Chapter 2

Large-Scale File Systems and Map-Reduce

Modern Internet applications have created a need 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 in the tens of billions, and

2. Searches in "friends" networks at social-networking sites, which involve graphs with hundreds of millions of nodes and many billions of edges.

To deal with applications such as these, a new software stack has developed. It begins with a new form of file system, which features much larger units than the disk blocks in a conventional operating system and also provides replication of data to protect against the frequent media failures that occur when data is distributed over thousands of disks.

On top of these file systems, we find higher-level programming systems developing. Central to many of these is a programming system called *map-reduce*. Implementations of map-reduce enable many of the most common calculations on large-scale data to be performed on large collections of computers, efficiently and in a way that is tolerant of hardware failures during the computation.

Map-reduce systems are evolving and extending rapidly. We include in this chapter a discussion of generalizations of map-reduce, first to acyclic workflows and then to recursive algorithms. We conclude with a discussion of communication cost and what it tells us about the most efficient algorithms in this modern computing environment.

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.

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

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

1. Files must be stored redundantly. If we did not duplicate the file at several compute nodes, then if one node failed, all its files would be unavailable until the node is replaced. If we did not back up the files at all, and the

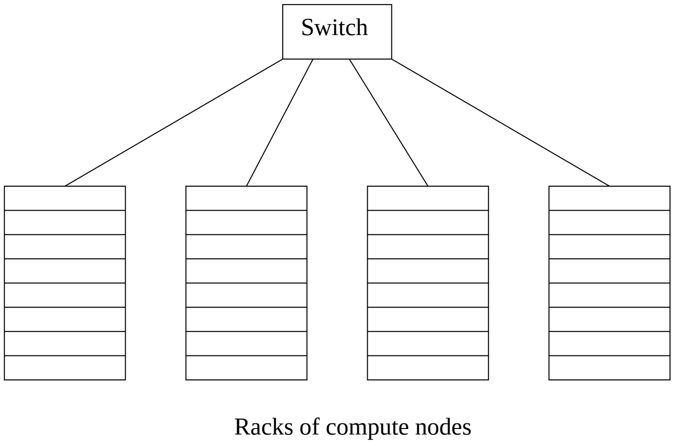

Figure 2.1: Compute nodes are organized into racks, and racks are interconnected by a switch

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 map-reduce programming system that we introduce in Section 2.2.

2.1.2 Large-Scale File-System Organization

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

- 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

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 map-reduce (see Section 2.2) and distributed by the Apache Software Foundation.
3. *CloudStore*, an open-source DFS originally developed by Kosmix.

racks, so we don't lose all copies due to a rack failure. Normally, both the chunk size and the degree of replication can be decided by the user.

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

2.2 Map-Reduce

Map-reduce is a style of computing that has been implemented several times. You can use an implementation of map-reduce to manage many large-scale computations in a way that is tolerant of hardware faults. All you need to write are two functions, called *Map* and *Reduce*, while the system manages the parallel execution, coordination of tasks that execute Map or Reduce, and also deals with the possibility that one of these tasks will fail to execute. In brief, a map-reduce computation executes as follows:

1. Some number of Map tasks each are given one or more chunks from a distributed file system. These Map tasks turn the chunk into a sequence of *key-value* pairs. The way key-value pairs are produced from the input data is determined by the code written by the user for the Map function.
2. The key-value pairs from each Map task are collected by a *master controller* and sorted by key. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task.
3. The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way. The manner of combination

of values is determined by the code written by the user for the Reduce function.

Figure 2.2 suggests this computation.

