Chapter 2

MapReduce and the New Software Stack

Modern data-mining applications, often called "big-data" analysis, require us to manage immense amounts of data quickly. In many of these applications, the data is extremely regular, and there is ample opportunity to exploit parallelism. Important examples are:

1. The ranking of Web pages by importance, which involves an iterated matrix-vector multiplication where the dimension is many billions.
2. Searches in "friends" networks at social-networking sites, which involve graphs with hundreds of millions of nodes and many billions of edges.

To deal with applications such as these, a new software stack has evolved. These programming systems are designed to get their parallelism not from a "supercomputer," but from "computing clusters" – large collections of commodity hardware, including conventional processors ("compute nodes") connected by Ethernet cables or inexpensive switches. The software stack begins with a new form of file system, called a "distributed file system," which features much larger units than the disk blocks in a conventional operating system. Distributed file systems also provide replication of data or redundancy to protect against the frequent media failures that occur when data is distributed over thousands of low-cost compute nodes.

On top of these file systems, many different higher-level programming systems have been developed. Central to the new software stack is a programming system called *MapReduce*. Implementations of MapReduce enable many of the most common calculations on large-scale data to be performed on computing clusters efficiently and in a way that is tolerant of hardware failures during the computation.

MapReduce systems are evolving and extending rapidly. Today, it is common for MapReduce programs to be created from still higher-level programming

systems, often an implementation of SQL. Further, MapReduce turns out to be a useful, but simple, case of more general and powerful ideas. We include in this chapter a discussion of generalizations of MapReduce, first to systems that support acyclic workflows and then to systems that implement recursive algorithms.

Our last topic for this chapter is the design of good MapReduce algorithms, a subject that often differs significantly from the matter of designing good parallel algorithms to be run on a supercomputer. When designing MapReduce algorithms, we often find that the greatest cost is in the communication. We thus investigate communication cost and what it tells us about the most efficient MapReduce algorithms. For several common applications of MapReduce we are able to give families of algorithms that optimally trade the communication cost against the degree of parallelism.

2.1 Distributed File Systems

Most computing is done on a single processor, with its main memory, cache, and local disk (a *compute node*). In the past, applications that called for parallel processing, such as large scientific calculations, were done on special-purpose parallel computers with many processors and specialized hardware. However, the prevalence of large-scale Web services has caused more and more computing to be done on installations with thousands of compute nodes operating more or less independently. In these installations, the compute nodes are commodity hardware, which greatly reduces the cost compared with special-purpose parallel machines.

These new computing facilities have given rise to a new generation of programming systems. These systems take advantage of the power of parallelism and at the same time avoid the reliability problems that arise when the computing hardware consists of thousands of independent components, any of which could fail at any time. In this section, we discuss both the characteristics of these computing installations and the specialized file systems that have been developed to take advantage of them.

2.1.1 Physical Organization of Compute Nodes

The new parallel-computing architecture, sometimes called *cluster computing*, is organized as follows. Compute nodes are stored on *racks*, perhaps 8–64 on a rack. The nodes on a single rack are connected by a network, typically gigabit Ethernet. There can be many racks of compute nodes, and racks are connected by another level of network or a switch. The bandwidth of inter-rack communication is somewhat greater than the intrarack Ethernet, but given the number of pairs of nodes that might need to communicate between racks, this bandwidth may be essential. Figure 2.1 suggests the architecture of a large-scale computing system. However, there may be many more racks and many more compute nodes per rack.

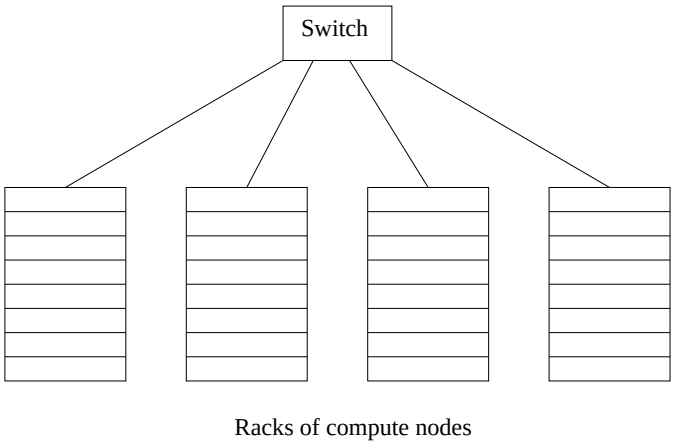

Figure 2.1: Compute nodes are organized into racks, and racks are interconnected by a switch

It is a fact of life that components fail, and the more components, such as compute nodes and interconnection networks, a system has, the more frequently something in the system will not be working at any given time. For systems such as Fig. 2.1, the principal failure modes are the loss of a single node (e.g., the disk at that node crashes) and the loss of an entire rack (e.g., the network connecting its nodes to each other and to the outside world fails).

