Mining of Massive Datasets

Jure Leskovec
Stanford Univ.

Anand Rajaraman
Milliway Labs

Jeffrey D. Ullman
Stanford Univ.

Preface

This book evolved from material developed over several years by Anand Rajaraman and Jeff Ullman for a one-quarter course at Stanford. The course CS345A, titled "Web Mining," was designed as an advanced graduate course, although it has become accessible and interesting to advanced undergraduates. When Jure Leskovec joined the Stanford faculty, we reorganized the material considerably. He introduced a new course CS224W on network analysis and added material to CS345A, which was renumbered CS246. The three authors also introduced a large-scale data-mining project course, CS341. The book now contains material taught in all three courses.

What the Book Is About

At the highest level of description, this book is about data mining. However, it focuses on data mining of very large amounts of data, that is, data so large it does not fit in main memory. Because of the emphasis on size, many of our examples are about the Web or data derived from the Web. Further, the book takes an algorithmic point of view: data mining is about applying algorithms to data, rather than using data to "train" a machine-learning engine of some sort. The principal topics covered are:

1. Distributed file systems and map-reduce as a tool for creating parallel algorithms that succeed on very large amounts of data.
2. Similarity search, including the key techniques of minhashing and locality-sensitive hashing.
3. Data-stream processing and specialized algorithms for dealing with data that arrives so fast it must be processed immediately or lost.
4. The technology of search engines, including Google's PageRank, link-spam detection, and the hubs-and-authorities approach.
5. Frequent-itemset mining, including association rules, market-baskets, the A-Priori Algorithm and its improvements.
6. Algorithms for clustering very large, high-dimensional datasets.

7. Two key problems for Web applications: managing advertising and recommendation systems.
8. Algorithms for analyzing and mining the structure of very large graphs, especially social-network graphs.
9. Techniques for obtaining the important properties of a large dataset by dimensionality reduction, including singular-value decomposition and latent semantic indexing.
10. Machine-learning algorithms that can be applied to very large data, such as perceptrons, support-vector machines, and gradient descent.

Prerequisites

To appreciate fully the material in this book, we recommend the following prerequisites:

1. An introduction to database systems, covering SQL and related programming systems.
2. A sophomore-level course in data structures, algorithms, and discrete math.
3. A sophomore-level course in software systems, software engineering, and programming languages.

Exercises

The book contains extensive exercises, with some for almost every section. We indicate harder exercises or parts of exercises with an exclamation point. The hardest exercises have a double exclamation point.

Support on the Web

Go to `http://www.mmds.org` for slides, homework assignments, project requirements, and exams from courses related to this book.

Gradiance Automated Homework

There are automated exercises based on this book, using the Gradiance root-question technology, available at `www.gradiance.com/services`. Students may enter a public class by creating an account at that site and entering the class with code `1EDD8A1D`. Instructors may use the site by making an account there

and then emailing `support at gradiance dot com` with their login name, the name of their school, and a request to use the MMDS materials.

Acknowledgements

Cover art is by Scott Ullman.

We would like to thank Foto Afrati, Arun Marathe, and Rok Sosic for critical readings of a draft of this manuscript.

Errors were also reported by Rajiv Abraham, Ruslan Aduk, Apoorv Agarwal, Aris Anagnostopoulos, Yokila Arora, Stefanie Anna Baby, Atilla Soner Balkir, Arnaud Belletoile, Robin Bennett, Susan Biancani, Richard Boyd, Amitabh Chaudhary, Leland Chen, Hua Feng, Marcus Gemeinder, Anastasios Gounaris, Clark Grubb, Shrey Gupta, Waleed Hameid, Saman Haratizadeh, Julien Hoachuck, Przemyslaw Horban, Hsiu-Hsuan Huang, Jeff Hwang, Rafi Kamal, Lachlan Kang, Ed Knorr, Haewoon Kwak, Ellis Lau, Greg Lee, David Z. Liu, Ethan Lozano, Yunan Luo, Michael Mahoney, Sergio Matos, Justin Meyer, Bryant Moscon, Brad Penoff, John Phillips, Philips Kokoh Prasetyo, Qi Ge, Harizo Rajaona, Timon Ruban, Rich Seiter, Hitesh Shetty, Angad Singh, Sandeep Sripada, Dennis Sidharta, Krzysztof Stencel, Mark Storus, Roshan Sumbaly, Zack Taylor, Tim Triche Jr., Wang Bin, Weng Zhen-Bin, Robert West, Steven Euijong Whang, Oscar Wu, Xie Ke, Christopher T.-R. Yeh, Nicolas Zhao, and Zhou Jingbo, The remaining errors are ours, of course.

