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. This application, called "PageRank," is the subject of Chapter 5.

2. Searches in "friends" networks at social-networking sites, which involve graphs with hundreds of millions of nodes and many billions of edges. Operations on graphs of this type are covered in Chapter 10.

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.

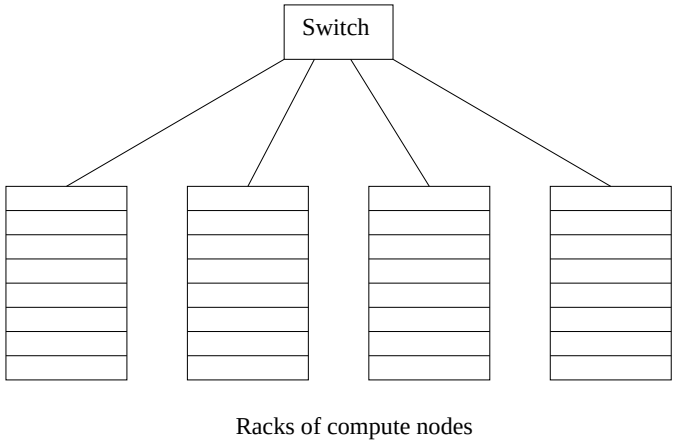

Figure 2.1: Compute nodes are organized into racks, and racks are interconnected by a switch

It is a fact of life that components fail, and the more components, such as compute nodes and communication links, 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.

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. *Colossus* is an improved version of GFS, about which little has been published. However, a goal of Colossus is to provide real-time file service.

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.

- 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. Typically, a rack "fails" because the interconnect among the compute nodes on the rack fails, and the rack can no longer communicate with anything outside itself. 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, parallel computations in a way that is tolerant of hardware faults. All you need to write are two functions, called *Map* and *Reduce*. The system manages the parallel execution and coordination of tasks that execute Map or Reduce. The system 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.

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

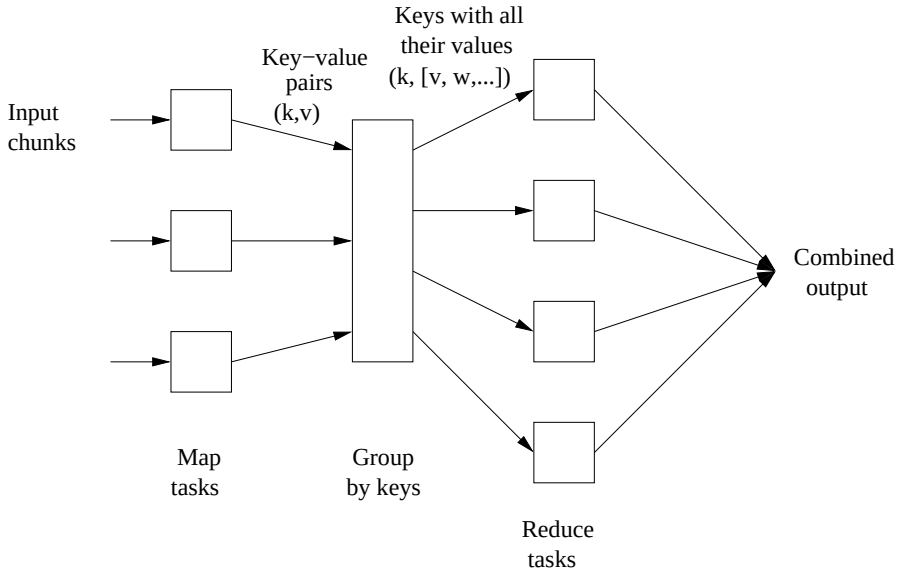

Figure 2.2: Schematic of a MapReduce computation

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 task, then there will be m key-value pairs $(w, 1)$ among its output. An option, which we discuss in Section 2.2.4, is for this Map task 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 single list of values for that key. 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 takes a key as argument 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 r Reduce tasks.[1]

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-values 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.

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 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 order or 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 each

[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.

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.

Map task in Example 2.1 producing many pairs $(w, 1)$, $(w, 1), \ldots$, we could apply the Reduce function within each Map task, before the outputs of the Map tasks are 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 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

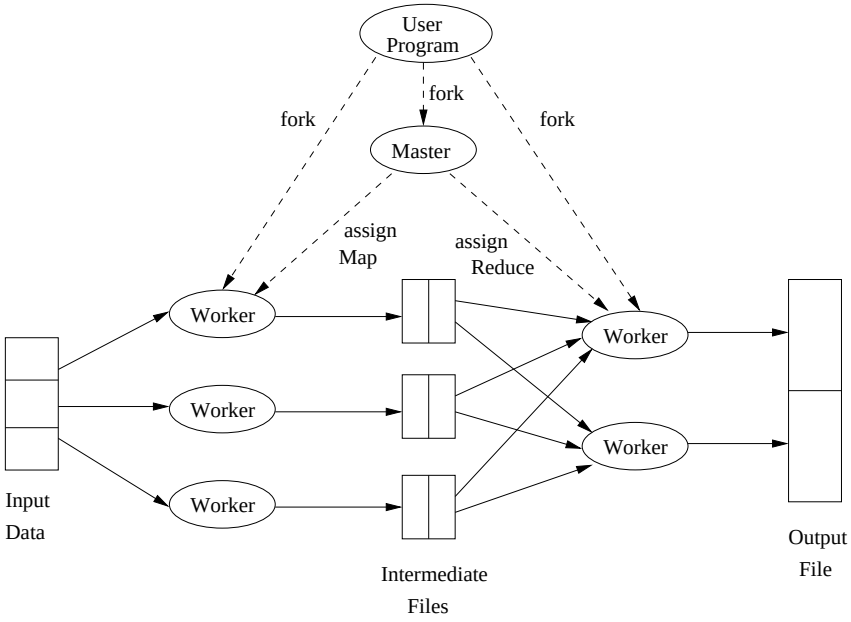

Figure 2.3: Overview of the execution of a MapReduce program

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, 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. For instance, 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 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 is 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 on the order of trillions.[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.

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

[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^{12}, since it would have 10^{24} elements.

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 **v** is so large that it will not fit in its entirety in main memory. It is not required that **v** fit in main memory at a compute node, but if it does not then there will be a very large number of 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.

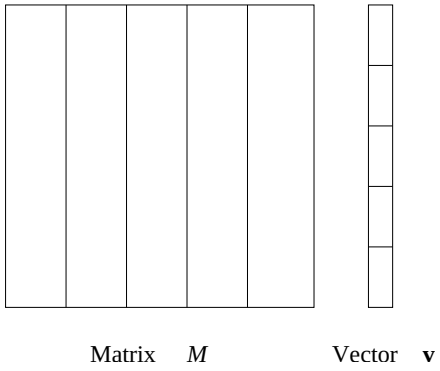

Figure 2.4: Division of a matrix and vector into five stripes

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 in Section 2.3.1 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 $(url1, url2)$. This tuple 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 trillions 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, typically only numbers. COUNT can be performed on data of any type. We denote a grouping-and-aggregation operation on a relation R by $\gamma_X(R)$, where X is a list of elements that are each 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.

[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.

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 $L1 \bowtie L2$, 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. If t does not satisfy C, then the mapper for t produces nothing.

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. However, 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'). After the system groups key-value pairs by key, 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 $S(B, C)$.[5] 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 for key b.

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.

[5]If you are familiar with database implementation, you will recognize the MapReduce implementation of join as the classic parallel hash join.

2.3.8 Grouping and Aggregation by MapReduce

As we did for the join, we shall discuss here only the minimal example of grouping and aggregation, where there is one grouping attribute (A), one aggregated attribute (B), and one attribute (C) that is neither grouped nor aggregated. 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, more precisely, a single bit that indicates whether the element comes from M or N (as we mentioned regarding the similar Map function we used for the natural join):.

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 applied to the output of the first 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 $P = MN$.[6] 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.

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

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.

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.

In each part, you can assume that the key of each output pair will be ignored or dropped.

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.

Exercise 2.3.5: The relational-algebra operation $R(A, B) \bowtie_{B<C} S(C, D)$ produces all tuples (a, b, c, d) such that tuple (a, b) is in relation R, tuple (c, d) is in S, and $b < c$. Give a MapReduce implementation of this operation, assuming R and S are sets.

! **Exercise 2.3.6:** In Section 2.3.5 we claimed that duplicate elimination is an associative and commutative operation. Prove this fact.

2.4 Extensions to MapReduce

MapReduce proved so influential that it spawned a number of extensions and modifications. These systems typically share a number of characteristics with MapReduce systems:

1. They are built on a distributed file system.
2. They manage very large numbers of tasks that are instantiations of a small number of user-written functions.
3. They incorporate a method for dealing with most of the failures that occur during the execution of a large job, without having to restart that job from the beginning.

We begin this section with a discussion of "workflow" systems, which extend MapReduce by supporting acyclic networks of functions, each function implemented by a collection of tasks. While many such systems have been implemented (see the bibliographic notes for this chapter), an increasingly popular choice is UC Berkeley's Spark. Also gaining in importance is Google's TensorFlow. The latter, while not generally recognized as a workflow system because of its very specific targeting of machine-learning applications, in fact has a workflow architecture at heart.

Another family of systems uses a graph model of data. Computation occurs at the nodes of the graph, and messages are sent from any node to any adjacent node. The original system of this type was Google's Pregel, which has its own unique way of dealing with failures. But it has now become common to implement a graph-model facility on top of a workflow system and use the latter's file system and failure-management facility.

2.4.1 Workflow Systems

Workflow systems extend MapReduce from the simple two-step workflow (the Map function feeds the Reduce function) to any collection of functions, with an acyclic graph representing workflow among the functions. That is, there is an acyclic *flow graph* whose arcs $a \to b$ represent the fact that function a's output is an input to function b.

The data passed from one function to the next is a file of elements of one type. If a function has a single input, then that function is applied to each input independently, just as Map and Reduce functions are applied to their input elements individually. The output of the function is a file collected from the result of applying the function to each input. If a function has inputs from more than one file, elements from each of the files can be combined in various ways. But the function itself is applied to combinations of input elements, at most one from each input file. We shall see examples of such combinations when we discuss the implementation of union and the relational join in Section 2.4.2.

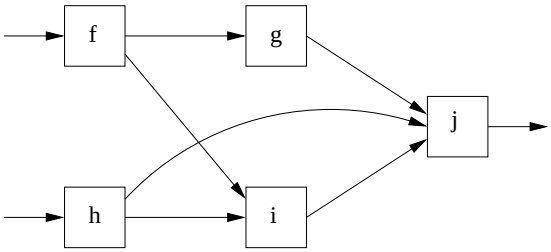

Figure 2.6: An example of a workflow that is more complex than Map feeding Reduce

Example 2.6 : A suggestion of what a workflow might look like is in Fig. 2.6. There, five functions, f through j, pass data from left to right in specific ways, so the flow of data is acyclic and no task needs to provide data out before its entire input is available. For instance, function h takes its input from a preexisting file of the distributed file system. Each of h's output elements is passed to the functions i and j, while i takes the outputs of both f and h as inputs. The output of j is either stored in the distributed file system or is passed to an application that invoked this dataflow. □

In analogy to Map and Reduce functions, each function of a workflow can be executed by many tasks, each of which is assigned a portion of the input to the function. A master controller is responsible for dividing the work among the tasks that implement a function, possibly by hashing the input elements to decide on the proper task to receive an element. Thus, like Map tasks, each task implementing a function f has an output file of data destined for each of the tasks that implement the successor function(s) of f. These files are delivered

by the master controller at the appropriate time – after the task has completed its work.

The functions of a workflow, and therefore the tasks, share with MapReduce tasks an important property: the *blocking property*, in that they only deliver output after they complete. As a result, if a task fails, it has not delivered output to any of its successors in the flow graph.[7] A master controller can therefore restart the failed task at another compute node, without worrying that the output of the restarted task will duplicate output that previously was passed to some other task.

