Chapter 8

MapReduce Fundamentals

In This Chapter

- The origins of MapReduce
- Looking at the map function
- Exploring the reduce function
- Putting map and reduce together
- Optimizing MapReduce tasks

While big data has dominated the headlines over the past year, large computing problems have existed since the beginning of the computer era. Each time a newer, faster, higher-capacity computer system was introduced, people found problems that were too big for the system to handle. Along came local-area networks, and the industry turned to combining the compute and storage capacities of systems on the network toward solving bigger and bigger problems. The distribution of compute- and data-intensive applications is at the heart of a solution to big data challenges. To best achieve reliable distribution at scale, new technology approaches were needed. MapReduce is one of those new approaches. MapReduce is a software framework that enables developers to write programs that can process massive amounts of unstructured data in parallel across a distributed group of processors.

Tracing the Origins of MapReduce

In the early 2000s, some engineers at Google looked into the future and determined that while their current solutions for applications such as web crawling, query frequency, and so on were adequate for most existing requirements, they were inadequate for the complexity they anticipated as the web scaled to more and more users. These engineers determined that if work could be distributed across inexpensive computers and then connected on the network in the form of a "cluster," they could solve the problem.

Distribution alone was not a sufficient answer. This distribution of work must be performed in parallel for the following three reasons:

- The processing must be able to expand and contract automatically.
- The processing must be able to proceed regardless of failures in the network or the individual systems.
- Developers leveraging this approach must be able to create services that are easy to leverage by other developers. Therefore, this approach must be independent of where the data and computations have executed.

MapReduce was designed as a generic programming model. Some of the initial implementations provided all the key requirements of parallel execution, fault tolerance, load balancing, and data manipulation. The engineers in charge of the project named the initiative MapReduce because it combines two capabilities from existing functional computer languages: *map* and *reduce*.

Google engineers designed MapReduce to solve a specific practical problem. Therefore, it was designed as a programming model combined with the implementation of that model — in essence, a reference implementation. The reference implementation was used to demonstrate the practicality and effectiveness of the concept and to help ensure that this model would be widely adopted by the computer industry. Over the years, other implementations of MapReduce have been created and are available as both open source and commercial products.

Functional versus procedural programming models

When we talk of map and reduce, we do so as operations within a functional programming model. Functional programming is one of the two ways that software developers create programs to address business problems. The other model is procedural programming. We take a quick look to understand the differences and to see when it's best to use one or the other model.

Procedural programs are highly structured and provide step-by-step instructions on what to do with input data. The order of the execution is important, and the input data is changed as it progresses through each step of the program. Examples of procedural languages include FORTRAN, COBOL, C, and C++. The best uses for procedural programs are those where it is okay to change the values of the input data or where you need to compare computed values in one of the steps to determine whether you need to continue processing or exit the program and deliver the result.

In contrast, functional programs do not change the input data. They look at all the data for specific patterns and then apply rules to identify the important elements and then assemble them into lists. The order of the processing is not important because each operation is independent of another. Examples of functional languages include LISP, Scheme, Prolog, and R. Functional programs do not change the input data and are most often used when it is necessary to look at the data again and again for different patterns. For example, you could look through a list of all the counties in the United States that voted Republican in the last election and then go through the list for all Democratic counties. This will produce two distinct output lists.

Understanding the map Function

The *map* function has been a part of many functional programming languages for years, first gaining popularity with an artificial intelligence language called LISP. Good software developers understand the value of reuse, so map has been reinvigorated as a core technology for processing lists of data elements (keys and values). To further your understanding of why the map function is a good choice for big data (and the *reduce* function is as well), it's important to understand a little bit about functional programming.

Operators in functional languages do not modify the structure of the data; they create new data structures as their output. More importantly, the original data itself is unmodified as well. So you can use the map function with impunity because it will not harm your precious stored data. Another advantage to functional programming is not having to expressly manage the movement or flow of the data. This is helpful because it absolves the programmer from explicitly managing the data output and placement. Because you are operating in a distributed environment, dealing with where the data is stored can be a nightmare. The map function takes care of that. Finally, in the world of functional programming, the order of the operations on the data is not prescribed. Again, this is a great advantage in a computing cluster where tasks are being performed in parallel.

So what exactly can you expect from the map function? It applies a function to each element (defined as a key-value pair) of a list and produces a new list. Suppose that you wanted to create a program that counts the number of characters in a series or list of words. The following is not official programming code; it's just a way to represent how to construct a solution to the problem.

One way to accomplish the solution is to identify the input data and create a list:

```
mylist = ("all counties in the US that participated in the
          most recent general election")
```

Create the function `howManyPeople` using the map function. This selects only the counties with more than 50,000 people:

```
map howManyPeople (mylist) = [ howManyPeople "county 1";
          howManyPeople "county 2"; howManyPeople "county
          3"; howManyPeople "county 4"; . . . ]
```

Now produce a new output list of all the counties with populations greater than 50,000:

```
(no, county 1; yes, county 2; no, county 3; yes, county 4;
          ?, county nnn)
```

The function executes without making any changes to the original list. In addition, you can see that each element of the output list maps to a corresponding element of the input list, with a `yes` or `no` attached. If the county has met the requirement of more than 50,000 people, the map function identifies it with a `yes`. If not, a `no` is indicated. This is an important feature, as you shall soon see when you look at the *reduce* function.