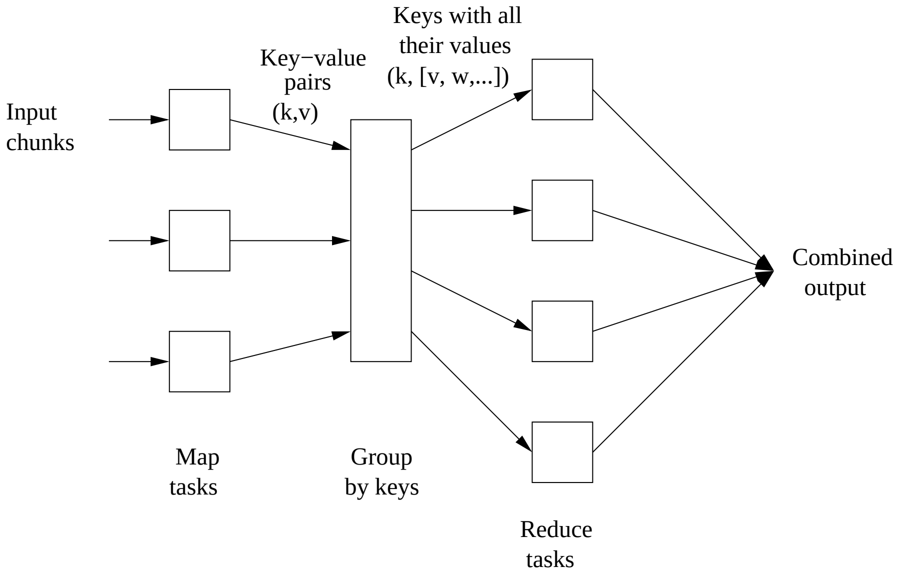

Figure 2.2: Schematic of a map-reduce computation

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 map-reduce processes.

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

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

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

2.2.2 Grouping and Aggregation

Grouping and aggregation is done the same way, regardless of what 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 map-reduce system what r should be. Then the master controller normally picks a hash function that applies to keys and produces a bucket number from 0 to $r - 1$. Each key that is output by a Map task is hashed and its key-value pair is put in one of r local files. Each file is destined for one of the Reduce tasks.[1]

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

2.2.3 The Reduce Tasks

The Reduce function is written to take pairs consisting of a key and its list of associated values and combine those values in some way. The output of a Reduce task is a sequence of key-value pairs consisting of each input key k that the Reduce task received, paired with the combined value constructed from the list of values that the Reduce task received along with key k. 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. Thus, the output of the

[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.

Implementations of Map-Reduce

The original implementation of map-reduce was as an internal and proprietary system at Google. It was called simply "Map-Reduce." There is an open-source implementation called Hadoop. It can be downloaded, along with the HDFS distributed file system, from the Apache Foundation.

Reduce tasks is a sequence of (w, m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents. □

2.2.4 Combiners

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

When the Reduce function is associative and commutative, it is possible to push some of what Reduce does to the Map tasks. For example, instead of the Map tasks in Example 2.1 producing many pairs $(w, 1)$, $(w, 1), \ldots$, we could apply the Reduce function within the Map task, before the output of the Map tasks is subject to grouping and aggregation. These key-value pairs would thus be replaced by one pair with key w and value equal to the sum of all the 1's in all those pairs. That is, the pairs with key w generated by a single Map task would be combined into 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 Map-Reduce Execution

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

The Master has many responsibilities. One is to create some number of Map tasks and some number of Reduce tasks, these numbers being selected by the user program. These tasks will be assigned to Worker processes by the Master. It is reasonable to create one Map task for every chunk of the input