Some applications of workflow systems are effectively cascades of MapReduce jobs. An example would be the join of three relations, where one MapReduce job joins the first two relations, and a second MapReduce job joins the third relation with the result of joining the first two relations. Both jobs would use an algorithm like that of Section 2.3.7.

There is an advantage to implementing such cascades as a single workflow. For example, the flow of data among tasks, and its replication, can be managed by the master controller, without need to store the temporary file that is output of one MapReduce job in the distributed file system. By locating tasks at compute nodes that have a copy of their input, we can avoid much of the communication that would be necessary if we stored the result of one MapReduce job and then initiated a second MapReduce job (although Hadoop and other MapReduce systems also try to locate Map tasks where a copy of their input is already present).

2.4.2 Spark

Spark is, at its heart, a workflow system. However, it is an advance over the early workflow systems in several ways, including:

1. A more efficient way of coping with failures.
2. A more efficient way of grouping tasks among compute nodes and scheduling execution of functions.
3. Integration of programming language features such as looping (which technically takes it out of the acyclic workflow class of systems) and function libraries.

The central data abstraction of Spark is called the *Resilient Distributed Dataset*, or *RDD*. An RDD is a file of objects of one type. The primary example of an RDD that we have seen so far is the files of key-value pairs that are used in MapReduce systems. They are also the files that get passed among functions that we talked about in connection with Fig. 2.6. RDD's are "distributed" in the sense that an RDD is normally broken into chunks that may be held at

[7] As we shall discuss in Section 2.4.5, the blocking property only holds for acyclic workflows, and systems that support recursion cannot use it to manage failures.

different compute nodes. They are "resilient" in the sense that we expect to be able to recover from the loss of any or all chunks of an RDD. However, unlike the key-value-pair abstraction of MapReduce, there is no restriction on the type of the elements that comprise an RDD.

A Spark program consists of a sequence of steps, each of which typically applies some function to an RDD to produce another RDD. Such operations are called *transformations*. It is also possible to take data from the surrounding file system, such as HDFS, and turn it into an RDD, and to take an RDD and return it to the surrounding file system or to produce a result that is passed back to an application that called a Spark program. The latter kinds of operations are called *actions*.

We shall not try to list all the available transformations and actions that are available. Neither shall we fix on the dictions of a particular programming language, since the Spark operations are designed to be expressable in a number of different programming languages. However, here are some of the commonly used operations.

Map, Flatmap, and Filter

The Map transformation takes a parameter that is a function, and it applies that function to every element of an RDD, producing another RDD. This operation should remind us of the Map of MapReduce, but it is not exactly the same. First of all, in MapReduce, a Map function can only apply to a key-value pair. Second, in MapReduce, a Map function produces a set of key-value pairs, and each key-value pair is considered an independent element of the output of the Map function. In Spark, a Map function can apply to any object type, but it produces exactly one object as a result. The type of the resulting object can be a set, but that is not the same as producing many objects from one input object. If you want to produce a set of objects from a single object, Spark provides for you another transformation called *Flatmap*, which is analogous to Map of MapReduce, but without the requirement that all types be key-value pairs.

Example 2.7: Suppose our input RDD is a file of documents, as in the "word-count" of Example 2.1. We could write a Spark Map function that takes one document and produces one set of pairs, with each pair of the form $(w, 1)$, where w is one of the words in the document. However, if we do so, then the output RDD is a list of sets, each set consisting of all the words of one document, each word paired with the integer 1. If we want to duplicate the Map function described in Example 2.1, then we need to use Spark's Flatmap transformation. That operation applied to the RDD of documents will produce another RDD, each of whose elements is a single pair $(w, 1)$. □

Spark also provides an operation similar to a limited form of Map, called *Filter*. Instead of a function as a parameter, the Filter transformation takes a predicate that applies to the type of objects in the input RDD. The predicate

returns true or false for each object, and the output RDD of a Filter transformation consists of only those objects in the input RDD for which the filter function returns true.

Example 2.8: Continuing Example 2.7, suppose we want to avoid counting stop words: the most common words like "the" or "and." We could write a filter function that has built into it the list of words we want to eliminate. When applied to a pair $(w, 1)$, this function returns true if and only if w is not on the list. We can then write a Spark program that first applies Flatmap to the RDD of documents, producing an RDD R_1 consisting of a pair $(w, 1)$ for each occurrence of the word w in any of the documents. The program then applies the stop-word-eliminating Filter to R_1, producing another RDD, R_2. The latter RDD consists of a pair $(w, 1)$ for each occurrence of word w in any of the documents, but only if w is not a stop word. □

Reduce

In Spark, the Reduce operation is an action, not a transformation. That is, the operation Reduce applies to an RDD but returns a value and not another RDD. Reduce takes a parameter that is a function which takes two elements of some particular type T and returns another element of the same type T. When applied to an RDD whose elements are of type T, Reduce is applied repeatedly to each pair of consecutive elements, reducing them to a single element. When only one element remains, that becomes the result of the Reduce operation.

For example, if the parameter is the addition function, and this instance of Reduce is applied to an RDD whose elements are integers, then the result will be a single integer that is the sum of all the integers in the RDD. As long as the function parameter is an associative and commutative function, such as addition, it does not matter in which order elements of the input RDD are combined. However, it is also possible to use an arbitrary function, as long as we are satisfied with combination of elements in any order.

Relational Database Operations

There are a number of built-in Spark operations that behave like relational-algebra operators on relations that are represented by RDD's. That is, think of the elements of the RDD's as tuples of a relation. The transformation Join takes two RDD's, each representing one of the relations. The type of each RDD must be a key-value pair, and the key types of both relations must be the same. The Join transformation then looks for two objects, one from each of its input RDD's, such that the key values are the same, say (k, x) and (k, y). For each such pair found, Join produces the key-value pair $\big(k, (x, y)\big)$, and the output RDD consists of all such objects.

The group-by operation of SQL is also implemented in Spark by the transformation GroupByKey. This transformation takes as input an RDD whose type is key-value pairs. The output RDD is also a set of key-value pairs with

the same key type. The value type for the output is a list of values of the input type. GroupByKey sorts its input RDD by key and for each key k produces the pair $(k, [v_1, v_2, \ldots, v_n])$ such that the v_i's are all the values associated with key k in the input RDD. Notice that GroupByKey is exactly the operation that is performed behind the scenes by MapReduce in order to group the output of the Map function by key.

2.4.3 Spark Implementation

There are a number of ways that Spark implementation differs from Hadoop or other MapReduce implementations. We shall discuss two important improvements: lazy evaluation of RDD's and lineage for RDD's. Before we do, we should mention one way in which Spark is similar to MapReduce: the way large RDD's are managed.

Recall that when applying Map to a large file, MapReduce divides that file into chunks and creates a Map task for each chunk or group of chunks. The chunks and their tasks are typically distributed among many different compute nodes. Likewise, many Reduce tasks can run in parallel on different compute nodes, and each of these tasks takes a portion of the entire set of key-value pairs that are passed from Map to Reduce. Spark also allows any RDD to be divided into chunks, which it calls *splits*. Each split can be given to a different compute node, and the transformation on that RDD can be performed in parallel on each of the splits.

Lazy Evaluation

As mentioned in Section 2.4.1, it is common for workflow systems to exploit the blocking property for error handling. To do so, a function is applied to a single intermediate file (analogous to an RDD) and the output of that function is made available to consumers of that output only after the function completes. However, Spark does not actually apply transformations to RDD's until it is required to do so, typically because it must apply some action, e.g., storing a computed RDD in the surrounding file system or returning a result to an application.

The benefit of this strategy of *lazy evaluation* is that many RDD's are not constructed all at once. When one split of an RDD is created at a node, it may be used immediately at the same compute node to apply another transformation. The benefit of this startegy is that this RDD is never stored on disk and never transmitted to another compute nodes, thus saving orders of magnitude in running time in some cases.

Example 2.9: Consider the situation suggested in Example 2.8, where Flatmap is applied to one RDD, which we shall refer to as R_0. Note that RDD R_0 is created by converting the external file of documents into an RDD. As R_0 is a large file, we shall want to divide it into splits and operate on the splits in parallel.

The first transformation on R_0 applies Flatmap to create a set of pairs $(w, 1)$ for each word. For each split of R_0, a split of the resulting RDD, which we called R_1 in Example 2.8, is created at the same compute node. This split of R_1 is then passed to the transformation Filter, which eliminates pairs whose first component is a stop word. When this Filter is applied to the split, the result is a split of the RDD R_2, located at the same compute node.

However, neither the Flatmap nor Filter transformations occur unless an action is applied to R_2. For example, the Spark program may store R_2 in the surrounding file system or perform a Reduce operation that counts occurrences of the words. Only when the program reaches this action does Spark apply the Flatmap and Filter transformations to R_0, running these transformations at each of the compute nodes that holds a split of R_0, in parallel. Thus, the splits of R_1 and R_2 exist only locally at the compute node that created them, and unless the programmer explicitly calls for them to be maintained, these splits are dropped as soon as they are used locally. □

Resilience of RDD's

One may naturally ask what happens in Example 2.9 if a compute node fails after creating a split of R_1 and before transforming that split into a split of R_2. Since R_1 is not backed up to the file system, is it not lost forever? Spark's substitute for redundant storage of intermediate values is to record the *lineage* of every RDD it creates. The lineage tells the Spark system how to recreate the RDD, or a split of the RDD, if that is needed.

Example 2.10: Considering again the situation described in Example 2.9, the lineage for R_2 would say that it is created by applying to R_1 the particular Filter operation that eliminates stop words. In turn, R_1 is created from R_0 by the Flatmap operation that turns words of a document into $(w, 1)$ pairs. And R_0 was created from a particular file of the surrounding file system.

For instance, if we lose a split of R_2, we know we can reconstruct it from the corresponding split of R_1. But since that split exists at the same compute node, we've probably lost that split also. If so, we could reconstruct it from the corresponding split of R_0, which is also probably lost if this compute node has failed. But we know that we can reconstruct the split of R_0 from the surrounding file system, which is presumably redundant and will not be lost. Thus, Spark will find another compute node, reconstruct the lost split of R_0 from the file system there, and then apply the known transformations needed to reconstruct the corresponding splits of R_1 and R_2. □

As we can see from Example 2.10, recovery from a node failure can be more complex in Spark than in MapReduce or in workflow systems that store intermediate values redundantly. However, the tradeoff of more complex recovery when things go wrong against greater speed when things go right is generally a good one. The faster a Spark program runs, the less chance there is of a node failure while running.

We should contrast Spark's need to be able to execute a program in the face of failures with the need for redundant storage of files that are expected to exist for a long period. Over a long period, failures are almost certain, so we are very likely to lose pieces of a file if we do not store it redundantly. But over a short period – minutes or even hours – there is a good chance of avoiding failures. Thus, it is reasonable to be willing to pay more when there *is* a failure in this case.

2.4.4 TensorFlow

TensorFlow is an open-source system developed initially at Google to support machine-learning applications. Like Spark, TensorFlow provides a programming interface in which one writes a sequence of steps. Programs are typically acyclic, although like Spark it is possible to iterate blocks of code.

One major difference between Spark and TensorFlow is the type of data that is passed between steps of the program. In place of the RDD, TensorFlow uses *tensors*; a tensor is simply a multidimensional matrix.

Example 2.11 : A constant, e.g. 3.14159, is regarded as a 0-dimensional tensor. A vector is a 1-dimensional tensor. For instance, the vector $(1, 2, 3)$ can be written in tensorFlow as $[1., 2., 3.]$. A matrix is a 2-dimensional tensor. For example, the matrix

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{array}$$

is expressed as $[[1., 2., 3., 4.],\ [5., 6., 7., 8.],\ [9., 10., 11., 12.]]$.

