNode HTTP site.

By default, the direct HTTP interface is read-only. In order to provide write access to an HDFS cluster, a proxy such as WebHDFS must be used. WebHDFS secures the cluster via authentication with Kerberos.

2.4-Working with Distributed Computation

2.4.1-MapReduce: A Functional Programming Model

When people refer to MapReduce, they're usually referring to the distributed programming model. MapReduce is a simple but very powerful computational framework specifically designed to enable fault-tolerant distributed computation across a cluster of centrally managed machines.

Functional programming is a style of programming that ensures unit computations are evaluated in a stateless manner. Data is transferred between functions by sending the output of one function as the input to another, wholly independent function. These traits make functional programming a great fit for distributed, big data computational systems, because it allows us to move the computation to any node that has the data input and guarantee that we will still get the same result. Because functions are stateless and depend solely on their input, many functions on many machines can work independently on smaller chunks of the dataset. By strategically chaining the outputs of functions to the inputs of other functions, we can guarantee that we will reach a final computation across the entire dataset.

It shouldn't be a surprise that the two functions that distribute work and aggregate results are called map and reduce, respectively. Furthermore, the data that is operated upon as input and output in these functions are not simple lists or collections of values; instead, MapReduce utilizes "key/value" pairs to coordinate computation.

Pseudocode for map and reduce functions in Python would therefore look as follows:

```
def map(key, value):
    # Perform processing
    return (intermed_key, intermed_value)

def reduce(intermed_key, values):
    # Perform processing
    return (key, output)
```

A map function takes as input a series of key/value pairs and operates singly on each individual pair. In the preceding pseudocode, we've expressed this as it is represented in the MapReduce Java API: a function that takes two arguments, a key and a value. After performing some analysis or transformation on the input data, the map function may then output zero or more resulting key/value pairs, represented as a single tuple in the preceding pseudocode. This is generally described as shown in Figure 2-3, where a map function is applied to an input list to create a new output list.

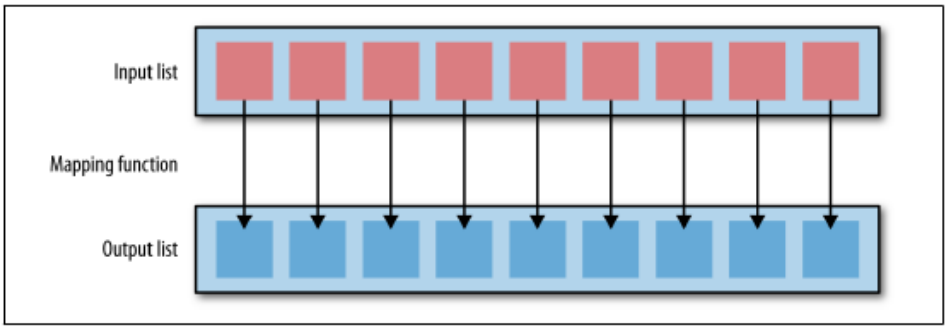

Figure 2-3. A map function takes as input a list of key/value pairs and operates singly upon each individual element in the list, outputting zero or more key/value pairs

Typically, the map operation is where the core analysis or processing takes place, as this is the function that sees each individual element in the dataset. Consider how filters are implemented in a map context: each key/value pair is tested to determine whether it belongs in the final dataset, and is emitted if it does or ignored if not. After the map phase, any emitted key/value pairs will then be grouped by key and those key/value groups are applied as input to reduce functions on a per-key basis. As shown in Figure 2-4, a reduce function is applied to an input list to output a single, aggregated value.

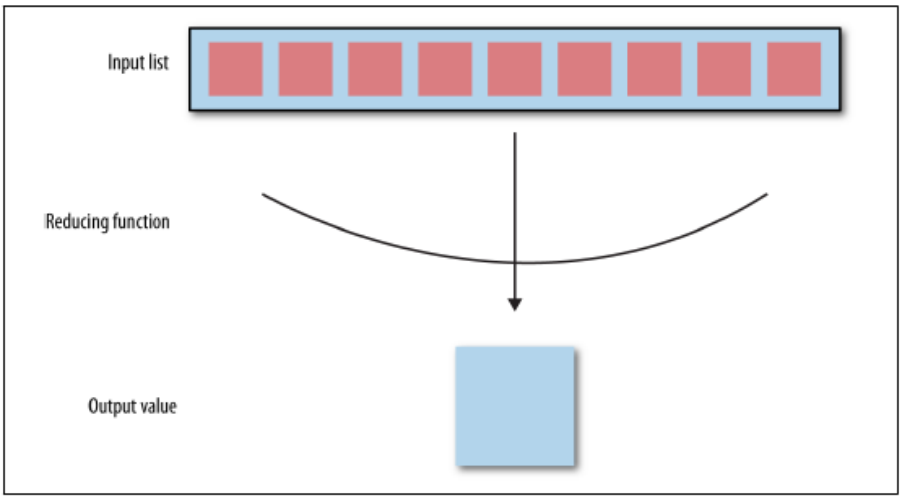

Figure 2-4. A reducer takes a key and a list of values as input, operates on the values list as a whole, usually through aggregation operations, and outputs zero or more key/value pairs

2.4.2-MapReduce: Implemented on a Cluster

Because mappers apply the same function to each element of any arbitrary list of items, they are well suited to distribution across nodes on a cluster. Each node gets a copy of the mapper operation, and applies the mapper to the key/value pairs that are stored in the blocks of data of the local HDFS data nodes. There can be any number of mappers working independently on as much data as possible, really only limited by the number of processors available on the cluster

Reducers require as input the output of the mappers on a per-key basis; therefore, reducer computation can also be distributed such that there can be as many reduce operations as there are keys available from the mapper output. We should correctly expect that each reducer sees *all* values for a single, unique key. In order to meet this requirement, a shuffle and sort operation is required to coordinate the map and reduce phases. Therefore, in broad strokes, the phases of MapReduce are shown in Figure 2-5.

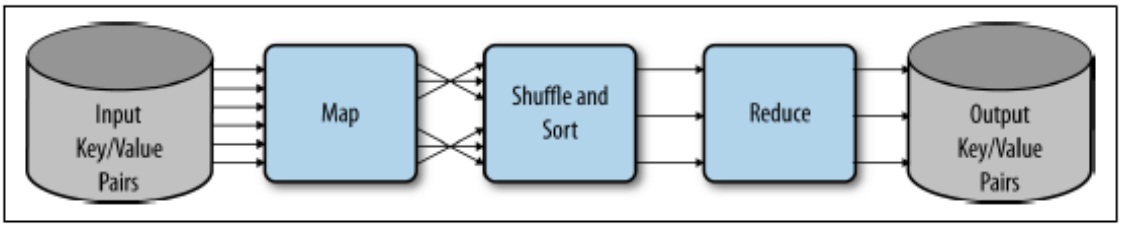

Figure 2-5. Broadly, MapReduce is implemented as a staged framework where a map phase is coordinated to a reduce phase via an intermediate shuffle and sort

The phases shown in Figure 2-5 are as follows:

Phase 1: *Local data is loaded into a mapping process as key/value pairs from HDFS.*
Phase 2: *Mappers output zero or more key/value pairs, mapping computed values to a particular key.*
Phase 3: *These pairs are then sorted and shuffled based on the key and are then passed to a reducer such that all values for a key are available to it.*
Phase 4: *Reducers then must output zero or more final key/value pairs, which are the output (reducing the results of the map).*