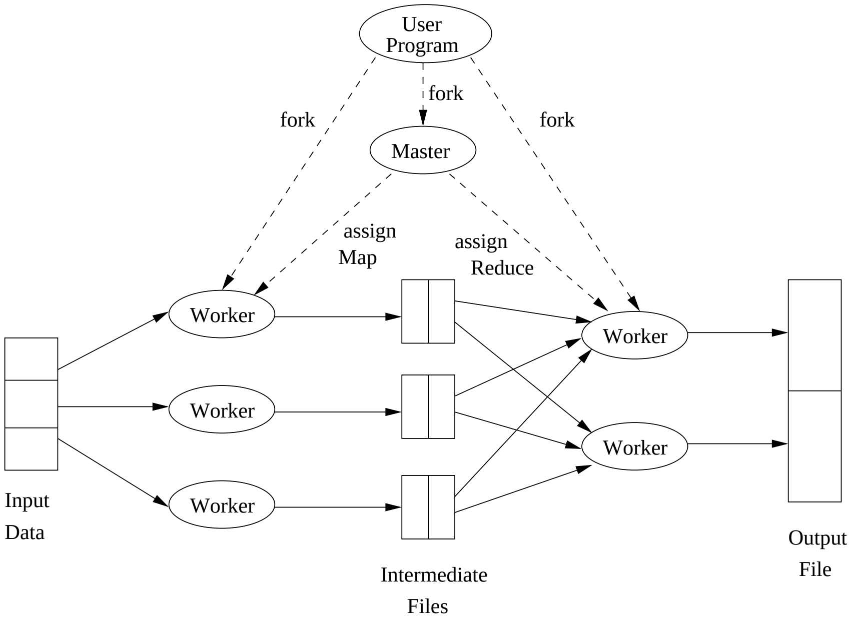

Figure 2.3: Overview of the execution of a map-reduce program

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 map-reduce 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 map-reduce 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.

2.3 Algorithms Using Map-Reduce

Map-reduce is not a solution to every problem, not even every problem that profitably can use many compute nodes operating in parallel. As we mentioned in Section 2.1.2, the entire distributed-file-system milieu makes sense only when files are very large and are rarely updated in place. Thus, we would not expect to use either a DFS or an implementation of map-reduce 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 map-reduce 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 map-reduce 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 map-reduce style of computing. Another important class of operations that can use map-reduce effectively are the relational-algebra operations. We shall examine the map-reduce execution of these operations as well.

2.3.1 Matrix-Vector Multiplication by Map-Reduce

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

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

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 map-reduce for this calculation. But this sort of calculation is at the heart of the ranking of Web pages that goes on at search engines, and there, n is in the tens of billions.[3] Let us first assume that n is large, but not so large that vector $\mathbf{v}$ cannot fit in main memory, and be part of the input to every Map task. It is useful to observe at this time that there is nothing in the definition of map-reduce that forbids providing the same input to more than one 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: Each Map task will take the entire vector $\mathbf{v}$ and 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.

The Reduce Function: A Reduce task has simply to sum all the values associated with a given key i. The result will be a pair (i, x_i).

2.3.2 If the Vector v Cannot Fit in Main Memory

However, it is possible that the vector $\mathbf{v}$ is so large that it will not fit in its entirety in main memory. We don't have to fit it in main memory at a compute node, but if we do 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.

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

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

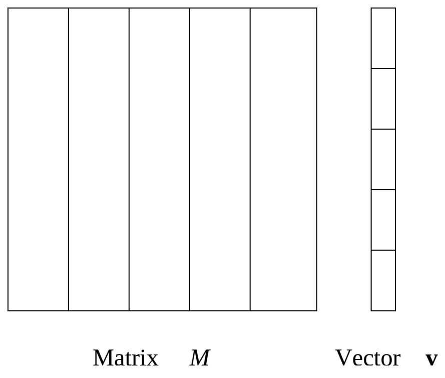

Figure 2.4: Division of a matrix and vector into five stripes

We shall take up matrix-vector multiplication using map-reduce 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. In many traditional database applications, these queries 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 map-reduce.

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 seeing applications of map-reduce 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, $A_2, \ldots, A_n$.

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

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

$(url1, url2)$ that says the Web page $url1$ has a link to page $url2$. While we have shown only four tuples, the real relation of the Web, or the portion of it that would be stored by a typical search engine, has billions of tuples. □

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

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

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

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

3. *Union, Intersection*, and *Difference*: These well-known set operations apply to the sets of tuples in two relations that have the same schema. There are also bag (multiset) versions of the operations in SQL, with somewhat unintuitive definitions, but we shall not go into the bag versions of these operations here.

4. *Natural Join*: Given two relations, compare each pair of tuples, one from each relation. If the tuples agree on all the attributes that are common to the two schemas, then produce a tuple that has components for each of the attributes in either schema and agrees with the two tuples on each attribute. If the tuples disagree on one or more shared attributes, then produce nothing from this pair of tuples. The natural join of relations R and S is denoted $R \bowtie S$. While we shall discuss executing only the natural join with map-reduce, 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 aggregrated attributes have a type that can be compared, e.g., numbers or strings, while SUM and AVG require that the type be arithmetic. We denote a grouping-and-aggregation operation on a relation R by $\gamma_X(R)$, where X is a list of elements that are either

   (a) A grouping attribute, or

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

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

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

[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.

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

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

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.[5] 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 Map-Reduce

Selections really do not need the full power of map-reduce. 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 map-reduce implementation of selection $\sigma_C(R)$.