Higher-dimensional arrays are possible as well. For instance, a 2-by-2-by-2 cube of 0's is represented as $[[[0., 0.],\ [0., 0.]],\ \ [[0., 0.],\ [0., 0.]]]$. □

Although tensors are in fact a restricted form of RDD, the power of TensorFlow comes from its selection of built-in operations. Linear algebra operations are available as functions. For example, if you want matrix C to be the product of matrices A and B, you can write

```
C = tensorflow.matmul(A,B)
```

Even more powerful are the common approaches to machine learning that are built in as operations. a single statement in the TensorFlow language can cause a model that is a tensor to be constructed from training data, which is also represented as a tensor, using a method like gradient descent. (We discuss gradient descent in Sections 9.4.5 and 12.3.4).

2.4.5 Recursive Extensions to MapReduce

Many large-scale computations are really recursions. An important example is PageRank, which is the subject of Chapter 5. That computation is, in simple terms, the computation of the fixedpoint of a matrix-vector multiplication. It is computed under MapReduce systems by the iterated application of the matrix-vector multiplication algorithm described in Section 2.3.1, or by a more complex strategy that we shall introduce in Section 5.2. The iteration typically continues for an unknown number of steps, each step being a MapReduce job, until the results of two consecutive iterations are sufficiently close that we believe convergence has occurred. A second important example of a recursive algorithm on massive data is gradient descent, which we just mentioned in connection with TensorFlow.

Recursions present a problem for failure recovery. Recursive tasks inherently lack the blocking property necessary for independent restart of failed tasks. It is impossible for a collection of mutually recursive tasks, each of which has an output that is input to at least some of the other tasks, to produce output only at the end of the task. If they all followed that policy, no task would ever receive any input, and nothing could be accomplished. As a result, some mechanism other than simple restart of failed tasks must be implemented in a system that handles recursive workflows (flow graphs that are not acyclic). We shall start by studying an example of a recursion implemented as a workflow, and then discuss approaches to dealing with task failures.

Example 2.12: Suppose we have a directed graph whose arcs are represented by the relation $E(X, Y)$, meaning that there is an arc from node X to node Y. We wish to compute the paths relation $P(X, Y)$, meaning that there is a path of length 1 or more from node X to node Y. That is, P is the *transitive closure* of E. A simple recursive algorithm to do so is:

1. Start with $P(X, Y) = E(X, Y)$.
2. While changes to the relation P occur, add to P all tuples in

$$\pi_{X,Y}\big(P(X, Z) \bowtie P(Z, Y)\big)$$

   That is, find pairs of nodes X and Y such that for some node Z there is known to be a path from X to Z and also a known path from Z to Y.

Figure 2.7 suggests how we could organize recursive tasks to perform this computation. There are two kinds of tasks: *Join tasks* and *Dup-elim tasks*. There are n Join tasks, for some n, and each corresponds to a bucket of a hash function h. A path tuple $P(a, b)$, when it is discovered, becomes input to two Join tasks: those numbered $h(a)$ and $h(b)$. The job of the ith Join task, when it receives input tuple $P(a, b)$, is to find certain other tuples seen previously (and stored locally by that task).

1. Store $P(a, b)$ locally.

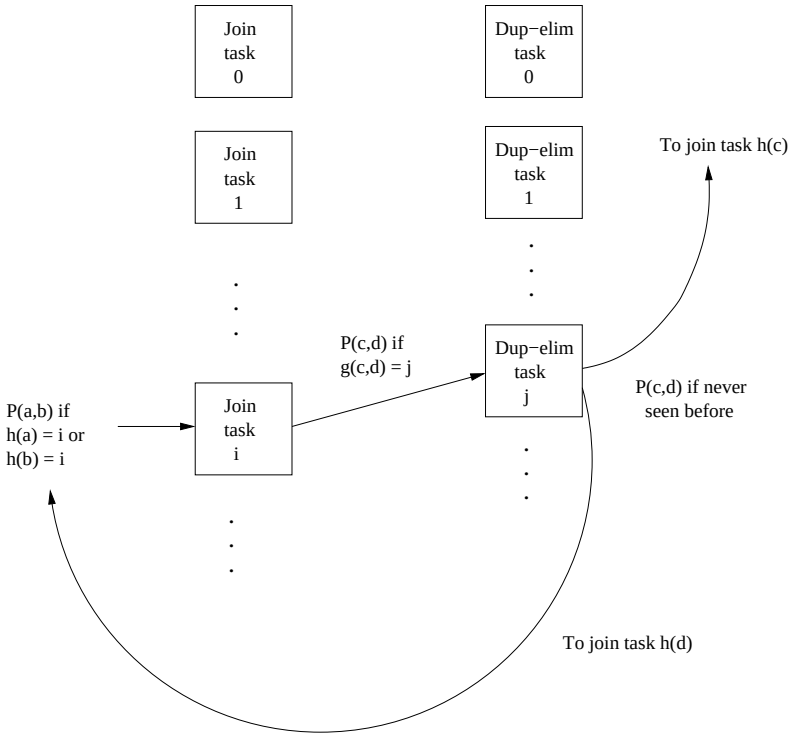

Figure 2.7: Implementation of transitive closure by a collection of recursive tasks

2. If $h(a) = i$ then look for tuples $P(x, a)$ and produce output tuple $P(x, b)$.

3. If $h(b) = i$ then look for tuples $P(b, y)$ and produce output tuple $P(a, y)$.

Note that in rare cases, we have $h(a) = h(b)$, so both (2) and (3) are executed. But generally, only one of these needs to be executed for a given tuple.

There are also m Dup-elim tasks, and each corresponds to a bucket of a hash function g that takes two arguments. If $P(c, d)$ is an output of some Join task, then it is sent to Dup-elim task $j = g(c, d)$. On receiving this tuple, the jth Dup-elim task checks that it has not received this tuple before, since its job is duplicate elimination. If previously received, the tuple is ignored. But if this tuple is new, it is stored locally and sent to two Join tasks, those numbered $h(c)$ and $h(d)$.

Every Join task has m output files – one for each Dup-elim task – and every Dup-elim task has n output files – one for each Join task. These files may be distributed according to any of several strategies. Initially, the $E(a, b)$ tuples representing the arcs of the graph are distributed to the Dup-elim tasks, with $E(a, b)$ being sent as $P(a, b)$ to Dup-elim task $g(a, b)$. The master controller

waits until each Join task has processed its entire input for a round. Then, all output files are distributed to the Dup-elim tasks, which create their own output. That output is distributed to the Join tasks and becomes their input for the next round. □

In Example 2.12 it is not essential to have two kinds of tasks. Rather, Join tasks could eliminate duplicates as they are received, since they must store their previously received inputs anyway. However, this arrangement has an advantage when we must recover from a task failure. If each task stores all the output files it has ever created, and we place Join tasks on different racks from the Dup-elim tasks, then we can deal with any single compute node or single rack failure. That is, a Join task needing to be restarted can get all the previously generated inputs that it needs from the Dup-elim tasks, and vice versa.

In the particular case of computing transitive closure, it is not necessary to prevent a restarted task from generating outputs that the original task generated previously. In the computation of the transitive closure, the rediscovery of a path does not influence the eventual answer. However, many computations cannot tolerate a situation where both the original and restarted versions of a task pass the same output to another task. For example, if the final step of the computation were an aggregation, say a count of the number of nodes reached by each node in the graph, then we would get the wrong answer if we counted a path twice.

There are at least three different approaches that have been used to deal with failures while executing a recursive program.

1. *Iterated MapReduce*: Write the recursion as repeated execution of a MapReduce job or of a sequence of MapReduce jobs. We can then rely on the failure mechanism of the MapReduce implementation to handle failures at any step. The first example of such a system was HaLoop (see the bibliographic notes for this chapter).

2. *The Spark Approach*: The Spark language actually includes iterative statements, such as for-loops that allow the implementation of recursions. Here, failure management is implemented using the lazy-evaluation and lineage mechanisms of Spark. In addition, the Spark programmer has options to store intermediate states of the recursion.

3. *Bulk-Synchronous Systems*: These systems use a graph-based model of computation that we shall describe next. They typically use another resilience approach: periodic checkpointing.

2.4.6 Bulk-Synchronous Systems

Another approach to implementing recursive algorithms on a computing cluster is represented by the Google's Pregel system, which was the first example of a

graph-based, bulk-synchronous system for processing massive amounts of data. Such a system views its data as a graph. Each node of the graph corresponds roughly to a task (although in practice many nodes of a large graph would be bundled into a single task, as in the Join tasks of Example 2.12). Each graph node generates output messages that are destined for other nodes of the graph, and each graph node processes the inputs it receives from other nodes.

Example 2.13: Suppose our data is a collection of weighted arcs of a graph, and we want to find, for each node of the graph, the length of the shortest path to each of the other nodes. As the algorithm executes, each node a will store a set of pairs (b, w), where w is the length of the shortest path from node a to node b that is currently known.

Initially, each graph node a stores the set of pairs (b, w) such that there is an arc from a to b of weight w. These facts are sent to all other nodes, as triples (a, b, w), with the intended meaning that node a knows about a path of length w to node b.[8] When the node a receives a triple (c, d, w), it must decide whether this fact implies a shorter path than a already knows about from itself to node d. Node a looks up its current distance to c; that is, it finds the pair (c, v) stored locally, if there is one. It also finds the pair (d, u) if there is one. If $w + v < u$, then the pair (d, u) is replaced by $(d, w + v)$, and if there was no pair (d, u), then the pair $(d, w + v)$ is stored at the node a. Also, the other nodes are sent the message $(a, d, w + v)$ in either of these two cases. □

Computations in Pregel are organized into *supersteps*. In one superstep, all the messages that were received by any of the nodes at the previous superstep (or initially, if it is the first superstep) are processed, and then all the messages generated by those nodes are sent to their destination. It is this packaging of many messages into one that gives this approach the name "bulk-synchronous."

There is a very important advantage to grouping messages in this way. Communication over a network generally requires a large amount of overhead to send any message, however short. Suppose that in Example 2.13 we sent a single new shortest-distance fact to the relevant node every time one was discovered. The number of messages sent would be enormous if the graph was large, and it would not be realistic to implement such an algorithm. However, in a bulk-synchonous system, a task that has the responsibility for managing many nodes of the graph can bundle together all the messages being sent by its nodes to any of the nodes being managed by another task. That choice typically saves orders of magnitude in the time required to send all the needed messages.

Failure Management in Pregel

In case of a compute-node failure, there is no attempt to restart the failed tasks at that compute node. Rather, Pregel *checkpoints* its entire computation after

[8] This algorithm uses much too much communication, but it will serve as a simple example of the Pregel computation model.

some of the supersteps. A checkpoint consists of making a copy of the entire state of each task, so it can be restarted from that point if necessary. If any compute node fails, the entire job is restarted from the most recent checkpoint.

Although this recovery strategy causes many tasks that have not failed to redo their work, it is satisfactory in many situations. Recall that the reason MapReduce systems support restart of only the failed tasks is that we want assurance that the expected time to complete the entire job in the face of failures is not too much greater than the time to run the job with no failures. Any failure-management system will have that property as long as the time to recover from a failure is much less than the average time between failures. Thus, it is only necessary that Pregel checkpoints its computation after a number of supersteps such that the probability of a failure during that number of supersteps is low.

2.4.7 Exercises for Section 2.4

! **Exercise 2.4.1:** Suppose a job consists of n tasks, each of which takes time t seconds. Thus, if there are no failures, the sum over all compute nodes of the time taken to execute tasks at that node is nt. Suppose also that the probability of a task failing is p per job per second, and when a task fails, the overhead of management of the restart is such that it adds $10t$ seconds to the total execution time of the job. What is the total expected execution time of the job?

! **Exercise 2.4.2:** Suppose a Pregel job has a probability p of a failure during any superstep. Suppose also that the execution time (summed over all compute nodes) of taking a checkpoint is c times the time it takes to execute a superstep. To minimize the expected execution time of the job, how many supersteps should elapse between checkpoints?

2.5 The Communication-Cost Model

In this section we shall introduce a model for measuring the quality of algorithms implemented on a computing cluster of the type so far discussed in this chapter. We assume the computation is described by an acyclic workflow, as discussed in Section 2.4.1. For many applications, the bottleneck is moving data among tasks, such as transporting the outputs of Map tasks to their proper Reduce tasks. As an example, we explore the computation of multiway joins as single MapReduce jobs, and we see that in some situations, this approach is more efficient than the straightforward cascade of 2-way joins.