Adding the reduce Function

Like the map function, *reduce* has been a feature of functional programming languages for many years. In some languages, it is called *fold,* but the behavior is exactly the same. The reduce function takes the output of a map function and "reduces" the list in whatever fashion the programmer desires. The first step that the reduce function requires is to place a value in something called an *accumulator,* which holds an initial value. After storing a starting value in the accumulator, the reduce function then processes each element of the list and performs the operation you need across the list. At the end of the list, the reduce function returns a value based on what operation you wanted to perform on the output list. Revisit the map function example now to see what the reduce function is capable of doing.

Suppose that you need to identify the counties where the majority of the votes were for the Democratic candidate. Remember that your `howMany-People` map function looked at each element of the input list and created an output list of the counties with more than 50,000 people (`yes`) and the counties with less than 50,000 people (`no`).

After invoking the `howManyPeople` map function, you are left with the following output list:

```
(no, county 1; yes, county 2; no, county 3; yes, county 4;
          ?, county nnn)
```

This is now the input for your reduce function. Here is what it looks like:

```
countylist = (no, county 1; yes, county 2; no, county 3;
          yes, county 4; ?, county nnn)
reduce isDemocrat (countylist)
```

The reduce function processes each element of the list and returns a list of all the counties with a population greater than 50,000, where the majority voted Democratic.

Now imagine that you would like to know in which counties with a population greater than 50,000 the majority voted Republican. All you need to do is invoke the reduce function again, but you will change the operator from `isDemocrat` to `isRepublican`:

```
reduce isRepublican (countylist)
```

This returns a list of all the counties where the majority of voters supported Republican candidates. Because you did not change the elements of `county list`, you can continue to perform the reduce functions on the input until you get the results you require. For example, you could look for independent majorities or refine the results to specific geographic regions.

Putting map and reduce Together

Sometimes producing an output list is just enough. Likewise, sometimes performing operations on each element of a list is enough. Most often, you want to look through large amounts of input data, select certain elements from the data, and then compute something of value from the relevant pieces of data. You don't always control the input data, so you need to do this work nondestructively — you don't want to change that input list so you can use it in different ways with new assumptions and new data.

Software developers design applications based on algorithms. An *algorithm* is nothing more than a series of steps that need to occur in service to an overall goal. It is very much like a cooking recipe. You start with the individual elements (flour, sugar, eggs, and so on) and follow step-by-step instructions (combine, knead, and bake) to produce the desired result (a loaf of bread). Putting the map and reduce functions to work efficiently requires an algorithm too. It might look a little like this:

1. Start with a large number or data or records.
2. Iterate over the data.
3. Use the map function to extract something of interest and create an output list.
4. Organize the output list to optimize for further processing.
5. Use the reduce function to compute a set of results.
6. Produce the final output.

Programmers can implement all kinds of applications using this approach, but the examples to this point have been very simple, so the real value of

MapReduce may not be apparent. What happens when you have extremely large input data? Can you use the same algorithm on terabytes of data? The good news is yes.

As illustrated in Figure 8-1, all of the operations seem independent. That's because they are. The real power of MapReduce is the capability to divide and conquer. Take a very large problem and break it into smaller, more manageable chunks, operate on each chunk independently, and then pull it all together at the end. Furthermore, the map function is commutative — in other words, the order that a function is executed doesn't matter.

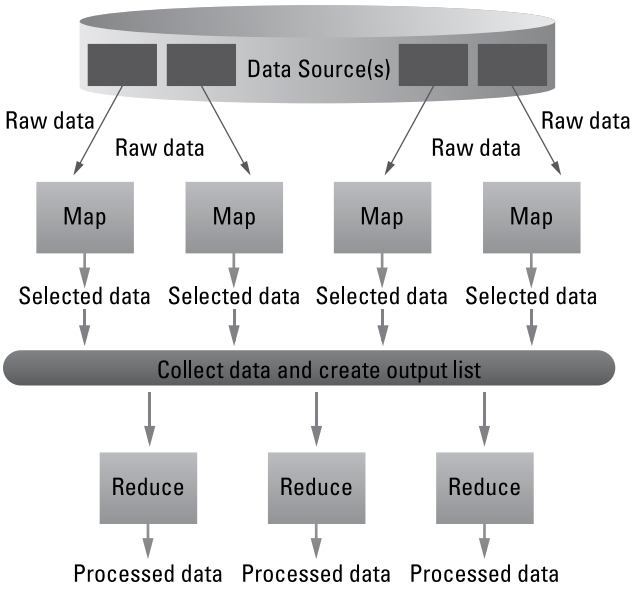

Figure 8-1: Data flow in MapReduce.