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

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

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

2.3.5 Computing Projections by Map-Reduce

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

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

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

[5] The COUNT operation applied to an attribute does not consider the values of that attribute, so it is really counting the number of tuples in the group. In SQL, there is a count-distinct operator that counts the number of different values, but we do not discuss this operator here.

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 Map-Reduce

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 two values $[t, t]$ associated with it, then the Reduce task for t should produce (t, t). However, if the value 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. We need to produce a value that indicates "no tuple," such as the SQL value `NULL`. When the result relation is constructed from the output, such a tuple will be ignored.

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 $(t, \texttt{NULL})$.

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, not the entire relation.

The Reduce Function: For each key t, do the following.

1. If the associated value list is $[R]$, then produce (t, t).

2. If the associated value list is anything else, which could only be $[R, S]$, $[S, R]$, or $[S]$, produce $(t, \texttt{NULL})$.

2.3.7 Computing Natural Join by Map-Reduce

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

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

The Reduce Function: Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c). Construct all pairs consisting of one with first component R and the other with first component S, say (R, a) and (S, c). The output for key b is $(b, [(a_1, b, c_1),\ (a_2, b, c_2), \ldots])$, that is, b associated with the list of tuples that can be formed from an R-tuple and an S-tuple with a common b value.

There are a few observations we should make about this join algorithm. First, the relation that is the result of the join is recovered by taking all the tuples that appear on the lists for any key. Second, map-reduce implementations such as Hadoop pass values to the Reduce tasks sorted by key. If so, then identifying all the tuples from both relations that have key b is easy. If another implementation were not to provide key-value pairs sorted by key, then the Reduce function could still manage its task efficiently by hashing key-value pairs locally by key. If enough buckets were used, most buckets would have only one key. Finally, if there are n tuples of R with B-value b and m tuples from S with B-value b, then there are mn tuples with middle component b in the result. In the extreme case, all tuples from R and S have the same b-value, and we are really taking a Cartesian product. However, it is quite common for the number of tuples with shared B-values to be small, and in that case, the time complexity of the Reduce function is closer to linear in the relation sizes than to quadratic.

2.3.8 Generalizing the Join Algorithm

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 and the values of all the attributes of R but not S, and the value for a tuple of S is the name S and the values of the attributes of S but not R.

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

2.3.9 Grouping and Aggregation by Map-Reduce

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

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

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

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

2.3.10 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 map-reduce 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 map-reduce operations, as follows. First:

The Map Function: Send each matrix element m_{ij} to the key value pair

$$\big(j, (M, i, m_{ij})\big)$$

Send each matrix element n_{jk} to the key value pair $\big(j, (N, k, n_{jk})\big)$.

The Reduce Function: For each key j, examine its list of associated values. For each value that comes from M, say $(M, i, m_{ij})\big)$, and each value that comes from N, say $(N, k, n_{jk})\big)$, produce the tuple $(i, k, m_{ij}n_{jk})$. Note that the output of the Reduce function is a key j paired with the list of all the tuples of this form that we get from j.

Now, we perform a grouping and aggregation by another map-reduce operation.

The Map Function: The elements to which this Map function is applied are the pairs that are output from the previous Reduce function. These pairs are of the form

$$(j, [(i_1, k_1, v_1),\ (i_2, k_2, v_2), \ldots, (i_p, k_p, v_p)]$$

where each v_q is the product of elements $m_{i_q j}$ and n_{jk_q}. From this element we produce p key-value pairs:

$$\big((i_1, k_1), v_1\big),\ \big((i_2, k_2), v_2\big), \ldots, \big((i_p, k_p), v_p\big)$$

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 $P = MN$.

2.3.11 Matrix Multiplication with One Map-Reduce Step

There often is more than one way to use map-reduce to solve a problem. You may wish to use only a single map-reduce pass to perform matrix multiplication $P = MN$. 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 a key-value pair $\big((i,k),\ (M,j,m_{ij})\big)$ for $k = 1, 2, \ldots$, up to the number of columns of N. Also, for each element n_{jk} of N, produce a key-value pair $\big((i,k),\ (N,j,n_{jk})\big)$ for $i = 1, 2, \ldots$, up to the number of rows of M.