2.5.1 Communication Cost for Task Networks

Imagine that an algorithm is implemented by an acyclic network of tasks. These tasks could be Map tasks feeding Reduce tasks, as in a standard MapReduce algorithm, or they could be several MapReduce jobs cascaded, or a more general

workflow structure, such as a collection of tasks each of which implements the workflow of Fig. 2.6.[9] The *communication cost* of a task is the size of the input to the task. This size can be measured in bytes. However, since we shall be using relational database operations as examples, we shall often use the number of tuples as a measure of size.

The *communication cost of an algorithm* is the sum of the communication cost of all the tasks implementing that algorithm. We shall focus on the communication cost as the way to measure the efficiency of an algorithm. In particular, we do not consider the amount of time it takes each task to execute when estimating the running time of an algorithm. While there are exceptions, where execution time of tasks dominates, these exceptions are rare in practice. We can explain and justify the importance of communication cost as follows.

- The algorithm executed by each task tends to be very simple, often linear in the size of its input.
- The typical interconnect speed for a computing cluster is one gigabit per second. That may seem like a lot, but it is slow compared with the speed at which a processor executes instructions. Moreover, in many cluster architectures, there is competition for the interconnect when several compute nodes need to communicate at the same time. As a result, the compute node can do a lot of work on a received input element in the time it takes to deliver that element.
- Even if a task executes at a compute node that has a copy of the chunk(s) on which the task operates, that chunk normally will be stored on disk, and the time taken to move the data into main memory may exceed the time needed to operate on the data once it is available in memory.

Assuming that communication cost is the dominant cost, we might still ask why we count only input size, and not output size. The answer to this question involves two points:

1. If the output of one task τ is input to another task, then the size of τ's output will be accounted for when measuring the input size for the receiving task. Thus, there is no reason to count the size of any output except for those tasks whose output forms the result of the entire algorithm.
2. But in practice, the algorithm output is rarely large compared with the input or the intermediate data produced by the algorithm. The reason is that massive outputs cannot be used unless they are summarized or aggregated in some way. For example, although we talked in Example 2.12 of computing the entire transitive closure of a graph, in practice we would want something much simpler, such as the count of the number of nodes

[9]Recall that this figure represented functions, not tasks. As a network of tasks, there would be, for example, many tasks implementing function f, each of which feeds data to each of the tasks for function g and each of the tasks for function i.

reachable from each node, or the set of nodes reachable from a single node.

Example 2.14: Let us evaluate the communication cost for the join algorithm from Section 2.3.7. Suppose we are joining $R(A, B) \bowtie S(B, C)$, and the sizes of relations R and S are r and s, respectively. Each chunk of the files holding R and S is fed to one Map task, so the sum of the communication costs for all the Map tasks is $r + s$. Note that in a typical execution, the Map tasks will each be executed at a compute node holding a copy of the chunk to which it applies. Thus, no internode communication is needed for the Map tasks, but they still must read their data from disk. Since all the Map tasks do is make a simple transformation of each input tuple into a key-value pair, we expect that the computation cost will be small compared with the communication cost, regardless of whether the input is local to the task or must be transported to its compute node.

The sum of the outputs of the Map tasks is roughly as large as their inputs. Each output key-value pair is sent to exactly one Reduce task, and it is unlikely that this Reduce task will execute at the same compute node. Therefore, communication from Map tasks to Reduce tasks is likely to be across the interconnect of the cluster, rather than memory-to-disk. This communication is $O(r + s)$, so the communication cost of the join algorithm is $O(r + s)$.

The Reduce tasks execute the reducer (application of the Reduce function to a key and its associated value list) for one or more values of attribute B. Each reducer takes the inputs it receives and divides them between tuples that came from R and those that came from S. Each tuple from R pairs with each tuple from S to produce one output. The output size for the join can be either larger or smaller than $r + s$, depending on how likely it is that a given R-tuple joins with a given S-tuple. For example, if there are many different B-values, we would expect the output to be small, while if there are few B-values, a large output is likely.

If the output is large, then the computation cost of generating all the outputs from a reducer could be much larger than $O(r+s)$. However, we shall rely on our supposition that if the output of the join is large, then there is probably some aggregation being done to reduce the size of the output. It will be necessary to communicate the result of the join to another collection of tasks that perform this aggregation, and thus the communication cost will be at least proportional to the computation needed to produce the output of the join. □

2.5.2 Wall-Clock Time

While communication cost often influences our choice of algorithm to use in a cluster-computing environment, we must also be aware of the importance of *wall-clock time*, the time it takes a parallel algorithm to finish. Using careless reasoning, one could minimize total communication cost by assigning all the work to one task, and thereby minimize total communication. However, the

wall-clock time of such an algorithm would be quite high. The algorithms we suggest, or have suggested so far, have the property that the work is divided fairly among the tasks. Therefore, the wall-clock time would be approximately as small as it could be, given the number of compute nodes available.

2.5.3 Multiway Joins

To see how analyzing the communication cost can help us choose an algorithm in the cluster-computing environment, we shall examine carefully the case of a multiway join. There is a general theory in which we:

1. Select certain attributes of the relations involved in the natural join of three or more relations to have their values hashed, each to some number of buckets.

2. Select the number of buckets for each of these attributes, subject to the constraint that the product of the numbers of buckets for each attribute is k, the number of reducers that will be used.

3. Identify each of the k reducers with a vector of bucket numbers. These vectors have one component for each of the attributes selected at step (1).

4. Send tuples of each relation to all those reducers where it might find tuples to join with. That is, the given tuple t will have values for some of the attributes selected at step (1), so we can apply the hash function(s) to those values to determine certain components of the vector that identifies the reducers. Other components of the vector are unknown, so t must be sent to reducers for all vectors having any value in these unknown components.

Some examples of this general technique appear in the exercises. Here, we shall look only at the join $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ as an example. Suppose that the relations R, S, and T have sizes r, s, and t, respectively, and for simplicity, suppose p is the probability that

1. An R-tuple and and S-tuple agree on B, and also the probability that

2. An S-tuple and a T-tuple agree on C.

If we join R and S first, using the MapReduce algorithm of Section 2.3.7, then the communication cost is $O(r + s)$, and the size of the intermediate join $R \bowtie S$ is prs. When we join this result with T, the communication of this second MapReduce job is $O(t + prs)$. Thus, the entire communication cost of the algorithm consisting of two 2-way joins is $O(r + s + t + prs)$. If we instead join S and T first, and then join R with the result, we get another algorithm whose communication cost is $O(r + s + t + pst)$.

A third way to take this join is to use a single MapReduce job that joins the three relations at once. Suppose that we plan to use k reducers for this

job. Pick numbers b and c representing the number of buckets into which we shall hash B- and C-values, respectively. Let h be a hash function that sends B-values into b buckets, and let g be another hash function that sends C-values into c buckets. We require that $bc = k$; that is, each reducer corresponds to a pair of buckets, one for the B-value and one for the C-value. The reducer corresponding to bucket pair (i, j) is responsible for joining the tuples $R(u, v)$, $S(v, w)$, and $T(w, x)$ whenever $h(v) = i$ and $g(w) = j$.

As a result, the Map tasks that send tuples of R, S, and T to the reducers that need them must send R- and T-tuples to more than one reducer. For an S-tuple $S(v, w)$, we know the B- and C-values, so we can send this tuple only to the reducer for $\big(h(v), g(w)\big)$. However, consider an R-tuple $R(u, v)$. We know it only needs to go to reducers that correspond to $\big(h(v), y\big)$, for some y. But we don't know y; the value of C could be anything as far as we know. Thus, we must send $R(u, v)$ to c reducers, since y could be any of the c buckets for C-values. Similarly, we must send the T-tuple $T(w, x)$ to each of the reducers $\big(z, g(w)\big)$ for any z. There are b such reducers.

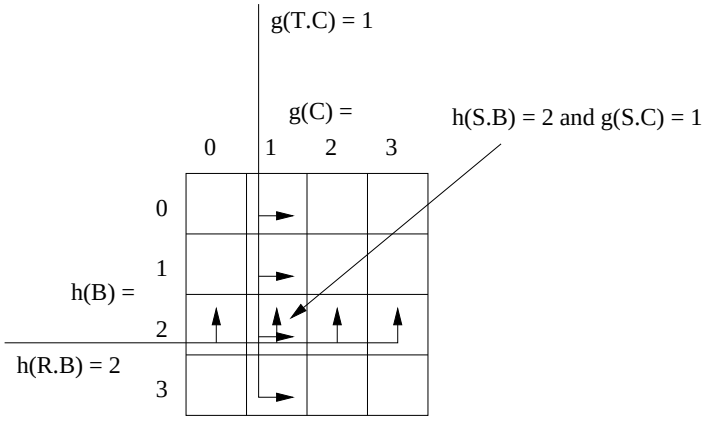

Figure 2.8: Sixteen reducers together perform a 3-way join

Example 2.15: Suppose that $b = c = 4$, so $k = 16$. The sixteen reducers can be thought of as arranged in a rectangle, as suggested by Fig. 2.8. There, we see a hypothetical S-tuple $S(v, w)$ for which $h(v) = 2$ and $g(w) = 1$. This tuple is sent by its Map task only to the reducer for key $(2, 1)$. We also see an R-tuple $R(u, v)$. Since $h(v) = 2$, this tuple is sent to all reducers $(2, y)$, for $y = 1, 2, 3, 4$. Finally, we see a T-tuple $T(w, x)$. Since $g(w) = 1$, this tuple is sent to all reducers $(z, 1)$ for $z = 1, 2, 3, 4$. Notice that these three tuples join, and they meet at exactly one reducer, the reducer for key $(2, 1)$. □

Now, suppose that the sizes of R, S, and T are different; recall we use r, s, and t, respectively, for those sizes. If we hash B-values to b buckets and

Computation Cost of the 3-Way Join

Each of the reducers must compute the join of parts of the three relations, and it is reasonable to ask whether this join can be taken in time that is as small as possible: linear in the sum of the sizes of the input and output for that Reduce task. While more complex joins might not be computable in linear time, the join of our running example can be executed at each Reduce process efficiently. First, create an index on $R.B$, to organize the R-tuples received. Likewise, create an index on $T.C$ for the T-tuples. Then, consider each received S-tuple, $S(v, w)$. Use the index on $R.B$ to find all R-tuples with $R.B = v$ and use the index on $T.C$ to find all T-tuples with $T.C = w$.

C-values to c buckets, where $bc = k$, then the total communication cost for moving the tuples to the proper reducers is the sum of:

1. s to send each tuple $S(v, w)$ to the reducer $\big(h(v), g(w)\big)$.
2. cr to send each tuple $R(u, v)$ to the c reducers $\big(h(v), y\big)$ for each of the c possible values of y.
3. bt to send each tuple $T(w, x)$ to the b reducers $\big(z, g(w)\big)$ for each of the b possible values of z.

There is also a cost $r + s + t$ to make each tuple of each relation be input to one of the Map tasks. This cost is fixed, independent of b, c, and k.

We must select b and c, subject to the constraint $bc = k$, to minimize $s + cr + bt$. We shall use the technique of Lagrangean multipliers to find the place where the function $s + cr + bt - \lambda(bc - k)$ has its derivatives with respect to b and c equal to 0. That is, we must solve the equations $r - \lambda b = 0$ and $t - \lambda c = 0$. Since $r = \lambda b$ and $t = \lambda c$, we may multiply corresponding sides of these equations to get $rt = \lambda^2 bc$. Since $bc = k$, we get $rt = \lambda^2 k$, or $\lambda = \sqrt{rt/k}$. Thus, the minimum communication cost is obtained when $c = t/\lambda = \sqrt{kt/r}$, and $b = r/\lambda = \sqrt{kr/t}$.