J. L.
A. R.
J. D. U.
Palo Alto, CA
March, 2014

Contents

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.

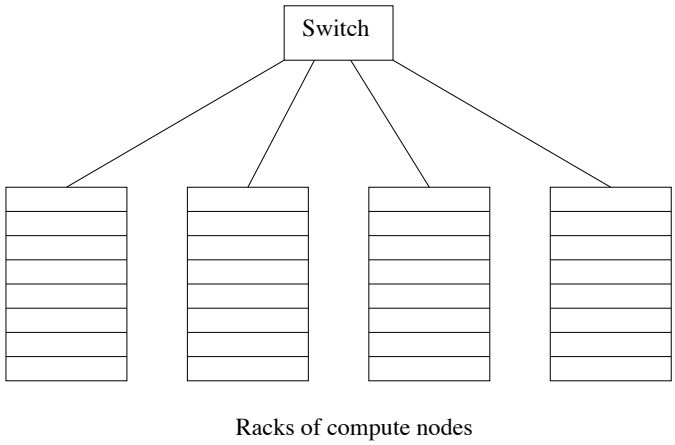

Figure 2.1: Compute nodes are organized into racks, and racks are interconnected by a switch

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.

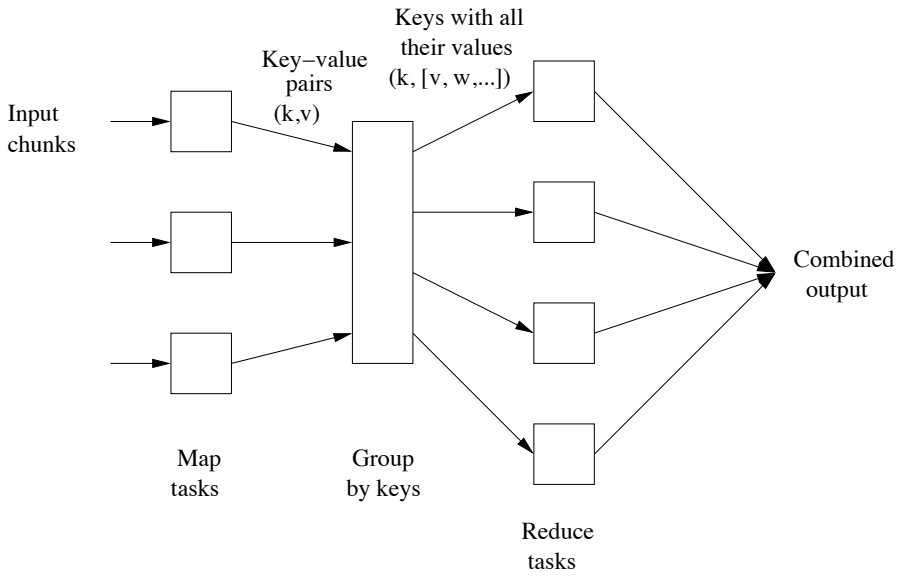

Figure 2.2: Schematic of a MapReduce computation

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,

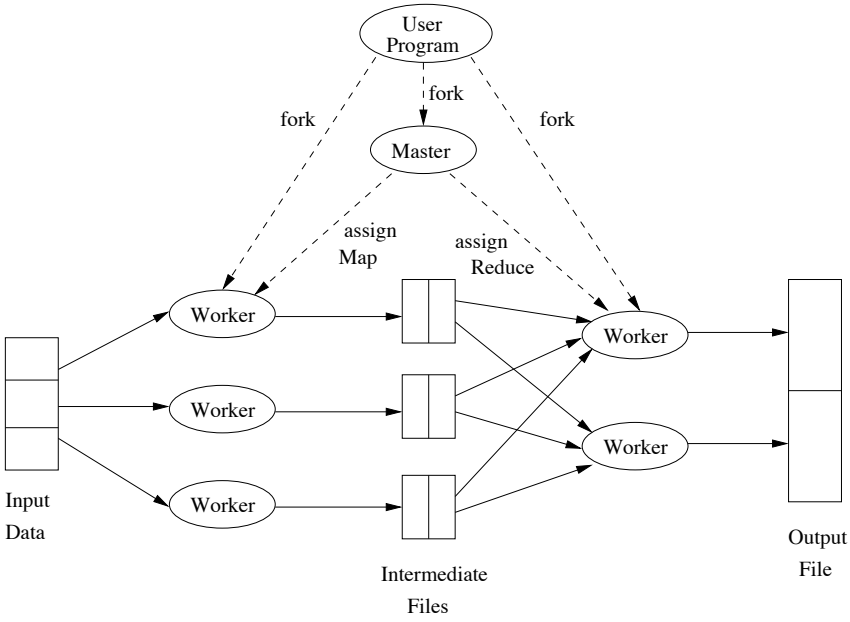

Figure 2.3: Overview of the execution of a MapReduce program

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.

needed in the calculation of PageRank (See Chapter 5). We shall see that matrix-vector and matrix-matrix calculations fit nicely into the MapReduce style of computing. Another important class of operations that can use MapReduce effectively are the relational-algebra operations. We shall examine the MapReduce execution of these operations as well.

2.3.1 Matrix-Vector Multiplication by MapReduce

Suppose we have an $n \times n$ matrix M, whose element in row i and column j will be denoted m_{ij}. Suppose we also have a vector $\mathbf{v}$ of length n, whose jth element is v_j. Then the matrix-vector product is the vector $\mathbf{x}$ of length n, whose ith element x_i is given by

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

If $n = 100$, we do not want to use a DFS or MapReduce for this calculation. But this sort of calculation is at the heart of the ranking of Web pages that goes on at search engines, and there, n is in the tens of billions.[3] Let us first assume that n is large, but not so large that vector $\mathbf{v}$ cannot fit in main memory and thus be available to every Map task.

The matrix M and the vector $\mathbf{v}$ each will be stored in a file of the DFS. We assume that the row-column coordinates of each matrix element will be discoverable, either from its position in the file, or because it is stored with explicit coordinates, as a triple (i, j, m_{ij}). We also assume the position of element v_j in the vector $\mathbf{v}$ will be discoverable in the analogous way.

The Map Function: The Map function is written to apply to one element of M. However, if $\mathbf{v}$ is not already read into main memory at the compute node executing a Map task, then $\mathbf{v}$ is first read, in its entirety, and subsequently will be available to all applications of the Map function performed at this Map task. Each Map task will operate on a chunk of the matrix M. From each matrix element m_{ij} it produces the key-value pair $(i, m_{ij}v_j)$. Thus, all terms of the sum that make up the component x_i of the matrix-vector product will get the same key, i.

The Reduce Function: The Reduce function simply sums all the values associated with a given key i. The result will be a pair (i, x_i).