Some important calculations take minutes or even hours on thousands of compute nodes. If we had to abort and restart the computation every time one component failed, then the computation might never complete successfully. The solution to this problem takes two forms:

1. Files must be stored redundantly. If we did not duplicate the file at several compute nodes, then if one node failed, all its files would be unavailable until the node is replaced. If we did not back up the files at all, and the disk crashes, the files would be lost forever. We discuss file management in Section 2.1.2.

2. Computations must be divided into tasks, such that if any one task fails to execute to completion, it can be restarted without affecting other tasks. This strategy is followed by the MapReduce programming system that we introduce in Section 2.2.

2.1.2 Large-Scale File-System Organization

To exploit cluster computing, files must look and behave somewhat differently from the conventional file systems found on single computers. This new file system, often called a *distributed file system* or *DFS* (although this term has had other meanings in the past), is typically used as follows.

DFS Implementations

There are several distributed file systems of the type we have described that are used in practice. Among these:

1. The *Google File System* (GFS), the original of the class.
2. *Hadoop Distributed File System* (HDFS), an open-source DFS used with Hadoop, an implementation of MapReduce (see Section 2.2) and distributed by the Apache Software Foundation.
3. *CloudStore*, an open-source DFS originally developed by Kosmix.

- Files can be enormous, possibly a terabyte in size. If you have only small files, there is no point using a DFS for them.
- Files are rarely updated. Rather, they are read as data for some calculation, and possibly additional data is appended to files from time to time. For example, an airline reservation system would not be suitable for a DFS, even if the data were very large, because the data is changed so frequently.

Files are divided into *chunks*, which are typically 64 megabytes in size. Chunks are replicated, perhaps three times, at three different compute nodes. Moreover, the nodes holding copies of one chunk should be located on different racks, so we don't lose all copies due to a rack failure. Normally, both the chunk size and the degree of replication can be decided by the user.

To find the chunks of a file, there is another small file called the *master node* or *name node* for that file. The master node is itself replicated, and a directory for the file system as a whole knows where to find its copies. The directory itself can be replicated, and all participants using the DFS know where the directory copies are.

2.2 MapReduce

MapReduce is a style of computing that has been implemented in several systems, including Google's internal implementation (simply called MapReduce) and the popular open-source implementation Hadoop which can be obtained, along with the HDFS file system from the Apache Foundation. You can use an implementation of MapReduce to manage many large-scale computations in a way that is tolerant of hardware faults. All you need to write are two functions, called *Map* and *Reduce*, while the system manages the parallel execution, coordination of tasks that execute Map or Reduce, and also deals with

the possibility that one of these tasks will fail to execute. In brief, a MapReduce computation executes as follows:

1. Some number of Map tasks each are given one or more chunks from a distributed file system. These Map tasks turn the chunk into a sequence of *key-value* pairs. The way key-value pairs are produced from the input data is determined by the code written by the user for the Map function.

2. The key-value pairs from each Map task are collected by a *master controller* and sorted by key. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task.

3. The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way. The manner of combination of values is determined by the code written by the user for the Reduce function.

Figure 2.2 suggests this computation.