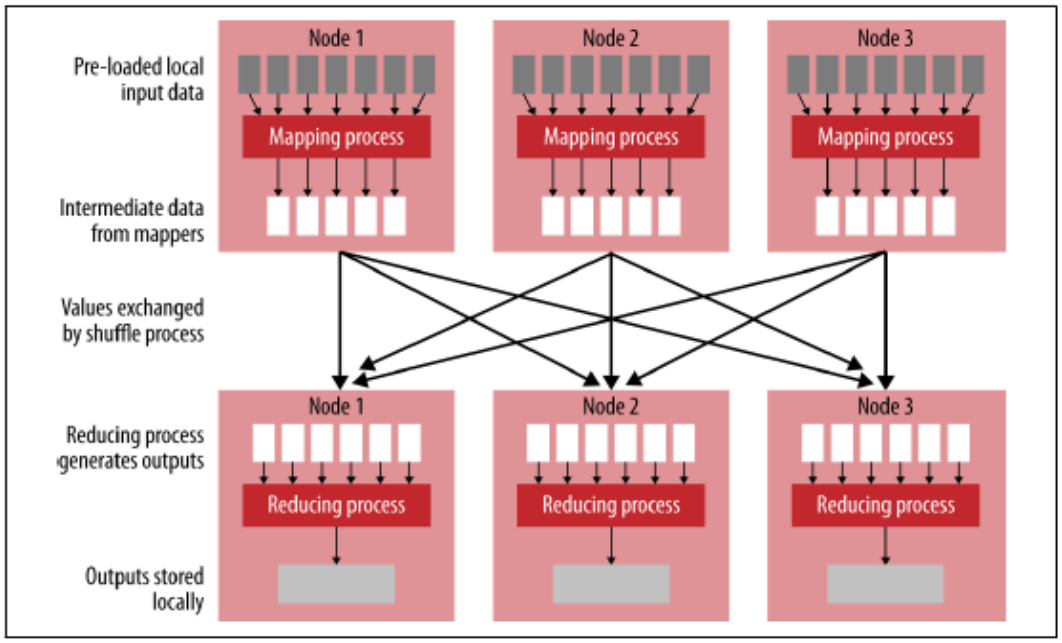

Figure 2-6. Data flow of a MapReduce job being executed on a cluster of a few nodes

There are a few more details that are required when executing MapReduce on a cluster. For example, consider how the key/value pairs are defined, and what is required in order to do correct partitioning of the keyspace. Enhancements and optimizations. The details of data flow in a MapReduce pipeline executed on a cluster of a few nodes are outlined in Figure 2-6.

In a cluster execution context, a map task is assigned to one or more nodes in the cluster, which contains local blocks of data specified as input to the map operation. Blocks are stored in HDFS and and are split into smaller chunks by an InputFormat class, which defines how data is presented to the map applications. For example, given text data, the key might be the file identifier and line number and the value might be the string of the line content. RecordReader presents each individual key/ value pair to the map operation supplied by the user, which then outputs one or more intermediate key/value pairs. A common optimization at this point is to apply a combiner—a process that aggregates map output for a single mapper, similar to how a reducer works, but without full knowledge of the keyspace. This prework leads to less work for the reducers and therefore better reducer performance.

The intermediate keys are pulled from the map processes to a partitioner. The partitioner decides how to allocate the keys to the reducers. The partitioner also sorts the key/value pairs such that the full "shuffle and sort" phase is implemented. Finally, the reducers start work, pulling an iterator of data for each key and performing a reduce operation such as an aggregation. Their output key/value pairs are then written back to HDFS using an Output Format class.

There are many other tools associated with the management of large-scale jobs inside of a MapReduce cluster execution context as well. To name a few, Counter and Reporter objects are used for job tracking and evaluation and caches are used to supply ancillary data during processing.

2.4.3-Beyond a Map and Reduce: Job Chaining

Many algorithms or data processing tasks can easily be implemented in MapReduce with a simple shift in normal problem-solving workflows to account for the stateless operation and interaction of the map and reduce functions. However, more complex algorithms and analyses cannot be distilled to a single MapReduce job. For example, many machine learning or predictive analysis techniques require optimization, an iterative process where error is minimized. MapReduce does not support native iteration through a single map or reduce.

In MapReduce, a job actually refers to the full application (program), and therefore the complete execution of map and reduce functions across all the input data. Jobs for complex analyses are generally comprised of many internal tasks, where a task is the execution of a single map or reduce operation on a block of data. Because there are many workers simultaneously performing similar tasks, some data processing workflows can take advantage of that fact and run "map-only" or "reduce-only" jobs. For example, a binning methodology can take advantage of the built-in partitioner to group similar data together. Binned data can then be used downstream in other MapReduce jobs to perform frequency analysis or compute probability distributions.

In fact, the use of multiple MapReduce jobs to perform a single computation is how more complex applications are constructed, through a process called "job chaining." By creating data flows through a system of intermediate MapReduce jobs, as shown in Figure 2-8, we can create a pipeline of analytical steps that lead us to our end result.

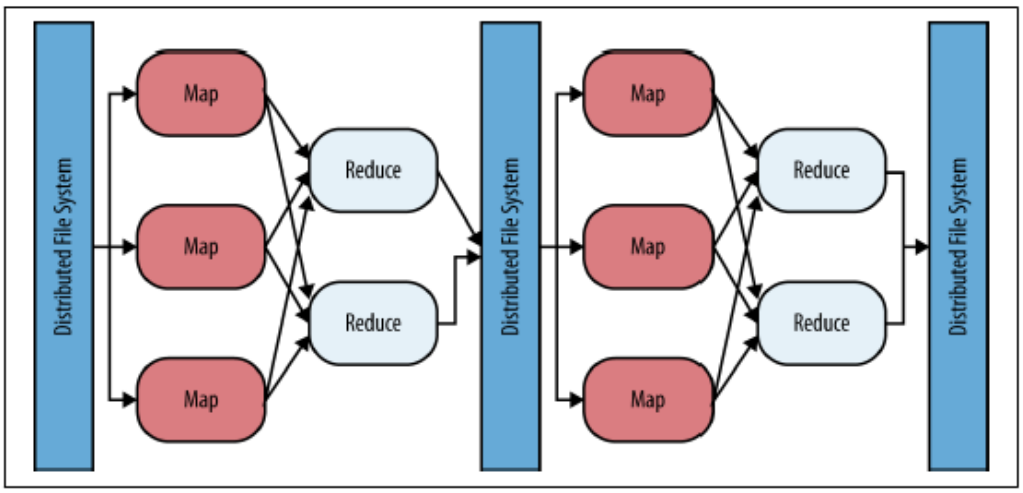

Figure 2-8. Complex algorithms or applications are actually made up through the chaining of MapReduce jobs where the input of a downstream MapReduce job is the output of a more recent one

3-A Framework for Python and Hadoop Streaming

The current version of Hadoop MapReduce is a software framework for composing jobs that process large amounts of data in parallel on a cluster, and is the native distributed processing framework that ships with Hadoop. The framework exposes a Java API that allows developers to specify input and output locations on HDFS, map and reduce functions, and other job parameters as a *job configuration.* Jobs are compiled and packaged into a JAR, which is submitted to the ResourceManager by the *job client*—usually via the command line. The ResourceManager then schedules tasks, monitors them, and provides status back to the client.