2.3.2 If the Vector v Cannot Fit in Main Memory

However, it is possible that the vector $\mathbf{v}$ is so large that it will not fit in its entirety in main memory. It is not required that $\mathbf{v}$ fit in main memory at a compute node, but if it does not then there will be a very large number of

[3]The matrix is sparse, with on the average of 10 to 15 nonzero elements per row, since the matrix represents the links in the Web, with m_{ij} nonzero if and only if there is a link from page j to page i. Note that there is no way we could store a dense matrix whose side was 10^{10}, since it would have 10^{20} elements.

disk accesses as we move pieces of the vector into main memory to multiply components by elements of the matrix. Thus, as an alternative, we can divide the matrix into vertical *stripes* of equal width and divide the vector into an equal number of horizontal stripes, of the same height. Our goal is to use enough stripes so that the portion of the vector in one stripe can fit conveniently into main memory at a compute node. Figure 2.4 suggests what the partition looks like if the matrix and vector are each divided into five stripes.

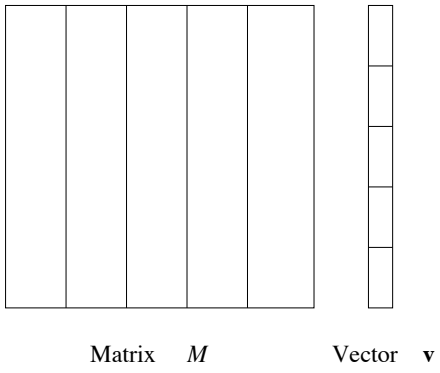

Figure 2.4: Division of a matrix and vector into five stripes

The ith stripe of the matrix multiplies only components from the ith stripe of the vector. Thus, we can divide the matrix into one file for each stripe, and do the same for the vector. Each Map task is assigned a chunk from one of the stripes of the matrix and gets the entire corresponding stripe of the vector. The Map and Reduce tasks can then act exactly as was described above for the case where Map tasks get the entire vector.

We shall take up matrix-vector multiplication using MapReduce again in Section 5.2. There, because of the particular application (PageRank calculation), we have an additional constraint that the result vector should be partitioned in the same way as the input vector, so the output may become the input for another iteration of the matrix-vector multiplication. We shall see there that the best strategy involves partitioning the matrix M into square blocks, rather than stripes.

2.3.3 Relational-Algebra Operations

There are a number of operations on large-scale data that are used in database queries. Many traditional database applications involve retrieval of small amounts of data, even though the database itself may be large. For example, a query may ask for the bank balance of one particular account. Such queries are not useful applications of MapReduce.

However, there are many operations on data that can be described easily in terms of the common database-query primitives, even if the queries themselves

are not executed within a database management system. Thus, a good starting point for exploring applications of MapReduce is by considering the standard operations on relations. We assume you are familiar with database systems, the query language SQL, and the relational model, but to review, a *relation* is a table with column headers called *attributes*. Rows of the relation are called *tuples*. The set of attributes of a relation is called its *schema*. We often write an expression like $R(A_1, A_2, \ldots, A_n)$ to say that the relation name is R and its attributes are A_1, $A_2, \ldots, A_n$.

From	*To*
url1	url2
url1	url3
url2	url3
url2	url4
...	...

Figure 2.5: Relation *Links* consists of the set of pairs of URL's, such that the first has one or more links to the second

Example 2.3: In Fig. 2.5 we see part of the relation *Links* that describes the structure of the Web. There are two attributes, *From* and *To*. A row, or tuple, of the relation is a pair of URL's, such that there is at least one link from the first URL to the second. For instance, the first row of Fig. 2.5 is the pair $(url1, url2)$ that says the Web page $url1$ has a link to page $url2$. While we have shown only four tuples, the real relation of the Web, or the portion of it that would be stored by a typical search engine, has billions of tuples. □

A relation, however large, can be stored as a file in a distributed file system. The elements of this file are the tuples of the relation.

There are several standard operations on relations, often referred to as *relational algebra*, that are used to implement queries. The queries themselves usually are written in SQL. The relational-algebra operations we shall discuss are:

1. *Selection*: Apply a condition C to each tuple in the relation and produce as output only those tuples that satisfy C. The result of this selection is denoted $\sigma_C(R)$.

2. *Projection*: For some subset S of the attributes of the relation, produce from each tuple only the components for the attributes in S. The result of this projection is denoted $\pi_S(R)$.