If we substitute these values into the formula $s + cr + bt$, we get $s + 2\sqrt{krt}$. That is the communication cost for the Reduce tasks, to which we must add the cost $s + r + t$ for the communication cost of the Map tasks. The total communication cost is thus $r + 2s + t + 2\sqrt{krt}$. In most circumstances, we can neglect $r + t$, because it will be less than $2\sqrt{krt}$, usually by a factor of $O(\sqrt{k})$.

Example 2.16: Let us see under what circumstances the 3-way join has lower communication cost than the cascade of two 2-way joins. To make matters simple, let us assume that R, S, and T are all the same relation R, which represents the "friends" relation in a social network like Facebook. There are

roughly a billion subscribers on Facebook, with an average of 300 friends each, so relation R has $r = 3 \times 10^{11}$ tuples. Suppose we want to compute $R \bowtie R \bowtie R$, perhaps as part of a calculation to find the number of friends of friends of friends each subscriber has, or perhaps just the person with the largest number of friends of friends of friends.[10] The communication cost of the 3-way join of R with itself is $4r + 2r\sqrt{k}$; $3r$ represents the cost of the Map tasks, and $r + 2\sqrt{kr^2}$ is the cost of the Reduce tasks. Since we assume $r = 3 \times 10^{11}$, this cost is $1.2 \times 10^{12} + 6 \times 10^{11}\sqrt{k}$.

Now consider the communication cost of joining R with itself, and then joining the result with R again. The Map and Reduce tasks for the first join each have a cost of $2r$, so the first join only has communication cost $4r = 1.2 \times 10^{12}$. But the size of $R \bowtie R$ is large. We cannot say exactly how large, since friends tend to fall into cliques, and therefore a person with 300 friends will have many fewer than the maximum possible number of friends of friends, which is 90,000. Let us estimate conservatively that the size of $R \bowtie R$ is not $300r$, but only $30r$, or 9×10^{12}. The communication cost for the second join of $(R \bowtie R) \bowtie R$ is thus $1.8 \times 10^{13} + 6 \times 10^{11}$. The total cost of the two joins is therefore $1.2 \times 10^{12} + 1.8 \times 10^{13} + 6 \times 10^{11} = 1.98 \times 10^{13}$.

We must ask whether the cost of the 3-way join, which is

$$1.2 \times 10^{12} + 6 \times 10^{11}\sqrt{k}$$

is less than 1.98×10^{13}. That is so, provided $6 \times 10^{11}\sqrt{k} < 1.86 \times 10^{13}$, or $\sqrt{k} < 31$. That is, the 3-way join will be preferable provided we use no more than $31^2 = 961$ reducers. □

2.5.4 Exercises for Section 2.5

Exercise 2.5.1: What is the communication cost of each of the following algorithms, as a function of the size of the relations, matrices, or vectors to which they are applied?

(a) The matrix-vector multiplication algorithm of Section 2.3.2.

(b) The union algorithm of Section 2.3.6.

(c) The aggregation algorithm of Section 2.3.8.

(d) The matrix-multiplication algorithm of Section 2.3.10.

! **Exercise 2.5.2:** Suppose relations R, S, and T have sizes r, s, and t, respectively, and we want to take the 3-way join $R(A, B) \bowtie S(B, C) \bowtie T(A, C)$, using k reducers. We shall hash values of attributes A, B, and C to a, b, and c buckets, respectively, where $abc = k$. Each reducer is associated with a vector

[10] This person, or more generally, people with large extended circles of friends, are good people to use to start a marketing campaign by giving them free samples.

Star Joins

A common structure for data mining of commercial data is the *star join*. For example, a chain store like Walmart keeps a *fact table* whose tuples each represent a single sale. This relation looks like $F(A_1, A_2, \ldots)$, where each attribute A_i is a key representing one of the important components of the sale, such as the purchaser, the item purchased, the store branch, or the date. For each key attribute there is a *dimension table* giving information about the participant. For instance, the dimension table $D(A_1, B_{11}, B_{12}, \ldots)$ might represent purchasers. A_1 is the purchaser ID, the key for this relation. The B_{1i}'s might give the purchaser's name, address, phone, and so on. Typically, the fact table is much larger than the dimension tables. For instance, there might be a fact table of a billion tuples and ten dimension tables of a million tuples each.

Analysts mine this data by asking *analytic queries* that typically join the fact table with several of the dimension tables (a "star join") and then aggregate the result into a useful form. For instance, an analyst might ask "give me a table of sales of pants, broken down by region and color, for each month of 2016." Under the communication-cost model of this section, joining the fact table and dimension tables by a multiway join is almost certain to be more efficient than joining the relations in pairs. In fact, it may make sense to store the fact table over however many compute nodes are available, and replicate the dimension tables permanently in exactly the same way as we would replicate them should we take the join of the fact table and all the dimension tables. In this special case, only the key attributes (the A's above) are hashed to buckets, and the number of buckets for each key attribute is proportional to the size of its dimension table.

of buckets, one for each of the three hash functions. Find, as a function of r, s, t, and k, the values of a, b, and c that minimize the communication cost of the algorithm.

! **Exercise 2.5.3:** Suppose we take a star join of a fact table $F(A_1, A_2, \ldots, A_m)$ with dimension tables $D_i(A_i, B_i)$ for $i = 1, 2, \ldots, m$. Let there be k reducers, each associated with a vector of buckets, one for each of the key attributes $A_1, A_2, \ldots, A_m$. Suppose the number of buckets into which we hash A_i is a_i. Naturally, $a_1 a_2 \cdots a_m = k$. Finally, suppose each dimension table D_i has size d_i, and the size of the fact table is much larger than any of these sizes. Find the values of the a_i's that minimize the cost of taking the star join as one MapReduce operation.

2.6 Complexity Theory for MapReduce

Now, we shall explore the design of MapReduce algorithms in more detail. Section 2.5 introduced the idea that communication between the Map and Reduce tasks is often the bottleneck when performing a MapReduce computation. Here, we shall look at how the communication cost relates to other desiderata for MapReduce algorithms, in particular our desire to shrink the wall-clock time and to execute each reducer in main memory. Recall that a "reducer" is the execution of the Reduce function on a single key and its associated value list. The point of the exploration in this section is that for many problems there is a spectrum of MapReduce algorithms requiring different amounts of communication. Moreover, the less communication an algorithm uses, the worse it may be in other respects, including wall-clock time and the amount of main memory it requires.

2.6.1 Reducer Size and Replication Rate

Let us now introduce the two parameters that characterize families of MapReduce algorithms. The first is the *reducer size*, which we denote by q. This parameter is the upper bound on the number of values that are allowed to appear in the list associated with a single key. Reducer size can be selected with at least two goals in mind.

1. By making the reducer size small, we can force there to be many reducers, i.e., many different keys according to which the problem input is divided by the Map tasks. If we also create many Reduce tasks – even one for each reducer – then there will be a high degree of parallelism, and we can expect a low wall-clock time.

2. We can choose a reducer size sufficiently small that we are certain the computation associated with a single reducer can be executed entirely in the main memory of the compute node where its Reduce task is located. Regardless of the computation done by the reducers, the running time will be greatly reduced if we can avoid having to move data repeatedly between main memory and disk.

The second parameter is the *replication rate*, denoted r. We define r to be the number of key-value pairs produced by all the Map tasks on all the inputs, divided by the number of inputs. That is, the replication rate is the average communication from Map tasks to Reduce tasks (measured by counting key-value pairs) per input.

Example 2.17: Let us consider the one-pass matrix-multiplication algorithm of Section 2.3.10. Suppose that all the matrices involved are $n \times n$ matrices. Then the replication rate r is equal to n. That fact is easy to see, since for each element m_{ij}, there are n key-value pairs produced; these have all keys of

the form (i, k), for $1 \leq k \leq n$. Likewise, for each element of the other matrix, say n_{jk}, we produce n key-value pairs, each having one of the keys (i, k), for $1 \leq i \leq n$. In this case, not only is n the average number of key-value pairs produced for an input element, but each input produces exactly this number of pairs.

We also see that q, the required reducer size, is $2n$. That is, for each key (i, k), there are n key-value pairs representing elements m_{ij} of the first matrix and another n key-value pairs derived from the elements n_{jk} of the second matrix. While this pair of values represents only one particular algorithm for one-pass matrix multiplication, we shall see that it is part of a spectrum of algorithms, and in fact represents an extreme point, where q is as small as can be, and r is at its maximum. More generally, there is a tradeoff between r and q, that can be expressed as $qr \geq 2n^2$. □

2.6.2 An Example: Similarity Joins

To see the tradeoff between r and q in a realistic situation, we shall examine a problem known as *similarity join*. In this problem, we are given a large set of elements X and a similarity measure $s(x, y)$ that tells how similar two elements x and y of set X are. In Chapter 3 we shall learn about the most important notions of similarity and also learn some tricks that let us find similar pairs quickly. But here, we shall consider only the raw form of the problem, where we have to look at each pair of elements of X and determine their similarity by applying the function s. We assume that s is symmetric, so $s(x, y) = s(y, x)$, but we assume nothing else about s. The output of the algorithm is those pairs whose similarity exceeds a given threshold t.

For example, let us suppose we have a collection of one million images, each of size one megabyte. Thus, the dataset has size one terabyte. We shall not try to describe the similarity function s, but it might, say, involve giving higher values when images have roughly the same distribution of colors or when images have corresponding regions with the same distribution of colors. The goal would be to discover pairs of images that show the same type of object or scene. This problem is extremely hard, but classifying by color distribution is generally of some help toward that goal.

Let us look at how we might do the computation using MapReduce to exploit the natural parallelism found in this problem. The input is key-value pairs (i, P_i), where i is an ID for the picture and P_i is the picture itself. We want to compare each pair of pictures, so let us use one key for each set of two ID's $\{i, j\}$. There are approximately 5×10^{11} pairs of two ID's. We want each key $\{i, j\}$ to be associated with the two values P_i and P_j, so the input to the corresponding reducer will be $(\{i, j\}, [P_i, P_j])$. Then, the Reduce function can simply apply the similarity function s to the two pictures on its value list, that is, compute $s(P_i, P_j)$, and decide whether the similarity of the two pictures is above threshold. The pair would be output if so.

Alas, this algorithm will fail completely. The reducer size is small, since no

list has more than two values, or a total of 2MB of input. Although we don't know exactly how the similarity function s operates, we can reasonably expect that it will not require more than the available main memory. However, the replication rate is 999,999, since for each picture we generate that number of key-value pairs, one for each of the other pictures in the dataset. The total number of bytes communicated from Map tasks to Reduce tasks is 1,000,000 (for the pictures) times 999,999 (for the replication), times 1,000,000 (for the size of each picture). That's approximately 10^{18} bytes, or one exabyte. To communicate this amount of data over gigabit Ethernet would take 10^{10} seconds, or about 300 years.[11]

Fortunately, this algorithm is only the extreme point in a spectrum of possible algorithms. We can characterize these algorithms by grouping pictures into g groups, each of $10^6/g$ pictures.

The Map Function: Take an input element (i, P_i) and generate $g-1$ key-value pairs. For each, the key is one of the sets $\{u, v\}$, where u is the group to which picture i belongs, and v is one of the other groups. The associated value is the pair (i, P_i).

The Reduce Function: Consider the key $\{u, v\}$. The associated value list will have the $2 \times 10^6/g$ elements (j, P_j), where j belongs to either group u or group v. The Reduce function takes each (i, P_i) and (j, P_j) on this list, where i and j belong to different groups, and applies the similarity function $s(P_i, P_j)$. In addition, we need to compare the pictures that belong to the same group, but we don't want to do the same comparison at each of the $g-1$ reducers whose key contains a given group number. There are many ways to handle this problem, but one way is as follows. Compare the members of group u at the reducer $\{u, u+1\}$, where the "+1" is taken in the end-around sense. That is, if $u = g$ (i.e., u is the last group), then $u+1$ is group 1. Otherwise, $u+1$ is the group whose number is one greater than u.

We can compute the replication rate and reducer size as a function of the number of groups g. Each input element is turned into $g-1$ key-value pairs. That is, the replication rate is $g-1$, or approximately $r = g$, since we suppose that the number of groups is still fairly large. The reducer size is $2 \times 10^6/g$, since that is the number of values on the list for each reducer. Each value is about a megabyte, so the number of bytes needed to store the input is $2 \times 10^{12}/g$.