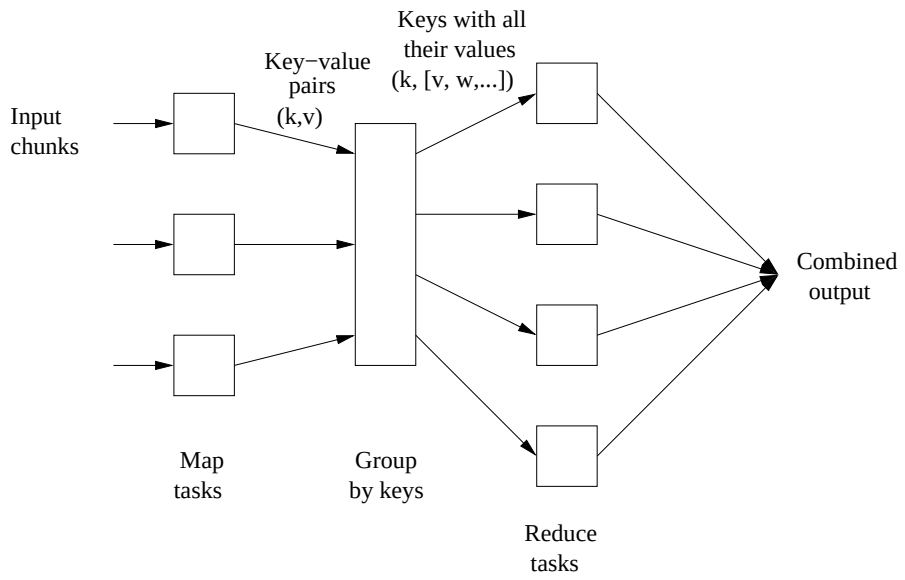

Figure 2.2: Schematic of a MapReduce computation

2.2.1 The Map Tasks

We view input files for a Map task as consisting of *elements*, which can be any type: a tuple or a document, for example. A chunk is a collection of elements, and no element is stored across two chunks. Technically, all inputs

to Map tasks and outputs from Reduce tasks are of the key-value-pair form, but normally the keys of input elements are not relevant and we shall tend to ignore them. Insisting on this form for inputs and outputs is motivated by the desire to allow composition of several MapReduce processes.

The Map function takes an input element as its argument and produces zero or more key-value pairs. The types of keys and values are each arbitrary. Further, keys are not "keys" in the usual sense; they do not have to be unique. Rather a Map task can produce several key-value pairs with the same key, even from the same element.

Example 2.1: We shall illustrate a MapReduce computation with what has become the standard example application: counting the number of occurrences for each word in a collection of documents. In this example, the input file is a repository of documents, and each document is an element. The Map function for this example uses keys that are of type String (the words) and values that are integers. The Map task reads a document and breaks it into its sequence of words $w_1, w_2, \ldots, w_n$. It then emits a sequence of key-value pairs where the value is always 1. That is, the output of the Map task for this document is the sequence of key-value pairs:

$$(w_1, 1),\ (w_2, 1), \ldots, (w_n, 1)$$

Note that a single Map task will typically process many documents – all the documents in one or more chunks. Thus, its output will be more than the sequence for the one document suggested above. Note also that if a word w appears m times among all the documents assigned to that process, then there will be m key-value pairs $(w, 1)$ among its output. An option, which we discuss in Section 2.2.4, is to combine these m pairs into a single pair (w, m), but we can only do that because, as we shall see, the Reduce tasks apply an associative and commutative operation, addition, to the values. □

2.2.2 Grouping by Key

As soon as the Map tasks have all completed successfully, the key-value pairs are grouped by key, and the values associated with each key are formed into a list of values. The grouping is performed by the system, regardless of what the Map and Reduce tasks do. The master controller process knows how many Reduce tasks there will be, say r such tasks. The user typically tells the MapReduce system what r should be. Then the master controller picks a hash function that applies to keys and produces a bucket number from 0 to $r - 1$. Each key that is output by a Map task is hashed and its key-value pair is put in one of r local files. Each file is destined for one of the Reduce tasks.[1]

[1]Optionally, users can specify their own hash function or other method for assigning keys to Reduce tasks. However, whatever algorithm is used, each key is assigned to one and only one Reduce task.

To perform the grouping by key and distribution to the Reduce tasks, the master controller merges the files from each Map task that are destined for a particular Reduce task and feeds the merged file to that process as a sequence of key-list-of-value pairs. That is, for each key k, the input to the Reduce task that handles key k is a pair of the form $(k, [v_1, v_2, \ldots, v_n])$, where (k, v_1), $(k, v_2), \ldots, (k, v_n)$ are all the key-value pairs with key k coming from all the Map tasks.

2.2.3 The Reduce Tasks

The Reduce function's argument is a pair consisting of a key and its list of associated values. The output of the Reduce function is a sequence of zero or more key-value pairs. These key-value pairs can be of a type different from those sent from Map tasks to Reduce tasks, but often they are the same type. We shall refer to the application of the Reduce function to a single key and its associated list of values as a *reducer*.