3. *Union, Intersection*, and *Difference*: These well-known set operations apply to the sets of tuples in two relations that have the same schema. There are also bag (multiset) versions of the operations in SQL, with

somewhat unintuitive definitions, but we shall not go into the bag versions of these operations here.

4. *Natural Join*: Given two relations, compare each pair of tuples, one from each relation. If the tuples agree on all the attributes that are common to the two schemas, then produce a tuple that has components for each of the attributes in either schema and agrees with the two tuples on each attribute. If the tuples disagree on one or more shared attributes, then produce nothing from this pair of tuples. The natural join of relations R and S is denoted $R \bowtie S$. While we shall discuss executing only the natural join with MapReduce, all *equijoins* (joins where the tuple-agreement condition involves equality of attributes from the two relations that do not necessarily have the same name) can be executed in the same manner. We shall give an illustration in Example 2.4.

5. *Grouping* and *Aggregation*:[4] Given a relation R, partition its tuples according to their values in one set of attributes G, called the *grouping attributes*. Then, for each group, aggregate the values in certain other attributes. The normally permitted aggregations are SUM, COUNT, AVG, MIN, and MAX, with the obvious meanings. Note that MIN and MAX require that the aggregated attributes have a type that can be compared, e.g., numbers or strings, while SUM and AVG require that the type allow arithmetic operations. We denote a grouping-and-aggregation operation on a relation R by $\gamma_X(R)$, where X is a list of elements that are either

   (a) A grouping attribute, or

   (b) An expression $\theta(A)$, where θ is one of the five aggregation operations such as SUM, and A is an attribute not among the grouping attributes.

   The result of this operation is one tuple for each group. That tuple has a component for each of the grouping attributes, with the value common to tuples of that group. It also has a component for each aggregation, with the aggregated value for that group. We shall see an illustration in Example 2.5.

Example 2.4: Let us try to find the paths of length two in the Web, using the relation *Links* of Fig. 2.5. That is, we want to find the triples of URL's (u, v, w) such that there is a link from u to v and a link from v to w. We essentially want to take the natural join of *Links* with itself, but we first need to imagine that it is two relations, with different schemas, so we can describe the desired connection as a natural join. Thus, imagine that there are two copies of *Links*, namely $L1(U1, U2)$ and $L2(U2, U3)$. Now, if we compute $L1 \bowtie L2$,

[4]Some descriptions of relational algebra do not include these operations, and indeed they were not part of the original definition of this algebra. However, these operations are so important in SQL, that modern treatments of relational algebra include them.

we shall have exactly what we want. That is, for each tuple $t1$ of $L1$ (i.e., each tuple of *Links*) and each tuple $t2$ of $L2$ (another tuple of *Links*, possibly even the same tuple), see if their $U2$ components are the same. Note that these components are the second component of $t1$ and the first component of $t2$. If these two components agree, then produce a tuple for the result, with schema $(U1, U2, U3)$. This tuple consists of the first component of $t1$, the second component of $t1$ (which must equal the first component of $t2$), and the second component of $t2$.

We may not want the entire path of length two, but only want the pairs (u, w) of URL's such that there is at least one path from u to w of length two. If so, we can project out the middle components by computing $\pi_{U1,U3}(L1 \bowtie L2)$. □

Example 2.5: Imagine that a social-networking site has a relation

Friends(User, Friend)

This relation has tuples that are pairs (a, b) such that b is a friend of a. The site might want to develop statistics about the number of friends members have. Their first step would be to compute a count of the number of friends of each user. This operation can be done by grouping and aggregation, specifically

$$\gamma_{\text{User,COUNT(Friend)}}(\text{Friends})$$

This operation groups all the tuples by the value in their first component, so there is one group for each user. Then, for each group the count of the number of friends of that user is made. The result will be one tuple for each group, and a typical tuple would look like $(\text{Sally}, 300)$, if user "Sally" has 300 friends. □

2.3.4 Computing Selections by MapReduce

Selections really do not need the full power of MapReduce. They can be done most conveniently in the map portion alone, although they could also be done in the reduce portion alone. Here is a MapReduce implementation of selection $\sigma_C(R)$.