Typically, a MapReduce application is composed of three Java classes: a Job, a Mapper, and a Reducer. Mappers and reducers handle the details of computation on key/value pairs and are connected through a shuffle and sort phase. The Job configures the input and output data format by specifying the InputFormat and OutputFormat classes of data being serialized to and from HDFS. All of these classes must extend abstract base classes or implement MapReduce interfaces.

However, Java is not the only option to use the MapReduce framework! For example, C++ developers can use *Hadoop Pipes,* which provides an API for using both HDFS and MapReduce. But what is of most interest to data scientists is *Hadoop Streaming,* a utility written in Java that allows the specification of any executable as the mapper and reducer. With Hadoop Streaming shell utilities, R, or Python, scripts can all be used to compose MapReduce jobs. This allows data scientists to easily integrate MapReduce into their workflows—particularly for routine data management tasks that don't require extensive software development.

3.1-Hadoop Streaming

Hadoop Streaming is a utility, packaged as a JAR file that comes with the Hadoop MapReduce distribution. Streaming is used as a normal Hadoop job passed to the cluster via the job client, but allows you to also specify arguments such as the input and output HDFS paths, along with the mapper and reducer executable. The job is then run as a normal MapReduce job, managed and monitored by the ResourceMan ager and the MRAppMaster as usual until the job completes.

Input to both mappers and reducers is read from stdin, which a Python process can access via the sys module. Hadoop expects the Python mappers and reducers to write their output key/value pairs to stdout. Figure 3-1

demonstrates this process in a MapReduce context.

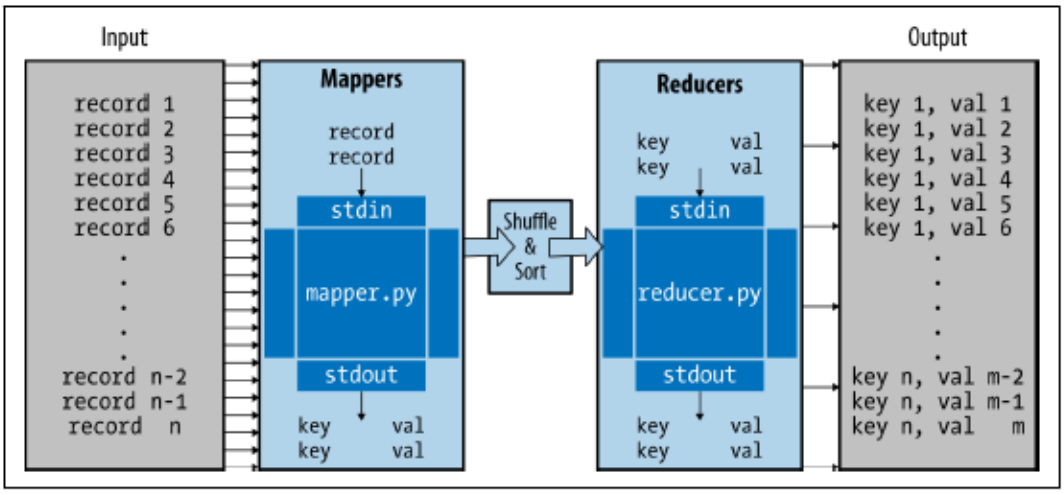

Figure 3-1. Data flow in Hadoop Streaming via Python mapper.py and reducer.py scripts

When Streaming executes a job, each mapper task will launch the supplied executable inside of its own process. The mapper then converts the input data into lines of text and pipes it to the stdin of the external process while simultaneously collecting output from stdout. The input conversion is usually a straightforward serialization of the value because data is being read from HDFS, where each line is a new value. The mapper expects output to be in a string key/value format, where the key is separated from the value by some separator character, tab (\t) by default. If there is no separator, then the mapper considers the output to only be a key with a null value. The separator can be customized by passing arguments to the Hadoop Streaming job.

The reducer is also launched as its own executable after the output from the mappers is shuffled and sorted to ensure that each key is sent to the same reducer. The key/ value output strings from the mapper are streamed to the reducer as input via stdin, matching the data output from the mapper, and guaranteed to be grouped by key. The output the reducer emits to stdout is expected to have the same key, separator, and value format as the mapper.

Therefore, in order to write Hadoop jobs using Python, we need to create two Python files, *mapper.py* and a *reducer.py*. We will implement the WordCount example.

First, we create our executable mapper in a file called *mapper.py:*

```
#!/usr/bin/env python

import sys

if __name__== "__main__":
    for tine in sys.stdin:
        for word in tine.sptit():
            sys.stdout.write("{}\t1\n".format(word))
```

The mapper simply reads each line from sys.stdin, splits on space, then writes each word and a 1 separated by a tab, line-by-line to sys.stdout. The reducer is a bit more complex because we have to track which key we're on at every line of input, and only emit a completed sum when we see a new key. In a file called *reducer.py*, we implement the reducer executable as follows:

```
#!/usr/bin/env python

import sys

if __name__== '__main__':
    curkey = None total = 0
    for line in sys.stdin:
        key, val = line.split("\t")
        val = int(val)

        if key == curkey: totat += vat else:
            if curkey is not None:
                sys.stdout.write("{}\t{}\n".format(curkey, total))

            curkey = key
            totat = vat
```

As the reducer iterates over each line in the input from stdin, it splits the line on the separator character and converts the value to an integer. It then performs a check to ensure that we're still computing the count for the same key; otherwise, it writes the output to stdout and restarts the count for the new key.

3.1.1-Computing on CSV Data with Streaming

String-typed input and output from our mappers and reducers means that we should carefully consider the datatypes that we're putting into the system, and how much parsing work we want our Python script to do. For example, we could use the built-in ast.literal_eval to parse simple data types (e.g., numbers, tuples, lists, dicts, or booleans); or we could input and output complex data structures with a structured serialization (e.g., JSON or even XML).

In this example, we'll consider a dataset of the on-time performance of domestic flights in the United States. We have CSV data that is as follows, where each row contains the flight date; the airline ID; a flight number; the origin and destination airport; the departure time and delay in minutes; the arrival time and delay in minutes; and finally, the amount of time in the air as well as the distance in miles:

```
2014-04-01,19805,1,JFK,LAX,0854,-6.00,1217,2.00,355.00,2475.00
2014-04-01,19805,2,LAX,JFK,0944,14.00,1736,-29.00,269.00,2475.00
```

We'll start our example of writing structured MapReduce Python code by computing the average departure delay for each airport. First, let's take a look at the mapper. Write the following code into a file called *mapper.py:*

```
#!/usr/bin/env python

import sys
import csv

SEP = "\t" class Mapper(object):

    def   __init__ (self, stream, sep=SEP):
        self.stream = stream
         self.sep = sep
```

```
    def emit(self, key, value):
        sys.stdout.write("{}{}{}\n".format(key, self.sep, value))

    def map(self):
        for row in self:
            self.emit(row[3], row[6])

    def  __iter__(self):
        reader = csv.reader(self.stream)
        for row in reader:
            yield row
if __name__ == '__main__':
    mapper = Mapper(sys.stdin)
    mapper.map()
```

The Mapper class for our average flight delay example takes two arguments on instantiation, both of which have defaults, namely the infile and the separator. The infile refers to the location that data will be received from, which by default is stdin. Hadoop computes with key/value pairs, so the second argument is used to determine what part of the input/output strings are the key and which are the values. By default, the separator is the tab character (\t).