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.

2.3.12 Exercises for Section 2.3

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

(a) The largest integer.

(b) The average of all the integers.

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

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

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

! **Exercise 2.3.3 :** In the form of relational algebra implemented in SQL, relations are not sets, but bags; that is, tuples are allowed to appear more than once. There are extended definitions of union, intersection, and difference for bags, which we shall define below. Write map-reduce 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 map-reduce 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 map-reduce implementation of this operation, assuming R and S are sets.

2.4 Extensions to Map-Reduce

Map-reduce has proved so influential that it has spawned a number of extensions and modifications. These systems typically share a number of characteristics with map-reduce 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.

In this section, we shall mention some of the interesting directions being explored. References to the details of the systems mentioned can be found in the bibliographic notes for this chapter.

2.4.1 Workflow Systems

Two experimental systems called Clustera from the University of Wisconsin and Hyracks from the University of California at Irvine extend map-reduce 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 $a \rightarrow b$ represent the fact that function a's output is input to function b. 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 input is available. For instance, function h takes ts input from a preexisting file of the distributed file system. Each of h's output elements is passed to at least one of the functions i and j.

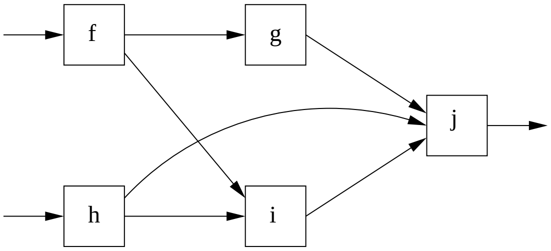

Figure 2.6: An example of a workflow that is more complex than Map feeding Reduce

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, usually 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 at the appropriate time – after the task has completed its work.

The functions of a workflow, and therefore the tasks, share with map-reduce tasks the important property 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. 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.

Many applications of workflow systems such as Clustera or Hyracks are cascades of map-reduce jobs. An example would be the join of three relations, where one map-reduce job joins the first two relations, and a second map-reduce 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 map-reduce 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 map-reduce job and then initiated a second map-reduce job (although Hadoop and other map-reduce systems also try to locate Map tasks where a copy of their input is already present).

2.4.2 Recursive Extensions to Map-Reduce

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 map-reduce 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 map-reduce job, until the results of two consecutive iterations are sufficiently close that we believe convergence has occurred.

The reason recursions are normally implemented by iterated map-reduce jobs is that a true recursive task does not have the 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 2.6 : 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. 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

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

   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 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.

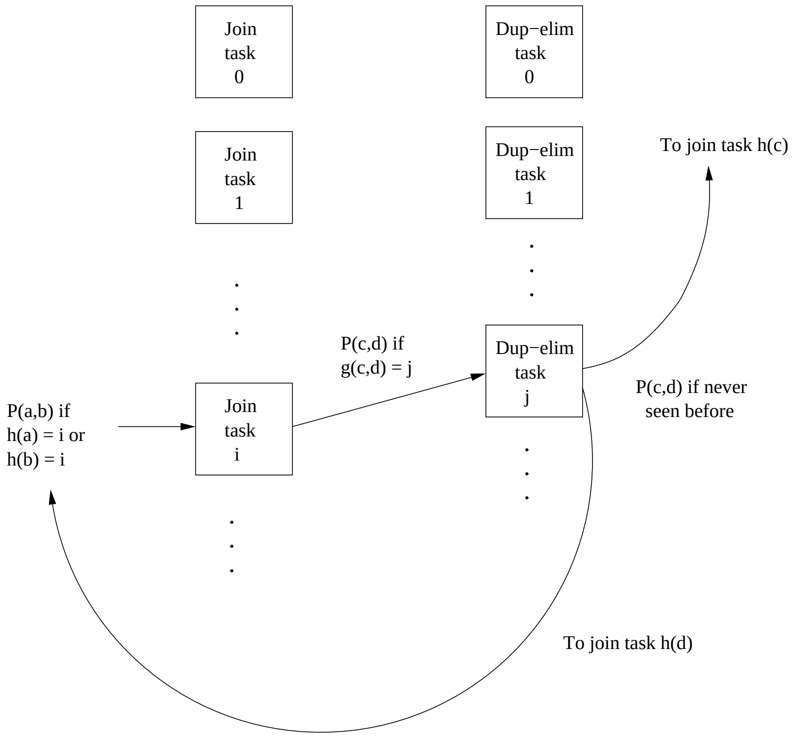

Figure 2.7: Implementation of transitive closure by a collection of recursive tasks