The Map Function: For each tuple t in R, test if it satisfies C. If so, produce the key-value pair (t, t). That is, both the key and value are t.

The Reduce Function: The Reduce function is the identity. It simply passes each key-value pair to the output.

Note that the output is not exactly a relation, because it has key-value pairs. However, a relation can be obtained by using only the value components (or only the key components) of the output.

2.3.5 Computing Projections by MapReduce

Projection is performed similarly to selection, because projection may cause the same tuple to appear several times, the Reduce function must eliminate duplicates. We may compute $\pi_S(R)$ as follows.

The Map Function: For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in S. Output the key-value pair (t', t').

The Reduce Function: For each key t' produced by any of the Map tasks, there will be one or more key-value pairs (t', t'). The Reduce function turns $(t', [t', t', \ldots, t'])$ into (t', t'), so it produces exactly one pair (t', t') for this key t'.

Observe that the Reduce operation is duplicate elimination. This operation is associative and commutative, so a combiner associated with each Map task can eliminate whatever duplicates are produced locally. However, the Reduce tasks are still needed to eliminate two identical tuples coming from different Map tasks.

2.3.6 Union, Intersection, and Difference by MapReduce

First, consider the union of two relations. Suppose relations R and S have the same schema. Map tasks will be assigned chunks from either R or S; it doesn't matter which. The Map tasks don't really do anything except pass their input tuples as key-value pairs to the Reduce tasks. The latter need only eliminate duplicates as for projection.

The Map Function: Turn each input tuple t into a key-value pair (t, t).

The Reduce Function: Associated with each key t there will be either one or two values. Produce output (t, t) in either case.

To compute the intersection, we can use the same Map function. However, the Reduce function must produce a tuple only if both relations have the tuple. If the key t has a list of two values $[t, t]$ associated with it, then the Reduce task for t should produce (t, t). However, if the value-list associated with key t is just $[t]$, then one of R and S is missing t, so we don't want to produce a tuple for the intersection.

The Map Function: Turn each tuple t into a key-value pair (t, t).

The Reduce Function: If key t has value list $[t, t]$, then produce (t, t). Otherwise, produce nothing.

The Difference $R - S$ requires a bit more thought. The only way a tuple t can appear in the output is if it is in R but not in S. The Map function can pass tuples from R and S through, but must inform the Reduce function whether the tuple came from R or S. We shall thus use the relation as the value associated with the key t. Here is a specification for the two functions.

The Map Function: For a tuple t in R, produce key-value pair (t, R), and for a tuple t in S, produce key-value pair (t, S). Note that the intent is that the value is the name of R or S (or better, a single bit indicating whether the relation is R or S), not the entire relation.

The Reduce Function: For each key t, if the associated value list is $[R]$, then produce (t, t). Otherwise, produce nothing.

2.3.7 Computing Natural Join by MapReduce

The idea behind implementing natural join via MapReduce can be seen if we look at the specific case of joining $R(A, B)$ with $S(B, C)$. We must find tuples that agree on their B components, that is the second component from tuples of R and the first component of tuples of S. We shall use the B-value of tuples from either relation as the key. The value will be the other component and the name of the relation, so the Reduce function can know where each tuple came from.

The Map Function: For each tuple (a, b) of R, produce the key-value pair $\big(b, (R, a)\big)$. For each tuple (b, c) of S, produce the key-value pair $\big(b, (S, c)\big)$.

The Reduce Function: Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c). Construct all pairs consisting of one with first component R and the other with first component S, say (R, a) and (S, c). The output from this key and value list is a sequence of key-value pairs. The key is irrelevant. Each value is one of the triples (a, b, c) such that (R, a) and (S, c) are on the input list of values.