The next built-in __iter__ method allows the class to be used as an iterable, and this function should return another iterable, or if it simply returns self. This class can now be used in for statements and by parsing each line of stdin using a csv.reader and yielding each row.

Now let's take a look at the reducer called *reducer.py:*

```
#!/usr/bin/env python

 import sys

from itertools import groupby
from operator import itemgetter

SEP = "\t"

class Reducer(object):

    def   __init__  (self, stream, sep=SEP):
        self.stream = stream
        self.sep = sep

    def emit(self, key, value):
        sys.stdout.write("{}{}{}\n".format(key, self.sep, value))

    def reduce(self):
            for current, group in groupby(self, itemgetter( )):
            total = 0
            count = 0

            for item in group:
```

```
                total += item[1]
                count +=1

        self.emit(current, float(total) / float(count))

    def __iter__ (self):
        for line in self.stream:
            try:
                parts = line.split(self.sep)
                yield parts[0], float(parts[1])
            except:
                continue


if__name__== '__main__' :
    reducer = Reducer(sys.stdin) reducer.reduce()
```

A memory-safe iterator helper groupby and an operator itemgetter. In the reduce function, as in the mapper, we loop through the entire dataset using an iterable that splits the key and value apart.

Because the data is coming to the reducer sorted alphabetically by token due to the shuffle and sort phase of the Hadoop pipeline, we want to automatically group the keys and their values together. The memory safety comes from the fact that groupby returns an iterator rather than a list that is held in memory, and only reads one line at a time. The itemgetter operator simply specifies by which value of each tuple being yielded that should be grouped on—in this case, the first element of the tuple.

After memory-efficiently grouping our values together, we simply sum the delays, divide by the number of flights, and emit the airport as the key and the mean as the value as output.

3.1.2-Executing Streaming Jobs

First test your code, make sure your *mapper.py* and *reducer.py* are executable. Simply use the chmod command in your terminal as follows:

hostname $ **chmod +x mapper.py**
hostname $ **chmod +x reducer.py**

To test your mapper and reducer using a CSV file as input, use the cat command to output the contents of the file, piping the output from stdout to the stdin of the *mapper.py,* which pipes to sort and then to *reducer.py* and finally prints the result to the screen. To test the average delay per airport mapper and reducer, execute the following in a terminal where *mapper.py, reducer.py,* and *flights.csv* are all in your current working directory:

hostname $ **cat flights.csv | ./mapper.py | sort | ./reducer.py**

In order to deploy the code to the cluster, we need to submit the Hadoop Streaming JAR to the job client, passing in our custom operators as arguments. The location of the Hadoop Streaming job depends on how you've set up and configured Hadoop. For now, we'll assume that you have an environment variable, $HADOOP_HOME, that specifies the location of the install and that $HADOOP_HOME/bin is in your $PATH. If so, execute the Streaming job against the cluster as follows:

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input flights.csv \
    -output average_delay \
    -mapper mapper.py \
    -reducer reducer.py \
    -file mapper.py \
    -file reducer.py
```

Note the use of the -file option, which causes the Streaming job to send the scripts across the cluster. Executing this command will cause the job to be started on the Hadoop cluster. The *mapper.py* and *reducer.py* scripts will be sent to each node in the cluster before processing and will be used in each phase of the pipeline.

If there are additional files that should be sent along with the job—for example, a lookup table for the airline IDs—they can also be packaged with the job using the -file option.

Hadoop Streaming has many other settings, allowing users to specify classes in the Hadoop library for partitioners, input and output formats, and so on.

3.2-A Framework for MapReduce with Python

Slightly more advanced usage of Hadoop Streaming takes advantage of standard error (stderr) to update the Hadoop status as well as Hadoop counters. This technique essentially allows Streaming jobs to access the Reporter object, a part of the MapReduce Java API that tracks the global status of a job. By writing specially formatted strings to stderr, both mappers and reducers can update the global job status to report their progress and indicate they're alive. For jobs that take a significant amount of time, this is critical to ensuring that the framework doesn't assume a task has timed out.

Hadoop natively implements a number of counters, counting the number of records and bytes processed, but custom counters are an easy way to track metrics within a job, or to provide an associated channel for side computation.

For example, we could implement counters in our simple WordCount program to keep track of the global word count as well as to count our *vocabulary*—that is, the number of unique words.

They can also be used to compute mean squared error or classification metrics when evaluating machine learning models across the dataset. To use the Counter and Status features of the Reporter, augment the Mapper and Reducer classes from the last section with the following methods:

```
def status(setf, message):
    sys.stderr.write("reporter:status:{}\n".format(message))

def counter(setf, counter, amount=1, group="AppticationCounter"):
    sys.stderr.write("reporter:counter:{},{},{}\n".format(group, counter, amount)
)
```

The counter method allows both the map and reduce functions to update the count of any named counter by any amount necessary (defaults to incrementing by one).

In order to extend our average flight delay application to provide a count of early and delayed flights and to send status updates on start and finish, update the map function as

```
Def map( self):
    setf.status("mapping started")
     def map( self):
        for row in self:
            if row[6]<0 :
                self.counter("early departure")
            else:
                self.counter("late departure")

             self.emit(row[3], row[6])

          self.status("mapping complete")
```

This simple addition gives us greater insight into what is happening with average delays without a lengthy Hadoop job simply to count early and late flights. In the reducer, we may want to compute the number of airports that we have flight data for. Because the reducer will see every unique airport in our dataset, we can update our reduce function as follows:

```
def reduce(self):
for current, group in groupby(self, itemgetter(0)):
    self.status("reducing airport {}".format(current))
    …
    self.counter("airports")
    self.emit(current, float(total) / float(count))
```

As our analytical applications grow, these techniques to implement the full functionality of Hadoop Streaming will become vitally important. Considering natural language processing again, in order to do part-of-speech tagging or named-entity recognition, the application necessarily will have to load pickled models into memory. This process can take a few seconds up to a few minutes—using the status mechanism to alert the framework that the task is still running properly will ensure that speculative execution doesn't bog down the cluster.

Speaking of global scope, there is one last tool that helps augment Streaming applications written in Python: Job Configuration variables (JobConf variables for short). The Hadoop Streaming application will automatically add the configuration variables of the job to the environment, renaming the configuration variable by replacing dots (.) with underscores (_). For example, to access the number of mappers in the job, you would request the "mapred.map.tasks" configuration variable. Although this particular example isn't necessarily useful, user-defined configuration values can be submitted to Hadoop Streaming with the -D argument in dot notation, and could contain important information like the URL to a shared resource. To access this in Python code, add the following function:

```
import os

def get_job_conf(name):
    name = name.reptace(".", "_").upper()
    return os.environ.get(name)
```

At this point, it is clear that Python development for Hadoop Streaming would benefit from a miniature, reusable framework.

3.2.1-Counting Bigrams

Let's use the Python micro-framework to write a MapReduce application that does word counting, but with a few improvements. First, we will normalize our tokens to be all lowercase, so words like "Apple" will be the same as "apple".

Furthermore, we are going to eliminate punctuation and stopwords from our token counting. Stopwords are words that are used functionally in a language—for example, articles ("a" or "an"), determiners ("the", "this", "my"), pronouns ("his", "they") and prepositions ("over", "on", "for"). Because of their functional use, stopwords are extremely common and make up the bulk of any corpus.

Finally, we'll use our normalized corpus to count *bigrams*—that is, words that tend to appear together often (e.g., are written consecutively twice in a row while ignoring stopwords).

Using our framework from before, the Mapper does the bulk of the work:

```
#!/usr/bin/env python

import sys
import nltk
import string

from framework import Mapper


class BigramMapper(Mapper):

    def __init__(self, infile=sys.stdin, separator='\t'):
        super(BigramMapper, self).__init__(infile, separator)

        self.stopwords   = nltk.corpus.stopwords.words("english")
        self.punctuation = string.punctuation

    def exclude(self, token):
        return token in self.punctuation or token in self.stopwords

    def normalize(self, token):
        return token.lower()

    def tokenize(self, value):
        for token in nltk.wordpunct_tokenize(value):
            token = self.normalize(token)
            if not self.exclude(token):
                yield token

    def map(self):
        for value in self:
            for bigram in nltk.bigrams(self.tokenize(value)):
                self.counter("words")  # Count the total number of bigrams
                self.emit(bigram, 1)


if __name__ == "__main__":
    mapper = BigramMapper()
    mapper.map()
```

The reducer implements a very common MapReduce pattern, the SumReducer. This reducer is used so often that you might want to add it to your microframework as a standard class, along with an IdentityMapper and other standard patterns. The code for the SumReducer is as follows:

```
#!/usr/bin/env python

from framework import Reducer


class SumReducer(Reducer):

    def reduce(self):
        for key, values in self:
            total = sum(int(count) for count in values)
            self.emit(key, total)


if __name__ == '__main__':
    reducer = SumReducer()
    reducer.reduce()
```

Note that this reducer ignores the key, which is simply a text string representation of the bigram tuple, because it is just counting the occurrences of that tuple

In order to submit this job to the cluster via Hadoop Streaming, use the same command as demonstrated earlier, but make sure to also include the *framework.py* file to be packaged and sent with the job. The job submission command is as follows:

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input corpus \
    -output bigrams \
    -mapper mapper.py \
    -reducer reducer.py \
    -file mapper.py \
    -file reducer.py \
    -file framework.py
```

Note that in this case, we'll assume that the third-party dependency nltk is simply installed on every node in the cluster, which is possible if you have administrative access to the cluster, or are launching the cluster from a specific AMI.

3.3-Advanced MapReduce

The purpose here is to introduce concepts that have played a large role in MapReduce algorithms and optimizations, primarily because you will encounter these terms as you read more about how to implement different analyses.

In particular, we will discuss combiners (the primary MapReduce optimization technique), partitioned (a technique for ensuring there is no bottleneck in the reduce step), and job chaining (a technique for putting together larger algorithms and data flows).

3.3.1-Combiners

Mappers produce a lot of intermediate data that must be sent over the network to be shuffled, sorted, and reduced. Because networking is a physical resource, large amounts of transmitted data can lead to job delays and memory bottlenecks. Combiners are the primary mechanism to solve this problem, and are essentially intermediate reducers that are associated with the mapper output. Combiners reduce network traffic by performing a mapper-local reduction of the data before forwarding it on to the appropriate reducer. Consider the following output from two mappers and a simple sum reduction.

Mapper 1 output:

(IAD, 14.4), (SFO, 3.9), (JFK, 3.9), (IAD, 12.2), (JFK, 5.8)

Mapper 2 output:

(SFO, 4.7), (IAD, 2.3), (SFO, 4.4), (IAD, 1.2)

Intended sum reduce output:

(IAD, 29.1), (JFK, 9.7), (SFO, 13.0)

Each mapper is emitting extra work for the reducer, namely in the duplication of the different keys coming from each mapper. A combiner that precomputes the sums for each key will reduce the number of key/value pairs being generated, and therefore the amount of network traffic.

It is extremely common for the combiner and the reducer to be identical, which is possible if the operation is commutative and associative, but this is not always the case. So long as the combiner takes as input the type of data that the mapper is exporting and produces the same data as output, the combiner can perform any partial reduction.. To specify a combiner in Hadoop Streaming, use the -combiner option, similar to specifying a mapper and reducer:

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input input_data \
    -output output_data \
    -mapper mapper.py \
    -combiner combiner.py \
    -reducer reducer.py \
    -file mapper.py \
    -file reducer.py \
    -file combiner.py
```

If the combiner matches the reducer, then you would simply specify the *reducer.py* file as the combiner, and not add an extra third combiner file. In the microframework that we have created, a combiner class would simply subclass the Reducer.

3.3.2-Partitioners

Partitioners control how keys and their values get sent to individual reducers by dividing up the keyspace. The default behavior is the HashPartitioner, which is often all that is needed. This partitioner allocates keys evenly to each reducer by computing the hash of the key and assigning the key to a keyspace determined by the number of reducers. Given a uniformly distributed keyspace, each reducer will get a relatively equal workload.

The issue arises when there is a key imbalance, such that a large number of values are associated with one key, and other keys are less likely. In this case, a significant portion of the reducers are underworked, and much of the

benefit of reduction parallelism is lost. A custom partitioner can ease this problem by dividing the keyspace according to some other semantic structure besides hashing (which is usually domain specific). Custom partitioners can also be required for some types of MapReduce algorithms, most notably to implement a left outer join. Finally, because every reducer writes output to its own *part-** file, the use of a custom partitioner also allows for clearer data organization, allowing you to write sectioned output to each file based on the partitioning criteria—for example, writing per-year output data.

Unfortunately, custom partitioners can only be created with the Java API. However, Hadoop Streaming users can still specify a partitioner Java class either from the Hadoop library, or by writing their own Java partitioner and submitting it with their streaming job.

3.3.3-Job Chaining

Most complex algorithms cannot be described as a simple ***map*** and ***reduce***, so in order to implement more complex analytics, a technique called *job chaining* is required. If a complex algorithm can be decomposed into several smaller MapReduce tasks, then these tasks can be chained together to produce a complete output. Consider a computation to compute the pairwise Pearson correlation coefficient for a number of variables in a dataset. The Pearson correlation requires a computation of the mean and standard deviation of each variable. Because this cannot be easily accomplished in a single MapReduce, we might employ the following strategy:

1. Compute the mean and standard deviation of each (X, Y) pair.
2. Use the output of the first job to compute the covariance and Pearson correlation coefficient.

The mean and standard deviation can be computed in the initial job by mapping the total, the sum, and the sum of squares to a reducer that computes the mean and standard deviation. The second job takes the mean and standard deviation and computes the covariance by mapping the difference of the value and the mean, and their product for each pair, then reducing by appropriately summing and taking a square root. As you can see in Figure 3-2, the second job is dependent on the first.

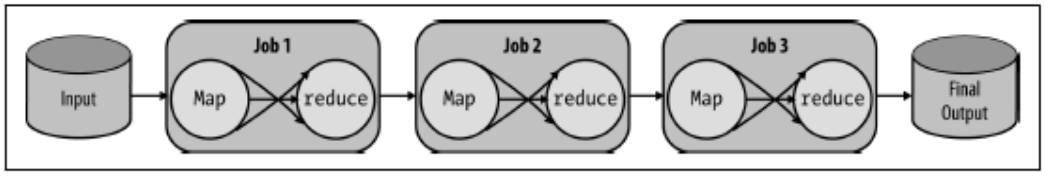

Figure 3-2. Linear job chaining produces complete computations by sending the output of one or more MapReduce jobs as the input to another

Job chaining is therefore the combination of many smaller jobs into a complete computation by sending the output of one or more previous jobs into the input of another. In order to implement algorithms like this, the developer must think about how each individual step of a computation can be reduced to intermediary values, not just between mappers and reducers but also between jobs. As shown in Figure 3-2, many jobs are typically thought of as *linear job chaining.* A linear dependency means that each MapReduce job is dependent only upon a single previous job. However, this is a simplification of the more general form of job chaining, which is expressed as a *data flow* where jobs are dependent on one or more previous jobs. Complex jobs are represented as directed acyclic graphs (DAGs) that describe how data flows from an input source through each job (the directed part) to the next job (never repeating a step, the acyclic part) and finally as final output (see Figure 3-3).

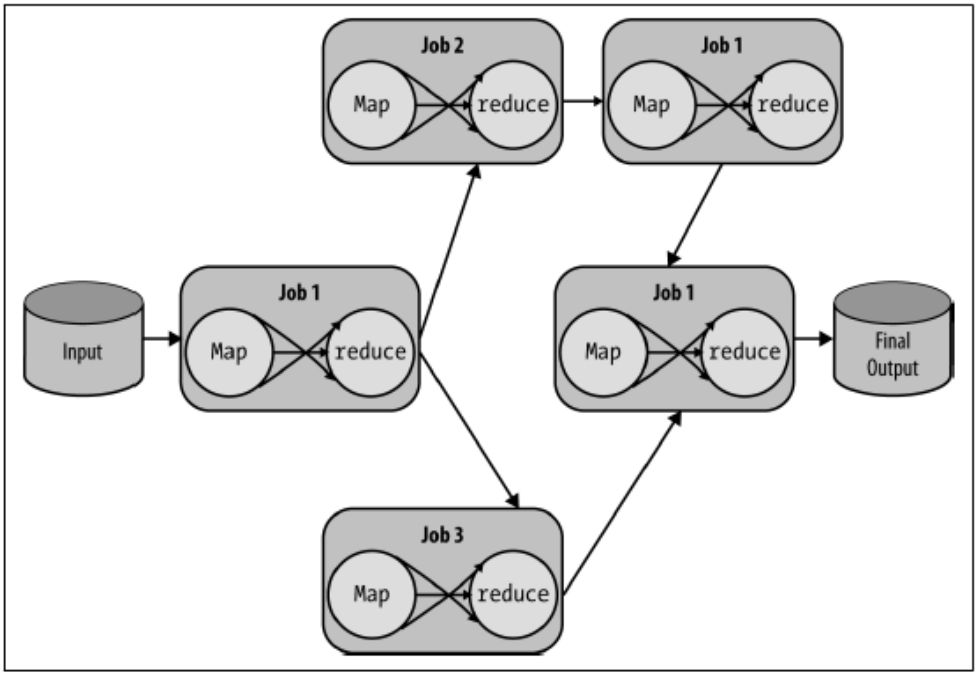

Figure 3-3. Data flow job chaining is an extension of linear chaining

4-In-Memory Computing with Spark

Together, HDFS and MapReduce have been the foundation of and the driver for the advent of large-scale machine learning, scaling analytics, and big data appliances for the last decade. The batch processing model of MapReduce was not well suited to common workflows including iterative, interactive, or on-demand computations upon a single dataset.

The primary MapReduce abstraction is parallelizable, easy to understand, and hides the details of distributed computing, thus allowing Hadoop to guarantee correctness. However, in order to achieve coordination and fault tolerance, the MapReduce model uses a pull execution model that requires intermediate writes of data back to HDFS. Worse, almost all applications must chain multiple MapReduce jobs together in multiple steps, creating a data flow toward the final required result. This results in huge amounts of intermediate data written to HDFS that is not required by the user, creating additional costs in terms of disk usage.

To address these problems, Hadoop has moved to a more general resource management framework for computation: YARN. YARN provides more general resource access to Hadoop applications. The result is that specialized tools no longer have to be decomposed into a series of MapReduce jobs and can become more complex.

Spark is the first fast, general-purpose distributed computing paradigm resulting from this shift, and is rapidly gaining popularity particularly because of its speed and adaptability. Spark primarily achieves this speed via a new data model called resilient distributed datasets (RDDs) that are stored in memory while being computed upon, thus eliminating expensive intermediate disk writes. It also takes advantage of a directed acyclic graph (DAG) execution engine that can optimize computation, particularly iterative computation, which is essential for data theoretic tasks such as optimization and machine learning. These speed gains allow Spark to be accessed in an interactive fashion (as though you were sitting at the Python interpreter), making the user an integral part of computation and allowing for data exploration of big datasets that was not previously possible, bringing the cluster to the data scientist.

4.1-Spark Basics

Apache Spark is a cluster-computing platform that provides an API for distributed programming similar to the MapReduce model, but is designed to be fast for interactive queries and iterative algorithms. It primarily achieves this by caching data required for computation in the memory of the nodes in the cluster. In-memory cluster computation enables Spark to run iterative algorithms, as programs can checkpoint data and refer back to it without reloading it from disk; in addition, it supports interactive querying and streaming data analysis at extremely fast speeds. Because Spark is compatible with YARN, it can run on an existing Hadoop cluster and access any Hadoop data source, including HDFS, S3, HBase, and Cassandra.

Importantly, Spark was designed from the ground up to support big data applications and data science in particular. Instead of a programming model that only supports map and reduce, the Spark API has many other powerful distributed abstractions similarly related to functional programming, including sample, filter, join, and collect, to name a few.

In order to understand the shift, consider the limitations of MapReduce with regards to iterative algorithms. These types of algorithms apply the same operation many times to blocks of data until they reach a desired result. Here, the algorithm applies the target function with one set of parameters to the entire dataset and computes the error, afterward slightly modifying the parameters of the function according to the computed error (descending down the error curve). This process is repeated (the iterative part) until the error is minimized below some threshold or until a maximum number of iterations is reached.

This basic technique is the foundation of many machine learning algorithms, particularly supervised learning, in which the correct answers are known ahead of time and can be used to optimize some decision space. In order to program this type of algorithm in MapReduce, the parameters of the target function would have to be mapped to every instance in the dataset, and the error computed and reduced. After the reduce phase, the parameters would be updated and fed into the next MapReduce job.