A Reduce task receives one or more keys and their associated value lists. That is, a Reduce task executes one or more reducers. The outputs from all the Reduce tasks are merged into a single file. Reducers may be partitioned among a smaller number of Reduce tasks is by hashing the keys and associating each Reduce task with one of the buckets of the hash function.

Example 2.2: Let us continue with the word-count example of Example 2.1. The Reduce function simply adds up all the values. The output of a reducer consists of the word and the sum. Thus, the output of all the Reduce tasks is a sequence of (w, m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents. □

2.2.4 Combiners

Sometimes, a Reduce function is associative and commutative. That is, the values to be combined can be combined in any order, with the same result. The addition performed in Example 2.2 is an example of an associative and commutative operation. It doesn't matter how we group a list of numbers $v_1, v_2, \ldots, v_n$; the sum will be the same.

When the Reduce function is associative and commutative, we can push some of what the reducers do to the Map tasks. For example, instead of the Map tasks in Example 2.1 producing many pairs $(w, 1)$, $(w, 1), \ldots$, we could apply the Reduce function within the Map task, before the output of the Map tasks is subject to grouping and aggregation. These key-value pairs would thus be replaced by one pair with key w and value equal to the sum of all the 1's in all those pairs. That is, the pairs with key w generated by a single Map task would be replaced by a pair (w, m), where m is the number of times that w appears among the documents handled by this Map task. Note that it is still necessary to do grouping and aggregation and to pass the result to the Reduce

Reducers, Reduce Tasks, Compute Nodes, and Skew

If we want maximum parallelism, then we could use one Reduce task to execute each reducer, i.e., a single key and its associated value list. Further, we could execute each Reduce task at a different compute node, so they would all execute in parallel. This plan is not usually the best. One problem is that there is overhead associated with each task we create, so we might want to keep the number of Reduce tasks lower than the number of different keys. Moreover, often there are far more keys than there are compute nodes available, so we would get no benefit from a huge number of Reduce tasks.

Second, there is often significant variation in the lengths of the value lists for different keys, so different reducers take different amounts of time. If we make each reducer a separate Reduce task, then the tasks themselves will exhibit *skew* – a significant difference in the amount of time each takes. We can reduce the impact of skew by using fewer Reduce tasks than there are reducers. If keys are sent randomly to Reduce tasks, we can expect that there will be some averaging of the total time required by the different Reduce tasks. We can further reduce the skew by using more Reduce tasks than there are compute nodes. In that way, long Reduce tasks might occupy a compute node fully, while several shorter Reduce tasks might run sequentially at a single compute node.

tasks, since there will typically be one key-value pair with key w coming from each of the Map tasks.

2.2.5 Details of MapReduce Execution

Let us now consider in more detail how a program using MapReduce is executed. Figure 2.3 offers an outline of how processes, tasks, and files interact. Taking advantage of a library provided by a MapReduce system such as Hadoop, the user program forks a Master controller process and some number of Worker processes at different compute nodes. Normally, a Worker handles either Map tasks (a *Map worker*) or Reduce tasks (a *Reduce worker*), but not both.

The Master has many responsibilities. One is to create some number of Map tasks and some number of Reduce tasks, these numbers being selected by the user program. These tasks will be assigned to Worker processes by the Master. It is reasonable to create one Map task for every chunk of the input file(s), but we may wish to create fewer Reduce tasks. The reason for limiting the number of Reduce tasks is that it is necessary for each Map task to create an intermediate file for each Reduce task, and if there are too many Reduce tasks the number of intermediate files explodes.