Example 2.18: If g is 1000, then the input consumes about 2GB. That's enough to hold everything in a typical main memory. Moreover, the total number of bytes communicated is now $10^6 \times 999 \times 10^6$, or about 10^{15} bytes. While that is still a huge amount of data to communicate, it is 1000 times less than that of the brute-force algorithm discussed first. Moreover, there are

[11] In a typical cluster, there are many switches connecting subsets of the compute nodes, so all the data does not need to go across a single gigabit switch. However, the total available communication is still small enough that it is not feasible to implement this algorithm for the scale of data we have hypothesized.

still about half a million reducers. Since we are unlikely to have available that many compute nodes, we can divide all the reducers into a smaller number of Reduce tasks and still keep all the compute nodes busy; i.e., we can get as much parallelism as our computing cluster offers us. □

The computation cost for algorithms in this family is independent of the number of groups g, as long as the input to each reducer fits in main memory. The reason is that the bulk of the computation is the application of function s to the pairs of pictures. No matter what value g has, s is applied to each pair once and only once. Thus, although the work of algorithms in the family may be divided among reducers in widely different ways, all members of the family do the same computation.

2.6.3 A Graph Model for MapReduce Problems

In this section, we begin the study of a technique that will enable us to prove lower bounds on the replication rate, as a function of reducer size, for a number of problems. Our first step is to introduce a graph model of problems. This graph describes how the outputs of the problem depend on the inputs. The key idea to be exploited is that, since reducers operate independently, for each output there must be some reducer that gets all of the inputs needed to compute that output. For each problem solvable by a MapReduce algorithm there is:

1. A set of inputs.
2. A set of outputs.
3. A many-many relationship between the inputs and outputs, which describes which inputs are necessary to produce which outputs.

Example 2.19: Figure 2.9 shows the graph for the similarity-join problem discussed in Section 2.6.2, if there were four pictures rather than a million. The inputs are the pictures, and the outputs are the six possible pairs of pictures. Each output is related to the two inputs that are members of its pair. This form of problem, where the outputs are all the pairs of inputs, is common, and we shall refer to it as the *all-pairs* problem. □

Example 2.20: Matrix multiplication presents a more complex graph. If we multiply $n \times n$ matrices M and N to get matrix P, then there are $2n^2$ inputs, m_{ij} and n_{jk}, and there are n^2 outputs p_{ik}. Each output p_{ik} is related to $2n$ inputs: $m_{i1}, m_{i2}, \ldots, m_{in}$ and $n_{1k}, n_{2k}, \ldots, n_{nk}$. Moreover, each input is related to n outputs. For example, m_{ij} is related to $p_{i1}, p_{i2}, \ldots, p_{in}$. Figure 2.10 shows the input-output relationship for matrix multiplication for the simple case of 2×2 matrices, specifically

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} i & j \\ k & l \end{bmatrix}$$

□

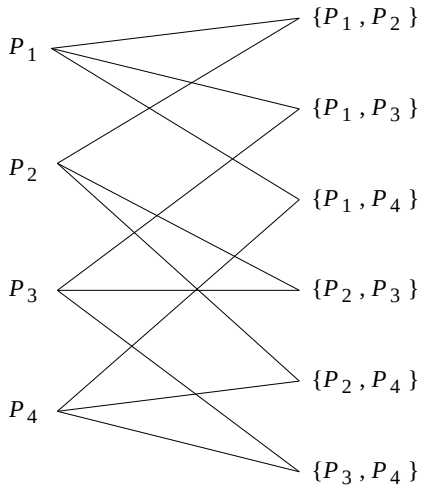

Figure 2.9: Input-output relationship for a similarity join

In the problems of Examples 2.19 and 2.20, the inputs and outputs were clearly all present. However, there are other problems where the inputs and/or outputs may not all be present in any instance of the problem. An example of such a problem is the natural join of $R(A, B)$ and $S(B, C)$ discussed in Section 2.3.7. We assume the attributes A, B, and C each have a finite domain, so there are only a finite number of possible inputs and outputs. The inputs are all possible R-tuples, those consisting of a value from the domain of A paired with a value from the domain of B, and all possible S-tuples – pairs from the domains of B and C. The outputs are all possible triples, with components from the domains of A, B, and C in that order. The output (a, b, c) is connected to two inputs, namely $R(a, b)$ and $S(b, c)$.

But in an instance of the join computation, only some of the possible inputs will be present, and therefore only some of the possible outputs will be produced. That fact does not influence the graph for the problem. We still need to know how every possible output relates to inputs, whether or not that output is produced in a given instance.

2.6.4 Mapping Schemas

Now that we see how to represent problems addressable by MapReduce as graphs, we can define the requirements for a MapReduce algorithm to solve a given problem. Each such algorithm must have a *mapping schema*, which expresses how outputs are produced by the various reducers used by the algorithm. That is, a mapping schema for a given problem with a given reducer size q is an assignment of inputs to one or more reducers, such that:

1. No reducer is assigned more than q inputs.

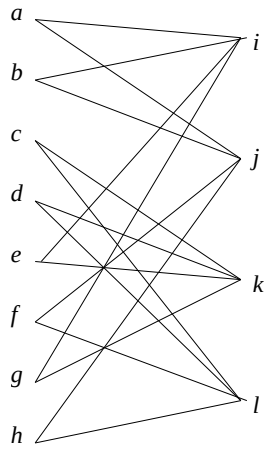

Figure 2.10: Input-output relationship for matrix multiplication

2. For every output of the problem, there is at least one reducer that is assigned all the inputs that are related to that output. We say this reducer *covers* the output.

Point (1) is simply the definition of "reducer size." Point (2) is justified by the fact that reducers can only see the inputs they are given. If no reducer sees all the inputs that an output depends upon, then no reducer can correctly produce that output, and therefore the supposed algorithm will not work. It can be argued that the existence of a mapping schema for any reducer size is what distinguishes problems that can be solved by a single MapReduce job from those that cannot.

Example 2.21 : Let us reconsider the "grouping" strategy we discussed in connection with the all-pairs problem in Section 2.6.2. To generalize the problem, suppose the input is p pictures, which we place in g equal-sized groups of p/g inputs each. The number of outputs is $\binom{p}{2}$, or approximately $p^2/2$ outputs. A reducer will get the inputs from two groups – that is $2p/g$ inputs – so the reducer size we need is $q = 2p/g$. Each picture is sent to the reducers corresponding to the pairs consisting of its group and any of the $g - 1$ other groups. Thus, the replication rate is $g - 1$, or approximately g. If we replace g by the replication rate r in $q = 2p/g$, we conclude that $r = 2p/q$. That is, the replication rate is inversely proportional to the reducer size. That relationship is common; the smaller the reducer size, the larger the replication rate, and therefore the higher the communication.

This family of algorithms is described by a family of mapping schemas, one for each possible q. In the mapping schema for $q = 2p/g$, there are $\binom{g}{2}$, or approximately $g^2/2$ reducers. Each reducer corresponds to a pair of groups, and an input P is assigned to all the reducers whose pair includes the group of

P. Thus, no reducer is assigned more than $2p/g$ inputs; in fact each reducer is assigned exactly that number. Moreover, every output is covered by some reducer. Specifically, if the output is a pair from two different groups u and v, then this output is covered by the reducer for the pair of groups $\{u, v\}$. If the output corresponds to inputs from only one group u, then the output is covered by several reducers – those corresponding to the set of groups $\{u, v\}$ for any $v \neq u$. Note that the algorithm we described has only one of these reducers computing the output, but any of them *could* compute it. □

The fact that an output depends on a certain input means that when that input is processed at the Map task, there will be at least one key-value pair generated to be used when computing that output. The value might not be exactly the input (as was the case in Example 2.21), but it is derived from that input. What is important is that for every related input and output there is a key-value pair that must be communicated. Note that there is technically never a need for more than one key-value pair for a given input and output, because the input could be transmitted to the reducer as itself, and whatever transformations on the input were applied by the Map function could instead be applied by the Reduce function at the reducer for that output.

2.6.5 When Not All Inputs Are Present

Example 2.21 describes a problem where we know every possible input is present, because we can define the input set to be those pictures that actually exist in the dataset. However, as discussed at the end of Section 2.6.3, there are problems like computing the join, where the graph of inputs and outputs describes inputs that might exist, and outputs that are only made when at least one of the inputs exists in the dataset. In fact, for the join, both inputs related to an output must exist if we are to make that output.

An algorithm for a problem where outputs can be missing still needs a mapping schema. The justification is that all inputs, or any subset of them, *might* be present, so an algorithm without a mapping schema would not be able to produce every possible output if all the inputs related to that output happened to be present, and yet no reducer covered that output.

The only way the absence of some inputs makes a difference is that we may wish to rethink the desired value of the reducer size q when we select an algorithm from the family of possible algorithms. Especially, if the value of q we select is that number such that we can be sure the input will just fit in main memory, then we may wish to increase q to take into account that some fraction of the inputs are not really there.

Example 2.22: Suppose that we know we can execute the Reduce function in main memory on a key and its associated list of q values. However, we also know that only 5% of the possible inputs are really present in the data set. Then a mapping schema for reducer size q will really send about $q/20$ of the inputs that exist to each reducer. Put another way, we could use the algorithm

for reducer size $20q$ and expect that an average of q inputs will actually appear on the list for each reducer. We can thus choose $20q$ as the reducer size, or since there will be some randomness in the number of inputs actually appearing at each reducer, we might wish to pick a slightly smaller value of reducer size, such as $18q$. □

2.6.6 Lower Bounds on Replication Rate

The family of similarity-join algorithms described in Example 2.21 lets us trade off communication against the reducer size, and through reducer size to trade communication against parallelism or against the ability to execute the Reduce function in main memory. How do we know we are getting the best possible tradeoff? We can only know we have the minimum possible communication if we can prove a matching lower bound. Using existence of a mapping schema as the starting point, we can often prove such a lower bound. Here is an outline of the technique.

1. Prove an upper bound on how many outputs a reducer with q inputs can cover. Call this bound $g(q)$. This step can be difficult, but for examples like the all-pairs problem, it is actually quite simple.

2. Determine the total number of outputs produced by the problem.

3. Suppose that there are k reducers, and the ith reducer has $q_i < q$ inputs. Observe that $\sum_{i=1}^{k} g(q_i)$ must be no less than the number of outputs computed in step (2).

4. Manipulate the inequality from (3) to get a lower bound on $\sum_{i=1}^{k} q_i$. Often, the trick used at this step is to replace some factors of q_i by their upper bound q, but leave a single factor of q_i in the term for i.

5. Since $\sum_{i=1}^{k} q_i$ is the total communication from Map tasks to Reduce tasks, divide the lower bound from (4) on this quantity by the number of inputs. The result is a lower bound on the replication rate.

Example 2.23: This sequence of steps may seem mysterious, but let us consider the all-pairs problem as an example that we hope will make things clear. Recall that in Example 2.21 we gave an upper bound on the replication rate r of $r \leq 2p/q$, where p was the number of inputs and q was the reducer size. We shall show a lower bound on r that is half that amount, which implies that, although improvements to the algorithm are possible,[12] any reduction in communication for a given reducer size will be by a factor of 2 at most.