Instead, Spark keeps the dataset in memory as much as possible throughout the course of the application, preventing the reloading of data between iterations. Spark programmers therefore do not simply specify map and reduce steps, but rather an entire series of data flow transformations to be applied to the input data before performing some action that requires coordination like a reduction or a write to disk. Because data flows can be described using directed acyclic graphs (DAGs), Spark's execution engine knows ahead of time how to distribute the computation across the cluster and manages the details of the computation.

By combining acyclic data flow and in-memory computing, Spark is extremely fast particularly when the cluster is large enough to hold all of the data in memory. In fact, by increasing the size of the cluster and therefore the amount of available memory to hold an entire, very large dataset, the speed of Spark means that it can be used *interactively*—making the user a key participant of analytical processes that are running on the cluster. As Spark evolved, the notion of user interaction became essential to its model of distributed computation; in fact, it is probably for this reason that so many languages are supported.

4.1.1-The Spark Stack

Spark is a general-purpose distributed computing abstraction and can run in a standalone mode. However, Spark focuses purely on *computation* rather than *data storage* and as such is typically run in a cluster that implements data warehousing and cluster management tools. When Spark is built with Hadoop, it utilizes YARN to allocate and manage cluster resources like processors and memory via the ResourceManager. Importantly, Spark can then

access any Hadoop data source—for example HDFS, HBase, or Hive, to name a few.

Spark exposes its primary programming abstraction to developers through the Spark Core module. This module contains basic and general functionality, including the API that defines resilient distributed datasets (RDDs). RDDs, which we will describe in more detail in the next section, are the essential functionality upon which all Spark computation resides. Spark then builds upon this core, implementing special-purpose libraries for a variety of data science tasks that interact with Hadoop, as shown in Figure 4-1.

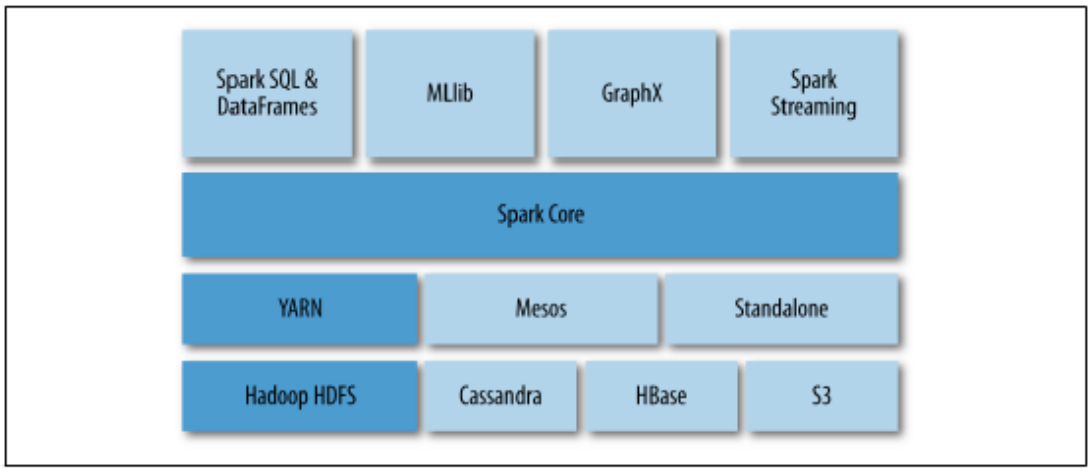

Figure 4-1. Spark is a computational framework designed to take advantage of cluster management platforms like YARN and distributed data storage like HDFS

The primary components included with Spark are as follows:

Spark SQL

Originally provided APIs for interacting with Spark via the Apache Hive variant of SQL called HiveQL; in fact, you can still directly access Hive via this library. However, this library is moving toward providing a more general, structured data-processing abstraction, DataFrames. DataFrames are essentially distributed collections of data organized into columns, conceptually similar to tables in relational databases.

Spark Streaming

Enables the processing and manipulation of unbounded streams of data in real time. Many streaming data libraries (such as Apache Storm) exist for handling real-time data. Spark Streaming enables programs to leverage this data similar to how you would interact with a normal RDD as data is flowing in.

MLlib

A library of common machine learning algorithms implemented as Spark operations on RDDs. This library contains scalable learning algorithms (e.g., classifications, regressions, etc.). that require iterative operations across large datasets. The Mahout library, formerly the big data machine learning library of choice, will move to Spark for its implementations in the future.

GraphX

A collection of algorithms and tools for manipulating graphs and performing parallel graph operations and computations. GraphX extends the RDD API to include operations for manipulating graphs, creating subgraphs, or accessing all vertices in a path.

These components combined with the Spark programming model provide a rich methodology of interacting with cluster resources. It is probably because of this completeness that Spark has become so immensely popular for distributed analytics.

4.1.2-Resilient Distributed Datasets

RDDs are essentially a programming abstraction that represents a read-only collection of objects that are partitioned across a set of machines. RDDs can be rebuilt from a lineage (and are therefore fault tolerant), are accessed via parallel operations, can be read from and written to distributed storages (e.g., HDFS or S3), and most importantly, can be cached in the memory of worker nodes for immediate reuse.

RDDs are operated upon with functional programming constructs that include and expand upon map and reduce. Programmers create new RDDs by loading data from an input source, or by transforming an existing collection to generate a new one. The history of applied transformations is primarily what defines the RDD's *lineage*, and because the collection is *immutable* (not directly modifiable), transformations can be reapplied to part or all of the collection in order to recover from failure. The Spark API is therefore essentially a collection of operations that create, transform, and export RDDs.

The fundamental programming model therefore is describing how RDDs are created and modified via programmatic operations. There are two types of operations that can be applied to RDDs: *transformations* and *actions*. Transformations are operations that are applied to an existing RDD to create a new RDD—for example, applying a filter operation on an RDD to generate a smaller RDD of filtered values. Actions, however, are operations that actually return a result back to the Spark driver program —resulting in a coordination or aggregation of all partitions in an RDD.

An additional benefit of Spark is that it applies transformations "lazily"—inspecting a complete sequence of transformations and an action before executing them by submitting a job to the cluster. This lazy-execution provides significant storage and computation optimizations, as it allows Spark to build up a lineage of the data and evaluate the complete transformation chain in order to compute upon only the data needed for a result; for example, if you run the first() action on an RDD, Spark will avoid reading the entire dataset and return just the first matching line.

4.1.3-Programming with RDDs

Code is written in a driver program that is evaluated lazily on the driver-local machine when submitted, and upon an action, the driver code is distributed across the cluster to be executed by workers on their partitions of the RDD. Results are then sent back to the driver for aggregation or compilation. As illustrated in Figure 4-2, the driver program creates one or more RDDs by parallelizing a dataset from a Hadoop data source, applies operations to transform the RDD, then invokes some action on the transformed RDD to retrieve output.

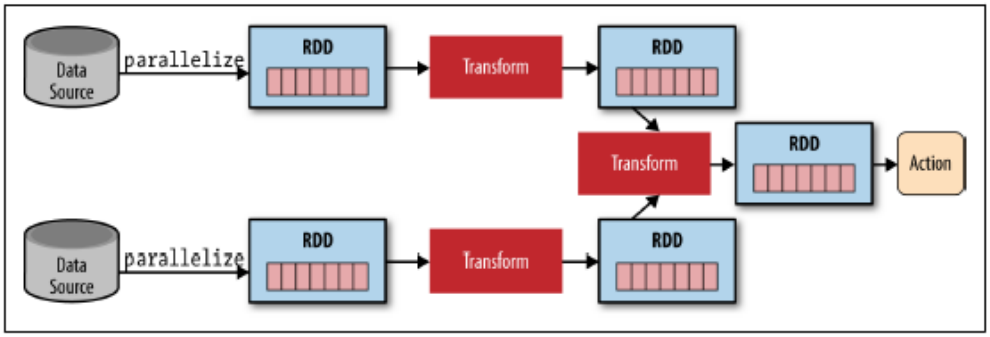

Figure 4-2. A typical Spark application parallelizes (partitions) a dataset across a cluster into RDDs

A typical data flow sequence for programming Spark is as follows:

1. Define one or more RDDs, either through accessing data stored on disk (e.g., HDFS, Cassandra, HBase, or S3), parallelizing some collection, transforming an existing RDD, or by caching. Caching is one of the fundamental procedures in Spark-storing an RDD in the memory of a node for rapid access as the computation progresses.
2. Invoke operations on the RDD by passing closures (here, a function that does not rely on external variables or data) to each element of the RDD. Spark offers many high-level operators beyond map and reduce.
3. Use the resulting RDDs with aggregating actions (e.g., count, collect, save, etc.). Actions kick off the computation on the cluster because no progress can be made until the aggregation has been computed.

When Spark runs a closure on a worker, any variables used in the closure are copied to that node, but are maintained within the local scope of that closure. If external data is required, Spark provides two types of shared variables that can be interacted with by all workers in a restricted fashion: *broadcast variables* and *accumulators*. Broadcast variables are distributed to all workers, but are read-only and are often used as lookup tables or stopword lists. Accumulators are variables that workers can "add" to using associative operations and are typically used as counters.

4.2-Interactive Spark Using PySpark

We'll begin exploring how to use RDDs with pyspark. In order to run the interactive shell, you will need to locate the *pyspark* command, which is in the *bin* directory of the Spark library. You should also have a $SPARK_HOME. Spark requires no configuration to run right off the bat, so simply downloading the Spark build for your system is enough. Replacing $SPARK_HOME with the download path (or setting your environment), you can run the interactive shell as follows:

```
hostname $ $SPARK_HOME/bin/pyspark
[… snip …]
>>>
```

PySpark automatically creates a SparkContext for you to work with, using the local Spark configuration. It is exposed to the terminal via the sc variable. Let's create our first RDD:

```
>>> text = sc.textFileCshakespeare.txt")
>>> print text
shakespeare.txt MappedRDD[1] at textFite at NativeMethodAccessorImpl.java:-2
```

The textFile method loads the complete works of William Shakespeare from the local disk into an RDD named text. If you inspect the RDD, you can see that it is a MappedRDD and that the path to the file is a relative path from the current working directory (pass in a correct path to the *shakespeare.txt* file on your system). Similar to our MapReduce example in Chapter 2, let's start to transform this RDD in order to compute the "Hello, World" of distributed computing and implement the word count application using Spark:

```
>>> from operator import add
>>> def tokenize(text):
... return text.split()
…
>>> words = text.flatMap(tokenize)
```

We imported the operator add, which is a named function that can be used as a closure for addition. The first thing we have to do is split our text into words. We created a function called tokenize whose argument is some piece

of text and returns a list of the tokens (words) in that text by simply splitting on whitespace. We then created a new RDD called words by transforming the text RDD through the application of the flatMap operator, and passed it the closure tokenize.

We can therefore continue to apply transformations to this RDD without waiting for a long, possibly erroneous or non-optimal distributed execution to take place. First, let's apply our map:

```
>>> wc = words.map(lambda x: (x,1))
>>> print wc.toDebugString()
(2) PythonRDD[3] at RDD at PythonRDD.scala:43
| shakespeare.txt MappedRDD[1] at textFile at NativeMethodAccessorImpl.java:-2
| shakespeare.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:-2
```

Instead of using a named function, we will use an anonymous function (with the lambda keyword in Python). This line of code will map the lambda to each element of words. Therefore, each x is a word, and the word will be transformed into a tuple (word, 1) by the anonymous closure. In order to inspect the lineage so far, we can use the toDebugString method to see how our PipelinedRDD is being transformed. We can then apply the reduceByKey action to get our word counts and then write those word counts to disk:

```
>>> counts = wc.reduceByKey(add)
>>> counts.saveAsTextFile("wc")
```

Once we finally invoke the action saveAsTextFile, the distributed job kicks off and you should see a lot of INFO statements as the job runs "across the cluster". If you exit the interpreter, you should see a directory called wc in your current working directory:

```
hostname $ ls wc/
_SUCCESS part-00000 part-00001
```

Each part file represents a partition of the final RDD that was computed by various processes on your computer and saved to disk. If you use the head command on one of the part files, you should see tuples of word count pairs:

```
hostname $ head wc/part-00000
(u'fawn', 14)
(u'Fame.', 1)
(u'Fame,', 2)
(u'kinghenryviii@7731', 1)
(u'othetlo@36737', 1)
(u'loveslabourslost@51678', 1)
(u'1kinghenryiv@54228', 1)
(u'troilusandcressida@83747', 1)
(u'fleeces', 1)
(u'midsummersnightsdream@71681', 1)
```

Note that in a MapReduce job, the keys would be sorted due to the mandatory intermediate shuffle and sort phase between map and reduce. Spark's repartitioning for reduction does not necessarily utilize a sort because all executors can communicate with each other and as a result, the preceding output is not sorted lexicographically. Even without the sort, however, you are guaranteed that each key appears only once across all part files because the reduceByKey operator was used to aggregate the counts RDD. If sorting is necessary, you could use the sort operator to ensure that all the keys are sorted before writing them to disk.

4.3-Writing Spark Applications

You need to create a complete, executable driver program to submit to the cluster. Many Spark programs are therefore simple Python scripts that contain some data (shared variables), define closures for transforming RDDs, and describe a step-bystep execution plan of RDD transformation and aggregation. A basic template for writing a Spark application in Python is as follows:

```
## Spark Application - execute with spark-submit

## Imports
from pyspark import SparkConf, SparkContext

## Shared variables and data
APP_NAME = "My Spark Application"

## Closure functions

## Main functionality
def main(sc):
    """
    Describe RDD transformations and actions here.
    """
    pass

if __name__ == "__main__":
    # Configure Spark
    conf = SparkConf().setAppName(APP_NAME)
    conf = conf.setMaster("local[*]")
    sc   = SparkContext(conf=conf)

    # Execute main functionality
    main(sc)
```

This template exposes the top-down structure of a Python Spark application: imports allow various Python libraries to be used for analysis as well as Spark components such as GraphX or SparkSQL. Shared data and variables are specified as module constants, including an identifying application name that is used in web Uls, for debugging, and in logging. Job-specific closures or custom operators are included with the driver program for easy debugging or to be imported in other Spark jobs, and finally some main method defines the analytical methodology that transforms and aggregates RDDs, which is run as the driver program.

The driver program defines the entirety of the Spark execution; for example, to stop or exit the program in code, programmers can use sc.stop() or sys.exit(0). This control extends to the execution environment as well—in this template, a Spark cluster configuration, local[*] is hardcoded into the SparkConf via the setMaster method. This tells Spark to run on the local machine using as many processes as available (multiprocess, but not distributed computation). While you can specify where Spark executes on the command line using spark-submit, driver programs often select this based on an environment variable using os.environ. Therefore, while developing Spark jobs (e.g., using a DEBUG variable), the job can be run locally, but in production run across the cluster on a larger data set.