The same algorithm works if the relations have more than two attributes. You can think of A as representing all those attributes in the schema of R but not S. B represents the attributes in both schemas, and C represents attributes only in the schema of S. The key for a tuple of R or S is the list of values in all the attributes that are in the schemas of both R and S. The value for a tuple of R is the name R together with the values of all the attributes belonging to R but not to S, and the value for a tuple of S is the name S together with the values of the attributes belonging to S but not R.

The Reduce function looks at all the key-value pairs with a given key and combines those values from R with those values of S in all possible ways. From each pairing, the tuple produced has the values from R, the key values, and the values from S.

2.3.8 Grouping and Aggregation by MapReduce

As with the join, we shall discuss the minimal example of grouping and aggregation, where there is one grouping attribute and one aggregation. Let $R(A, B, C)$ be a relation to which we apply the operator $\gamma_{A,\theta(B)}(R)$. Map will perform the grouping, while Reduce does the aggregation.

The Map Function: For each tuple (a, b, c) produce the key-value pair (a, b).

The Reduce Function: Each key a represents a group. Apply the aggregation operator θ to the list $[b_1, b_2, \ldots, b_n]$ of B-values associated with key a. The output is the pair (a, x), where x is the result of applying θ to the list. For example, if θ is SUM, then $x = b_1 + b_2 + \cdots + b_n$, and if θ is MAX, then x is the largest of b_1, $b_2, \ldots, b_n$.

If there are several grouping attributes, then the key is the list of the values of a tuple for all these attributes. If there is more than one aggregation, then the Reduce function applies each of them to the list of values associated with a given key and produces a tuple consisting of the key, including components for all grouping attributes if there is more than one, followed by the results of each of the aggregations.

2.3.9 Matrix Multiplication

If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product $P = MN$ is the matrix P with element p_{ik} in row i and column k, where

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

It is required that the number of columns of M equals the number of rows of N, so the sum over j makes sense.

We can think of a matrix as a relation with three attributes: the row number, the column number, and the value in that row and column. Thus, we could view matrix M as a relation $M(I, J, V)$, with tuples (i, j, m_{ij}), and we could view matrix N as a relation $N(J, K, W)$, with tuples (j, k, n_{jk}). As large matrices are often sparse (mostly 0's), and since we can omit the tuples for matrix elements that are 0, this relational representation is often a very good one for a large matrix. However, it is possible that i, j, and k are implicit in the position of a matrix element in the file that represents it, rather than written explicitly with the element itself. In that case, the Map function will have to be designed to construct the I, J, and K components of tuples from the position of the data.

The product MN is almost a natural join followed by grouping and aggregation. That is, the natural join of $M(I, J, V)$ and $N(J, K, W)$, having only attribute J in common, would produce tuples (i, j, k, v, w) from each tuple (i, j, v) in M and tuple (j, k, w) in N. This five-component tuple represents the pair of matrix elements (m_{ij}, n_{jk}). What we want instead is the product of these elements, that is, the four-component tuple $(i, j, k, v \times w)$, because that represents the product $m_{ij} n_{jk}$. Once we have this relation as the result of one MapReduce operation, we can perform grouping and aggregation, with I and K as the grouping attributes and the sum of $V \times W$ as the aggregation. That is, we can implement matrix multiplication as the cascade of two MapReduce operations, as follows. First:

The Map Function: For each matrix element m_{ij}, produce the key value pair $\big(j, (M, i, m_{ij})\big)$. Likewise, for each matrix element n_{jk}, produce the key value

pair $\big(j,(N,k,n_{jk})\big)$. Note that M and N in the values are not the matrices themselves. Rather they are names of the matrices or (as we mentioned for the similar Map function used for natural join) better, a bit indicating whether the element comes from M or N.

The Reduce Function: For each key j, examine its list of associated values. For each value that comes from M, say (M,i,m_{ij}), and each value that comes from N, say (N,k,n_{jk}), produce a key-value pair with key equal to (i,k) and value equal to the product of these elements, $m_{ij}n_{jk}$.