If you remember algebra at all, you may recall that when something is commutative, the result is the same, regardless of the order of the elements. For example:

```
5 + 7 = 7 + 5
```

or

```
3 * 4 = 4 * 3
```

So MapReduce can perform its work on different machines in a network and get the same result as if all the work was done on a single machine. It can also draw from multiple data sources, internal or external. MapReduce keeps

track of its work by creating a unique key to ensure that all the processing is related to solving the same problem. This key is also used to pull all the output together at the end of all the distributed tasks.

When the map and reduce functions are used in this fashion, they work collectively to run as a single job within the cluster. All the dividing and conquering is done transparently by the execution framework of the MapReduce engine, and all the work is distributed to one or many nodes in the network.

You need to understand some characteristics of the execution framework so that you may get a better understanding of why things work the way they do. This can help you design better applications and also to optimize the execution for performance or efficiency. The following are the foundational behaviors of MapReduce:

- **Scheduling:** MapReduce jobs get broken down into individual tasks for the map and the reduce portions of the application. Because the mapping must be concluded before reducing can take place, those tasks are prioritized according to the number of nodes in the cluster. If you have more tasks than nodes, the execution framework will manage the map tasks until all are complete. Then the reduce tasks will run with the same behaviors. The entire process is complete only when all the reduce tasks have run successfully.
- **Synchronization:** When multiple processes execute concurrently in a cluster, you need a way to keep things running smoothly. Synchronization mechanisms do this automatically. Because the execution framework knows that the program is mapping and reducing, it keeps track of what has run and when. When all the mapping is complete, the reducing begins. Intermediate data is copied over the network as it is produced using a mechanism called "shuffle and sort." This gathers and prepares all the mapped data for reduction.
- **Code/data colocation:** The most effective processing occurs when the mapping functions (the code) is colocated on the same machine with the data it needs to process. The process scheduler is very clever and can place the code and its related data on the same node prior to execution (or vice versa).
- **Fault/error handling:** What happens when a failure occurs? Hopefully, nothing. Most MapReduce engines have very robust error handling and fault tolerance. With all the nodes in a MapReduce cluster and all the parts in each node, something is going to fail at some point. The engine must recognize that something is wrong and make the necessary correction. For example, if some of the mapping tasks do not return as complete, the engine could assign the tasks to a different node to finish the job. The engine is designed so that it recognizes when a job is incomplete and will automatically assign the task to a different node.

Optimizing MapReduce Tasks

Aside from optimizing the actual application code, you can use some optimization techniques to improve the reliability and performance of your MapReduce jobs. They fall into three categories: hardware/network topology, synchronization, and file system.

Hardware/network topology

Independent of application, the fastest hardware and networks will likely yield the fastest run times for your software. A distinct advantage of MapReduce is the capability to run on inexpensive clusters of commodity hardware and standard networks. If you don't pay attention to where your servers are physically organized, you won't get the best performance and high degree of fault tolerance necessary to support big data tasks.

Commodity hardware is often stored in racks in the data center. The proximity of the hardware within the rack offers a performance advantage as opposed to moving data and/or code from rack to rack. During implementation, you can configure your MapReduce engine to be aware of and take advantage of this proximity. Keeping the data and the code together is one of the best optimizations for MapReduce performance. In essence, the closer the hardware processing elements are to each other, the less latency you will have to deal with.

Synchronization

Because it is inefficient to hold all the results of your mapping within the node, the synchronization mechanisms copy the mapping results to the reducing nodes immediately after they have completed so that the processing can begin right away. All values from the same key are sent to the same reducer, again ensuring higher performance and better efficiency. The reduction outputs are written directly to the file system, so it must be designed and tuned for best results.

File system

Your MapReduce implementation is supported by a distributed file system. The major difference between local and distributed file systems is capacity. To handle the huge amounts of information in a big data world, file

systems need to be spread across multiple machines or nodes in a network. MapReduce implementations rely on a master-slave style of distribution, where the master node stores all the metadata, access rights, mapping and location of files and blocks, and so on. The slaves are nodes where the actual data is stored. All the requests go to the master and then are handled by the appropriate slave node. As you contemplate the design of the file system you need to support a MapReduce implementation, you should consider the following:

- **Keep it warm:** As you might expect, the master node could get overworked because everything begins there. Additionally, if the master node fails, the entire file system is inaccessible until the master is restored. A very important optimization is to create a "warm standby" master node that can jump into service if a problem occurs with the online master.
- **The bigger the better:** File size is also an important consideration. Lots of small files (less than 100MB) should be avoided. Distributed file systems supporting MapReduce engines work best when they are populated with a modest number of large files.
- **The long view:** Because workloads are managed in batches, highly sustained network bandwidth is more important than quick execution times of the mappers or reducers. The optimal approach is for the code to stream lots of data when it is reading and again when it is time to write to the file system.
- **Keep it secure:** But not overly so. Adding layers of security on the distributed file system will degrade its performance. The file permissions are there to guard against unintended consequences, not malicious behavior. The best approach is to ensure that only authorized users have access to the data center environment and to keep the distributed file system protected from the outside.

Now that you understand a bit about this powerful capability, we take a deep dive into the most widely used MapReduce engine and its ecosystem.