For step (1), observe that if a reducer gets q inputs, it cannot cover more than $\binom{q}{2}$, or approximately $q^2/2$ outputs. For step (2), we know there are a

[12] In fact, an algorithm with r very close to p/q exists, for at least some values of p.

total of $\binom{p}{2}$, or approximately $p^2/2$ outputs that each must be covered. The inequality constructed at step (3) is thus

$$\sum_{i=1}^{k} q_i^2/2 \geq p^2/2$$

or, multiplying both sides by 2,

$$\sum_{i=1}^{k} q_i^2 \geq p^2 \tag{2.1}$$

Now, we must do the manipulation of step (4). Following the hint, we note that there are two factors of q_i in each term on the left of Equation (2.1), so we replace one factor by q and leave the other as q_i. Since $q \geq q_i$, we can only increase the left side by doing so, and thus the inequality continues to hold:

$$q \sum_{i=1}^{k} q_i \geq p^2$$

or, dividing by q:

$$\sum_{i=1}^{k} q_i \geq p^2/q \tag{2.2}$$

The final step, which is step (5), is to divide both sides of Equation 2.2 by p, the number of inputs. As a result, the left side, which is $(\sum_{i=1}^{k} q_i)/p$ is equal to the replication rate, and the right side becomes p/q. That is, we have proved the lower bound on r:

$$r \geq p/q$$

As claimed, this shows that the family of algorithms from Example 2.21 all have a replication rate that is at most twice the lowest possible replication rate. □

2.6.7 Case Study: Matrix Multiplication

In this section we shall apply the lower-bound technique to one-pass matrix-multiplication algorithms. We saw one such algorithm in Section 2.3.10, but that is only an extreme case of a family of possible algorithms. In particular, for that algorithm, a reducer corresponds to a single element of the output matrix. Just as we grouped inputs in the all-pairs problem to reduce the communication at the expense of a larger reducer size, we can group rows and columns of the two input matrices into bands. Each pair consisting of a band of rows of the first matrix and a band of columns of the second matrix is used by one reducer to produce a square of elements of the output matrix. An example is suggested by Fig. 2.11.

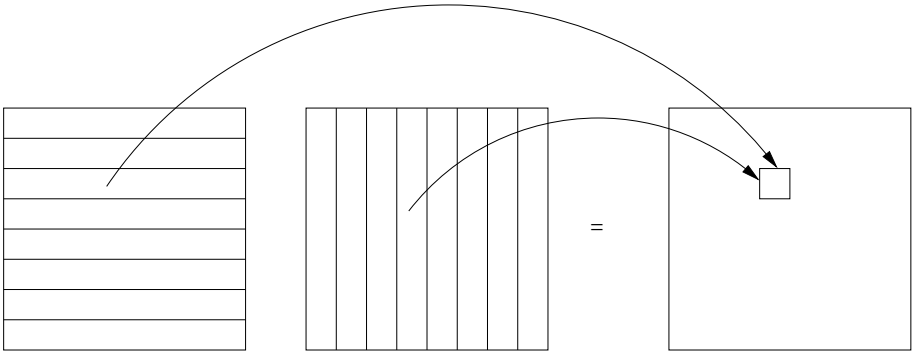

Figure 2.11: Dividing matrices into bands to reduce communication

In more detail, suppose we want to compute $MN = P$, and all three matrices are $n \times n$. Group the rows of M into g bands of n/g rows each, and group the columns of N into g bands of n/g columns each. This grouping is as suggested by Fig. 2.11. Keys correspond to two groups (bands), one from M and one from N.

The Map Function: For each element of M, the Map function generates g key-value pairs. The value in each case is the element itself, together with its row and column number so it can be identified by the Reduce function. The key is the group to which the element belongs, paired with any of the groups of the matrix N. Similarly, for each element of N, the Map function generates g key-value pairs. The key is the group of that element paired with any of the groups of M, and the value is the element itself plus its row and column.

The Reduce Function: The reducer corresponding to the key (i, j), where i is a group of M and j is a group of N, gets a value list consisting of all the elements in the ith band of M and the jth band of N. It thus has all the values it needs to compute the elements of P whose row is one of those rows comprising the ith band of M and whose column is one of those comprising the jth band of N. For instance, Fig. 2.11 suggests the third group of M and the fourth group of N, combining to compute a square of P at the reducer $(3, 4)$.

Each reducer gets $n(n/g)$ elements from each of the two matrices, so $q = 2n^2/g$. The replication rate is g, since each element of each matrix is sent to g reducers. That is, $r = g$. Combining $r = g$ with $q = 2n^2/g$ we can conclude that $r = 2n^2/q$. That is, just as for similarity join, the replication rate varies inversely with the reducer size.

It turns out that this upper bound on replication rate is also a lower bound. That is, we cannot do better than the family of algorithms we described above in a single round of MapReduce. Interestingly, we shall see that we can get a lower total communication for the same reducer size, if we use two passes of MapReduce as we discussed in Section 2.3.9. We shall not give the complete proof of the lower bound, but will suggest the important elements.

For step (1) we need to get an upper bound on how many outputs a reducer of size q can cover. First, notice that if a reducer gets some of the elements in a row of M, but not all of them, then the elements of that row are useless; the reducer cannot produce any output in that row of P. Similarly, if a reducer receives some but not all of a column of N, these inputs are also useless. Thus, we may assume that the best mapping schema will send to each reducer some number of full rows of M and some number of full columns of N. This reducer is then capable of producing output element p_{ik} if and only if it has received the entire ith row of M and the entire kth column of N. The remainder of the argument for step (1) is to prove that the largest number of outputs are covered when the reducer receives the same number of rows as columns. We leave this part as an exercise.

However, assuming a reducer receives k rows of M and k columns of N, then $q = 2nk$, and k^2 outputs are covered. That is, $g(q)$, the maximum number of outputs covered by a reducer that receives q inputs, is $q^2/4n^2$.

For step (2), we know the number of outputs is n^2. In step (3) we observe that if there are k reducers, with the ith reducer receiving $q_i \leq q$ inputs, then

$$\sum_{i=1}^{k} q_i^2/4n^2 \geq n^2$$

or

$$\sum_{i=1}^{k} q_i^2 \geq 4n^4$$

From this inequality, you can derive

$$r \geq 2n^2/q$$

We leave the algebraic manipulation, which is similar to that in Example 2.23, as an exercise.

Now, let us consider the generalization of the two-pass matrix-multiplication algorithm that we described in Section 2.3.9. First, notice that we could have designed the first pass to use one reducer for each triple (i, j, k). This reducer would get only the two elements m_{ij} and n_{jk}. We can generalize this idea to use reducers that get larger sets of elements from each matrix; these sets of elements form squares within their respective matrices. The idea is suggested by Fig. 2.12. We may divide the rows and columns of both input matrices M and N into g groups of n/g rows or columns each. The intersections of the groups partition each matrix into g^2 squares of n^2/g^2 elements each.

The square of M corresponding to set of rows I and set of columns J combines with the square of N corresponding to set of rows J and set of columns K. These two squares compute some of the terms that are needed to produce the square of the output matrix P that has set of rows I and set of columns K. However, these two squares do not compute the full value of these elements of P; rather they produce a contribution to the sum. Other pairs of squares, one

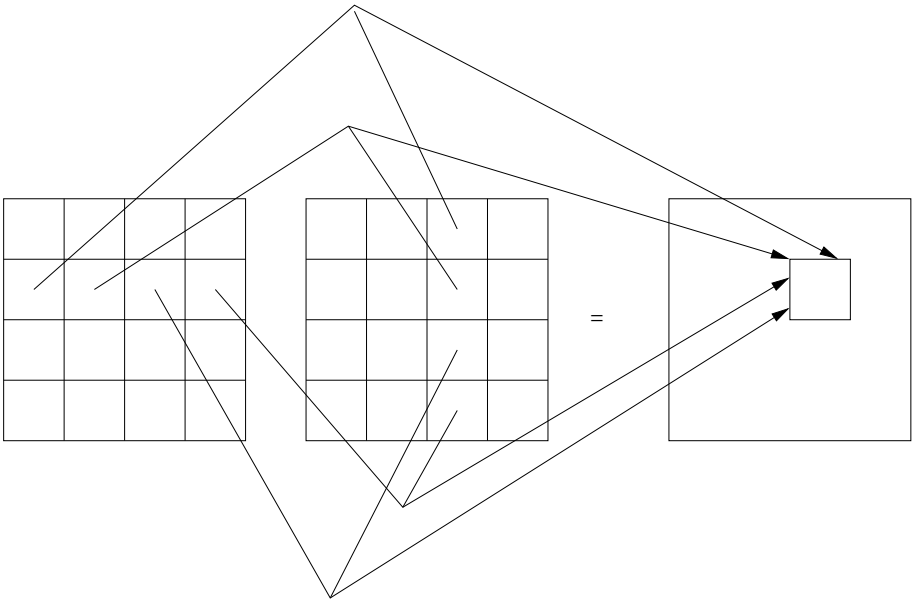

Figure 2.12: Partitioning matrices into squares for a two-pass MapReduce algorithm

from M and one from N, contribute to the same square of P. These contributions are suggested in Fig. 2.12. There, we see how all the squares of M with a fixed value for set of rows I pair with all the squares of N that have a fixed value for the set of columns K by letting the set J vary.

So in the first pass, we compute the products of the square (I, J) of M with the square (J, K) of N, for all I, J, and K. Then, in the second pass, for each I and K we sum the products over all possible sets J. In more detail, the first MapReduce job does the following.

The Map Function: The keys are triples of sets of rows and/or column numbers (I, J, K). Suppose the element m_{ij} belongs to group of rows I and group of columns J. Then from m_{ij} we generate g key-value pairs with value equal to m_{ij}, together with its row and column numbers, i and j, to identify the matrix element. There is one key-value pair for each key (I, J, K), where K can be any of the g groups of columns of N. Similarly, from element n_{jk} of N, if j belongs to group J and k to group K, the Map function generates g key-value pairs with value consisting of n_{jk}, j, and k, and with keys (I, J, K) for any group I.

The Reduce Function: The reducer corresponding to (I, J, K) receives as input all the elements m_{ij} where i is in I and j is in J, and it also receives all the elements n_{jk}, where j is in J and k is in K. It computes

$$x_{iJk} = \sum_{j \text{ in } J} m_{ij} n_{jk}$$

for all i in I and k in K.

Notice that the replication rate for the first MapReduce job is g, and the total communication is therefore $2gn^2$. Also notice that each reducer gets $2n^2/g^2$ inputs, so $q = 2n^2/g^2$. Equivalently, $g = n\sqrt{2/q}$. Thus, the total communication $2gn^2$ can be written in terms of q as $2\sqrt{2}n^3/\sqrt{q}$.

The second MapReduce job is simple; it sums up the x_{iJk}'s over all sets J.

The Map Function: We assume that the Map tasks execute at whatever compute nodes executed the Reduce tasks of the previous job. Thus, no communication is needed between the jobs. The Map function takes as input one element x_{iJk}, which we assume the previous reducers have left labeled with i and k so we know to what element of matrix P this term contributes. One key-value pair is generated. The key is (i, k) and the value is x_{iJk}.

The Reduce Function: The Reduce function simply sums the values associated with key (i, k) to compute the output element P_{ik}.

The communication between the Map and Reduce tasks of the second job is gn^2, since there are n possible values of i, n possible values of k, and g possible values of the set J, and each x_{iJk} is communicated only once. If we recall from our analysis of the first MapReduce job that $g = n\sqrt{2/q}$, we can write the communication for the second job as $n^2g = \sqrt{2}n^3/\sqrt{q}$. This amount is exactly half the communication for the first job, so the total communication for the two-pass algorithm is $3\sqrt{2}n^3/\sqrt{q}$. Although we shall not examine this point here, it turns out that we can do slightly better if we divide the matrices M and N not into squares but into rectangles that are twice as long on one side as on the other. In that case, we get the slightly smaller constant 4 in place of $3\sqrt{2} = 4.24$, and we get a two-pass algorithm with communication equal to $4n^3/\sqrt{q}$.

Now, recall that the communication cost we computed for the one-pass algorithm is $4n^4/q$. We may as well assume q is less than n^2, or else we can just use a serial algorithm at one compute node and not use MapReduce at all. Thus, $n^3/\sqrt{q}$ is smaller than n^4/q, and if q is close to its minimum possible value of $2n$,[13] then the two-pass algorithm beats the one-pass algorithm by a factor of $O(\sqrt{n})$ in communication. Moreover, we can expect the difference in communication to be the significant cost difference. Both algorithms do the same $O(n^3)$ arithmetic operations. The two-pass method naturally has more overhead managing tasks than does the one-job method. On the other hand, the second pass of the two-pass algorithm applies a Reduce function that is associative and commutative. Thus, it might be possible to save some communication cost by using a combiner on that pass.