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 had not received it 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 can wait 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. Alternatively, each task can wait until it has produced enough output to justify transmitting its output files to their destination, even if the task has not consumed all its input. □

In Example 2.6 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. In such a case, the master controller can record what files each task generated and passed to other tasks. It can then restart a failed task and ignore those files when the restarted version produces them a second time.

2.4.3 Pregel

Another approach to managing failures when implementing recursive algorithms on a computing cluster is represented by the Pregel system. This 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 2.6). 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 2.7: 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. 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 initially sent to all other nodes, as triples (a, b, w).[6] When the node a receives a triple (c, d, w), it 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.

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

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.

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 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 map-reduce 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.

2.4.4 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 Efficiency of Cluster-Computing Algorithms

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. We then argue that 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 map-reduce jobs, and we see that in some situations, this approach is more efficient than the straightforward cascade of 2-way joins.

2.5.1 The Communication-Cost Model for Cluster Computing

Imagine that an algorithm is implemented by an acyclic network of tasks. These could be Map tasks feeding Reduce tasks, as in a standard map-reduce algorithm, or they could be several map-reduce jobs cascaded, or a more general workflow structure, such as a collection of tasks each of which implements the workflow of Fig. 2.6.[7] 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, we justify the focus on communication cost as follows.

- The algorithm executed by each task tends to be very simple, at most 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. 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:

[7] Note 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.

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. In practice, the output of a job is rarely large compared with the input or the intermediate data produced by the job. 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 2.6 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 reachable from each node, or the set of nodes reachable from a single node.

Example 2.8 : 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 $O(r + s)$.

Observe that the Reduce tasks can use a hash join of the tuples received. This process involves hashing each of the tuples received on their B-values, using a different hash function from the one that divided the tuples among Reduce tasks. The local hash join takes time that is linear in the number of tuples received, and thus is also $O(r + s)$. We do not count this execution time in the communication-cost model, but it is comforting to know that the computation cost is surely not the dominant factor for this algorithm.

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. 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. This aggregation typically can be executed by the Reduce tasks as they produce their output. □

2.5.2 Elapsed Communication Cost

There is another measure of cost based on communication that is worth mentioning, although we shall not use in the developments of this section. The *elapsed communication cost* is the maximum, over all paths through the acyclic network, of the sum of the communication costs of the tasks along that path. For example, in a map-reduce job, the elapsed communication cost is the sum of the maximum input size for any Map task, plus the maximum input size for any Reduce task.

Elapsed communication cost corresponds to the minimum wall-clock time for the execution of a parallel algorithm. Using careless reasoning, one could minimize total communication cost by assigning all the work to one task, and thereby minimize total communication. However, the elapsed 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, so the elapsed communication cost would be approximately as small as it could be.

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 a natural join to have their values hashed 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 Reduce tasks that will be used.

3. Identify each of the k Reduce tasks with a vector of bucket numbers, one for each of the hashed attributes.

4. Send tuples of each relation to all those Reduce tasks where it might find tuples to join with. That is, the given tuple t will have values for some of the hashed attributes, so we can apply the hash function(s) to those values to determine certain components of the vector identifying the Reduce tasks. Other components of the vector are unknown, so t must be sent to all the Reduce tasks 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 $R(A,B) \bowtie S(B,C) \bowtie T(C,D)$. Suppose that the relations R, S, and T have sizes r, s, and t, respectively, and for simplicity, suppose that the probability is p 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 map-reduce 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 map-reduce 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 map-reduce job that joins the three relations at once. Suppose that we plan to use k Reduce tasks 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 Reduce task corresponds to a pair of buckets, one for the B-value and one for the C-value. The Reduce task 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 Reduce tasks that need them must send R- and T-tuples to more than one Reduce task. For an S-tuple $S(v, w)$, we know the B- and C-values, so we can send this tuple only to the Reduce task $\big(h(v), g(w)\big)$. However, consider an R-tuple $R(u, v)$. We know it only needs to go to Reduce tasks $\big(h(v), y\big)$, but we don't know the value of y. The value of C could be anything as far as we know. Thus, we must send $R(u, v)$ to c reduce tasks, 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 Reduce tasks $\big(z, g(w)\big)$ for any z. There are b such tasks.