Now, we perform a grouping and aggregation by another MapReduce operation.

The Map Function: This function is just the identity. That is, for every input element with key (i,k) and value v, produce exactly this key-value pair.

The Reduce Function: For each key (i,k), produce the sum of the list of values associated with this key. The result is a pair $\big((i,k),v\big)$, where v is the value of the element in row i and column k of the matrix $P = MN$.

2.3.10 Matrix Multiplication with One MapReduce Step

There often is more than one way to use MapReduce to solve a problem. You may wish to use only a single MapReduce pass to perform matrix multiplication $P = MN$. [5] It is possible to do so if we put more work into the two functions. Start by using the Map function to create the sets of matrix elements that are needed to compute each element of the answer P. Notice that an element of M or N contributes to many elements of the result, so one input element will be turned into many key-value pairs. The keys will be pairs (i,k), where i is a row of M and k is a column of N. Here is a synopsis of the Map and Reduce functions.

The Map Function: For each element m_{ij} of M, produce all the key-value pairs $\big((i,k),\ (M,j,m_{ij})\big)$ for $k = 1,2,\ldots$, up to the number of columns of N. Similarly, for each element n_{jk} of N, produce all the key-value pairs $\big((i,k),\ (N,j,n_{jk})\big)$ for $i = 1,2,\ldots$, up to the number of rows of M. As before, M and N are really bits to tell which of the two matrices a value comes from.

The Reduce Function: Each key (i,k) will have an associated list with all the values (M,j,m_{ij}) and (N,j,n_{jk}), for all possible values of j. The Reduce function needs to connect the two values on the list that have the same value of j, for each j. An easy way to do this step is to sort by j the values that begin with M and sort by j the values that begin with N, in separate lists. The jth values on each list must have their third components, m_{ij} and n_{jk} extracted and multiplied. Then, these products are summed and the result is paired with (i,k) in the output of the Reduce function.

[5] However, we show in Section 2.6.7 that two passes of MapReduce are usually better than one for matrix multiplication.

You may notice that if a row of the matrix M or a column of the matrix N is so large that it will not fit in main memory, then the Reduce tasks will be forced to use an external sort to order the values associated with a given key (i, k). However, in that case, the matrices themselves are so large, perhaps 10^{20} elements, that it is unlikely we would attempt this calculation if the matrices were dense. If they are sparse, then we would expect many fewer values to be associated with any one key, and it would be feasible to do the sum of products in main memory.

2.3.11 Exercises for Section 2.3

Exercise 2.3.1: Design MapReduce algorithms to take a very large file of integers and produce as output:

(a) The largest integer.

(b) The average of all the integers.

(c) The same set of integers, but with each integer appearing only once.

(d) The count of the number of distinct integers in the input.

Exercise 2.3.2: Our formulation of matrix-vector multiplication assumed that the matrix M was square. Generalize the algorithm to the case where M is an r-by-c matrix for some number of rows r and columns c.

! **Exercise 2.3.3:** In the form of relational algebra implemented in SQL, relations are not sets, but bags; that is, tuples are allowed to appear more than once. There are extended definitions of union, intersection, and difference for bags, which we shall define below. Write MapReduce algorithms for computing the following operations on bags R and S:

(a) *Bag Union*, defined to be the bag of tuples in which tuple t appears the sum of the numbers of times it appears in R and S.

(b) *Bag Intersection*, defined to be the bag of tuples in which tuple t appears the minimum of the numbers of times it appears in R and S.

(c) *Bag Difference*, defined to be the bag of tuples in which the number of times a tuple t appears is equal to the number of times it appears in R minus the number of times it appears in S. A tuple that appears more times in S than in R does not appear in the difference.

! **Exercise 2.3.4:** Selection can also be performed on bags. Give a MapReduce implementation that produces the proper number of copies of each tuple t that passes the selection condition. That is, produce key-value pairs from which the correct result of the selection can be obtained easily from the values.