2.6.8 Exercises for Section 2.6

Exercise 2.6.1: Describe the graphs that model the following problems.

[13] If q is less than $2n$, then a reducer cannot get even one row and one column, and therefore cannot compute any outputs at all.

(a) The multiplication of an $n \times n$ matrix by a vector of length n.

(b) The natural join of $R(A, B)$ and $S(B, C)$, where A, B, and C have domains of sizes a, b, and c, respectively.

(c) The grouping and aggregation on the relation $R(A, B)$, where A is the grouping attribute and B is aggregated by the MAX operation. Assume A and B have domains of size a and b, respectively.

! **Exercise 2.6.2:** Provide the details of the proof that a one-pass matrix-multiplication algorithm requires replication rate at least $r \geq 2n^2/q$, including:

(a) The proof that, for a fixed reducer size, the maximum number of outputs are covered by a reducer when that reducer receives an equal number of rows of M and columns of N.

(b) The algebraic manipulation needed, starting with $\sum_{i=1}^{k} q_i^2 \geq 4n^4$.

!! **Exercise 2.6.3:** Suppose our inputs are bit strings of length b, and the outputs correspond to pairs of strings at Hamming distance 1.[14]

(a) Prove that a reducer of size q can cover at most $(q/2) \log_2 q$ outputs.

(b) Use part (a) to show the lower bound on replication rate: $r \geq b/\log_2 q$.

(c) Show that there are algorithms with replication rate as given by part (b) for the cases $q = 2$, $q = 2^b$, and $q = 2^{b/2}$.

!! **Exercise 2.6.4:** For p that is the square of a prime, show that there is a mapping schema for the all-pairs problem that has $r \leq 1 + p/q$.

2.7 Summary of Chapter 2

- ✦ *Cluster Computing*: A common architecture for very large-scale applications is a cluster of compute nodes (processor chip, main memory, and disk). Compute nodes are mounted in racks, and the nodes on a rack are connected, typically by gigabit Ethernet. Racks are also connected by a high-speed network or switch.

- ✦ *Distributed File Systems*: An architecture for very large-scale file systems has developed recently. Files are composed of chunks of about 64 megabytes, and each chunk is replicated several times, on different compute nodes or racks.

[14] Bit strings have *Hamming distance 1* if they differ in exactly one bit position. You may look ahead to Section 3.5.6 for the general definition.

- *MapReduce*: This programming system allows one to exploit parallelism inherent in cluster computing, and manages the hardware failures that can occur during a long computation on many nodes. Many Map tasks and many Reduce tasks are managed by a Master process. Tasks on a failed compute node are rerun by the Master.

- *The Map Function*: This function is written by the user. It takes a collection of input objects and turns each into zero or more key-value pairs. Keys are not necessarily unique.

- *The Reduce Function*: A MapReduce programming system sorts all the key-value pairs produced by all the Map tasks, forms all the values associated with a given key into a list and distributes key-list pairs to Reduce tasks. Each Reduce task combines the elements on each list, by applying the function written by the user. The results produced by all the Reduce tasks form the output of the MapReduce process.

- *Reducers*: It is often convenient to refer to the application of the Reduce function to a single key and its associated value list as a "reducer."

- *Hadoop*: This programming system is an open-source implementation of a distributed file system (HDFS, the Hadoop Distributed File System) and MapReduce (Hadoop itself). It is available through the Apache Foundation.

- *Managing Compute-Node Failures*: MapReduce systems support restart of tasks that fail because their compute node, or the rack containing that node, fail. Because Map and Reduce tasks deliver their output only after they finish (the blocking property), it is possible to restart a failed task without concern for possible repetition of the effects of that task. It is necessary to restart the entire job only if the node at which the Master executes fails.

- *Applications of MapReduce*: While not all parallel algorithms are suitable for implementation in the MapReduce framework, there are simple implementations of matrix-vector and matrix-matrix multiplication. Also, the principal operators of relational algebra are easily implemented in MapReduce.

- *Workflow Systems*: MapReduce has been generalized to systems that support any acyclic collection of functions, each of which can be instantiated by any number of tasks, each responsible for executing that function on a portion of the data.

- *Spark*: This popular workflow system introduces Resilient, Distributed Datasets (RDD's) and a language in which many common operations on RDD's can be written. Spark has a number of efficiencies, including lazy evaluation of RDD's to avoid secondary storage of intermediate results

and the recording of lineage for RDD's so they can be reconstructed as needed.

- *TensorFlow*: This workflow system is specifically designed to support machine-learning. Data is represented as multidimensional arrays, or tensors, and built-in operations perform many powerful operations, such as linear algebra and model training.

- *Recursive Workflows*: When implementing a recursive collection of functions, it is not always possible to preserve the ability to restart any failed task, because recursive tasks may have produced output that was consumed by another task before the failure. A number of schemes for checkpointing parts of the computation to allow restart of single tasks, or restart all tasks from a recent point, have been proposed.

- *Communication-Cost*: Many applications of MapReduce or similar systems do very simple things for each task. Then, the dominant cost is usually the cost of transporting data from where it is created to where it is used. In these cases, efficiency of a MapReduce algorithm can be estimated by calculating the sum of the sizes of the inputs to all the tasks.

- *Multiway Joins*: It is sometimes more efficient to replicate tuples of the relations involved in a join and have the join of three or more relations computed as a single MapReduce job. The technique of Lagrangean multipliers can be used to optimize the degree of replication for each of the participating relations.

- *Star Joins*: Analytic queries often involve a very large fact table joined with smaller dimension tables. These joins can always be done efficiently by the multiway-join technique. An alternative is to distribute the fact table and replicate the dimension tables permanently, using the same strategy as would be used if we were taking the multiway join of the fact table and every dimension table.

- *Replication Rate and Reducer Size*: It is often convenient to measure communication by the replication rate, which is the communication per input. Also, the reducer size is the maximum number of inputs associated with any reducer. For many problems, it is possible to derive a lower bound on replication rate as a function of the reducer size.

- *Representing Problems as Graphs*: It is possible to represent many problems that are amenable to MapReduce computation by a graph in which nodes represent inputs and outputs. An output is connected to all the inputs that are needed to compute that output.

- *Mapping Schemas*: Given the graph of a problem, and given a reducer size, a mapping schema is an assignment of the inputs to one or more reducers

so that no reducer is assigned more inputs than the reducer size permits, and yet for every output there is some reducer that gets all the inputs needed to compute that output. The requirement that there be a mapping schema for any MapReduce algorithm is a good expression of what makes MapReduce algorithms different from general parallel computations.

- ✦ *Matrix Multiplication by MapReduce*: There is a family of one-pass MapReduce algorithms that performs multiplication of $n \times n$ matrices with the minimum possible replication rate $r = 2n^2/q$, where q is the reducer size. On the other hand, a two-pass MapReduce algorithm for the same problem with the same reducer size can use up to a factor of n less communication.

2.8 References for Chapter 2

GFS, the Google File System, was described in [13]. The paper on Google's MapReduce is [10]. Information about Hadoop and HDFS can be found at [15]. More detail on relations and relational algebra can be found in [25].

Several of the earliest workflow systems were Clustera [11] at the Univ. of Wisconsin, and Hyracks (previously called Hyrax) [6], from UC Irvine, and Microsoft's Dryad [17] and later DryadLINQ [26].

Flink is an open-source workflow system designed to handle streaming data [12]. It was originally developed in the Stratosphere project [5] at TU Berlin. Many of the innovations in Spark are described in [27]. The open-source implementation of Spark is at [21]. The information page about TensorFlow is [24].

The iterated-MapReduce approach to implementing recursion is from Haloop [7]. For a discussion of cluster implementation of recursion, see [1].

Pregel is from [19]. There is an open-source version of Pregel called Giraph [14]. GraphLab [18] is another notable parallel implementation system for graph algorithms. GraphX [22] is a graph-based front-end for Spark.

There are a number of other systems built on a distributed file system and/or MapReduce, which have not been covered here, but may be worth knowing about. [8] describes *BigTable*, a Google implementation of an object store of very large size. A somewhat different direction was taken at Yahoo! with *Pnuts* [9]. The latter supports a limited form of transaction processing, for example.

PIG [20] is an implementation of relational algebra on top of Hadoop. Similarly, *Hive* [16] implements a restricted form of SQL on top of Hadoop. Spark also provides a SQL-like front end [23].

The communication-cost model for MapReduce algorithms and the optimal implementations of multiway joins is from [3]. The material on replication rate, reducer size, and their relationship is from [2]. Solutions to Exercises 2.6.2 and 2.6.3 can be found there. The solution to Exercise 2.6.4 is in [4].

1. F.N. Afrati, V. Borkar, M. Carey, A. Polyzotis, and J.D. Ullman, "Cluster computing, recursion, and Datalog," to appear in *Proc. Datalog 2.0 Workshop*, Elsevier, 2011.

2. F.N. Afrati, A. Das Sarma, S. Salihoglu, and J.D. Ullman, "Upper and lower bounds on the cost of a MapReduce computation." to appear in *Proc. Intl. Conf. on Very Large Databases*, 2013. Also available as CoRR, abs/1206.4377.

3. F.N. Afrati and J.D. Ullman, "Optimizing joins in a MapReduce environment," *Proc. Thirteenth Intl. Conf. on Extending Database Technology*, 2010.

4. F.N. Afrati and J.D. Ullman, "Matching bounds for the all-pairs MapReduce problem," *IDEAS* 2013, pp. 3–4.

5. A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinlander, M.J. Sax, S. Schelter, M. Hoger, K. Tzoumas, and D. Warneke, "The Stratosphere platform for big data analytics," *VLDB J.* **23**:6, pp. 939–964, 2014.

6. V.R. Borkar, M.J. Carey, R, Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," *Intl. Conf. on Data Engineering*, pp. 1151–1162, 2011.

7. Y. Bu, B. Howe, M. Balazinska, and M. Ernst, "HaLoop: efficient iterative data processing on large clusters," *Proc. Intl. Conf. on Very Large Databases*, 2010.

8. F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber, "Bigtable: a distributed storage system for structured data," *ACM Transactions on Computer Systems* **26**:2, pp. 1–26, 2008.

9. B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *PVLDB* **1**:2, pp. 1277–1288, 2008.

10. J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Comm. ACM* **51**:1, pp. 107–113, 2008.

11. D.J. DeWitt, E. Paulson, E. Robinson, J.F. Naughton, J. Royalty, S. Shankar, and A. Krioukov, "Clustera: an integrated computation and data management system," *PVLDB* **1**:1, pp. 28–41, 2008.

12. `flink.apache.org`, Apache Foundation.

13. S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *19th ACM Symposium on Operating Systems Principles*, 2003.

14. `giraph.apache.org`, Apache Foundation.

15. `hadoop.apache.org`, Apache Foundation.

16. `hadoop.apache.org/hive`, Apache Foundation.

17. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks," *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pp. 59–72, ACM, 2007.

18. Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J.M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," —em Proc. VLDB Endowment **5**:8, pp. 716–727, 2012.

19. G. Malewicz, M.N. Austern, A.J.C. Sik, J.C. Denhert, H. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," *Proc. ACM SIGMOD Conference*, 2010.

20. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," *Proc. ACM SIGMOD Conference*, pp. 1099–1110, 2008.

21. `spark.apache.org`, Apache Foundation.

22. `spark.apache.org/graphx`, Apache Foundation.

23. `spark.apache.org/sql`, Apache Foundation.

24. `www.tensorflow.org`.

25. J.D. Ullman and J. Widom, *A First Course in Database Systems*, Third Edition, Prentice-Hall, Upper Saddle River, NJ, 2008.

26. Y. Yu, M. Isard, D. Fetterly, M. Budiu, I. Erlingsson, P.K. Gunda, and J. Currey, "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language," *OSDI*, pp. 1–14, USENIX Association, 2008.

27. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *Proc. 9th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, 2012.