The Master keeps track of the status of each Map and Reduce task (idle,

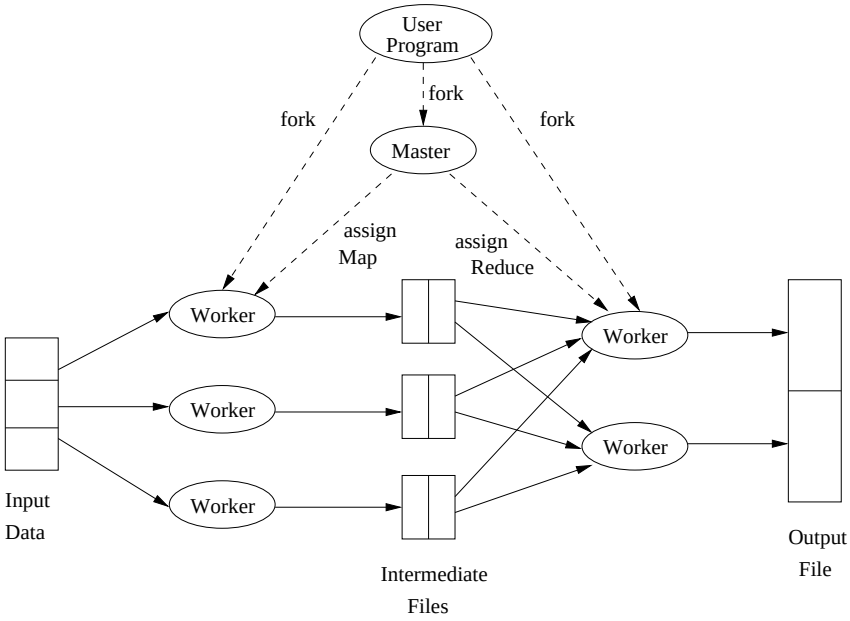

Figure 2.3: Overview of the execution of a MapReduce program

executing at a particular Worker, or completed). A Worker process reports to the Master when it finishes a task, and a new task is scheduled by the Master for that Worker process.

Each Map task is assigned one or more chunks of the input file(s) and executes on it the code written by the user. The Map task creates a file for each Reduce task on the local disk of the Worker that executes the Map task. The Master is informed of the location and sizes of each of these files, and the Reduce task for which each is destined. When a Reduce task is assigned by the Master to a Worker process, that task is given all the files that form its input. The Reduce task executes code written by the user and writes its output to a file that is part of the surrounding distributed file system.

2.2.6 Coping With Node Failures

The worst thing that can happen is that the compute node at which the Master is executing fails. In this case, the entire MapReduce job must be restarted. But only this one node can bring the entire process down; other failures will be managed by the Master, and the MapReduce job will complete eventually.

Suppose the compute node at which a Map worker resides fails. This failure will be detected by the Master, because it periodically pings the Worker processes. All the Map tasks that were assigned to this Worker will have to be redone, even if they had completed. The reason for redoing completed Map

tasks is that their output destined for the Reduce tasks resides at that compute node, and is now unavailable to the Reduce tasks. The Master sets the status of each of these Map tasks to idle and will schedule them on a Worker when one becomes available. The Master must also inform each Reduce task that the location of its input from that Map task has changed.

Dealing with a failure at the node of a Reduce worker is simpler. The Master simply sets the status of its currently executing Reduce tasks to idle. These will be rescheduled on another reduce worker later.

2.2.7 Exercises for Section 2.2

Exercise 2.2.1: Suppose we execute the word-count MapReduce program described in this section on a large repository such as a copy of the Web. We shall use 100 Map tasks and some number of Reduce tasks.

(a) Suppose we do not use a combiner at the Map tasks. Do you expect there to be significant skew in the times taken by the various reducers to process their value list? Why or why not?

(b) If we combine the reducers into a small number of Reduce tasks, say 10 tasks, at random, do you expect the skew to be significant? What if we instead combine the reducers into 10,000 Reduce tasks?

! (c) Suppose we do use a combiner at the 100 Map tasks. Do you expect skew to be significant? Why or why not?

2.3 Algorithms Using MapReduce

MapReduce is not a solution to every problem, not even every problem that profitably can use many compute nodes operating in parallel. As we mentioned in Section 2.1.2, the entire distributed-file-system milieu makes sense only when files are very large and are rarely updated in place. Thus, we would not expect to use either a DFS or an implementation of MapReduce for managing on-line retail sales, even though a large on-line retailer such as Amazon.com uses thousands of compute nodes when processing requests over the Web. The reason is that the principal operations on Amazon data involve responding to searches for products, recording sales, and so on, processes that involve relatively little calculation and that change the database.[2] On the other hand, Amazon might use MapReduce to perform certain analytic queries on large amounts of data, such as finding for each user those users whose buying patterns were most similar.

The original purpose for which the Google implementation of MapReduce was created was to execute very large matrix-vector multiplications as are

[2]Remember that even looking at a product you don't buy causes Amazon to remember that you looked at it.