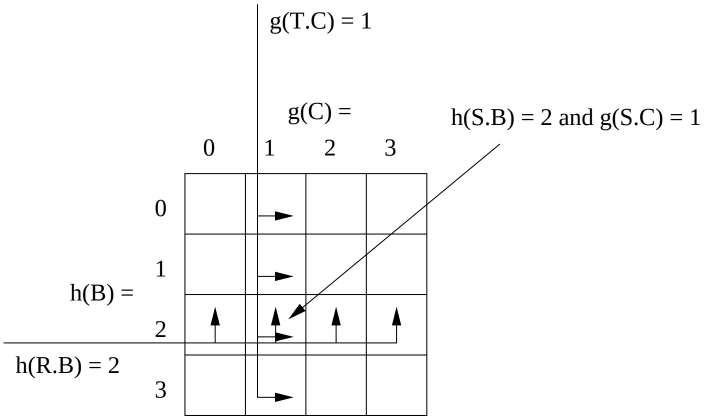

Figure 2.8: Sixteen Reduce tasks together perform a 3-way join

Example 2.9: Suppose that $b = c = 4$, so $k = 16$. The sixteen Reduce tasks

Computation Cost of the 3-Way Join

Each of the Reduce tasks must join of parts of the three relations, and it is reasonable to ask whether this join can be taken in time that is linear in the size of the input to 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 $T.C = w$.

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 Reduce task $(2, 1)$. We also see an R-tuple $R(u, v)$. Since $h(v) = 2$, this tuple is sent to all Reduce tasks $(2, y)$, 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 Reduce tasks $(z, 1)$ for $z = 1, 2, 3, 4$. Notice that these three tuples join, and they meet at exactly one Reduce task, the task numbered $(2, 1)$. □

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

1. s to move each tuple $S(v, w)$ once to the Reduce task $\big(h(v), g(w)\big)$.

2. cr to move each tuple $R(u, v)$ to the c Reduce tasks $\big(h(v), y\big)$ for each of the c possible values of y.

3. bt to move each tuple $T(w, x)$ to the b Reduce tasks $\big(z, g(w)\big)$ 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 latter term typically will be smaller than the first term by a factor $O(\sqrt{k})$, so we can neglect it in most situations, as long as the number of Reduce tasks is reasonably large.

Example 2.10 : 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 300,000,000 subscribers on Facebook, with an average of 300 friends each, so relation R has $r = 9 \times 10^{10}$ 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.[8] The 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 $r = 9 \times 10^{10}$, this cost is $3.6 \times 10^{11} + 1.8 \times 10^{11}\sqrt{k}$.

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 costs $4r = 3.6 \times 10^{11}$. 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 2.7×10^{12}. The communication cost for the second join of $(R \bowtie R) \bowtie R$ is thus $5.4 \times 10^{12} + 1.8 \times 10^{11}$. The total cost of the two joins is therefore $3.6 \times 10^{11} + 5.4 \times 10^{12} + 1.8 \times 10^{11} = 5.94 \times 10^{12}$.

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

$$3.6 \times 10^{11} + 1.8 \times 10^{11}\sqrt{k}$$

is less than 5.94×10^{12}. That is so, provided $1.8 \times 10^{11}\sqrt{k} < 5.58 \times 10^{12}$, or $\sqrt{k} < 31$. That is, the 3-way join will be preferable provided we use no more than $31^2 = 961$ Reduce tasks. □

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.

[8] 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 2010." 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 inversely proportional to the size of its dimension table.

(b) The union algorithm of Section 2.3.6.

(c) The aggregation algorithm of Section 2.3.9.

(d) The matrix-multiplication algorithm of Section 2.3.11.

! **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 Reduce tasks. We shall hash values of attributes A, B, and C to a, b, and c buckets, respectively, where $abc = k$. Each Reduce task is associated with a vector 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 Reduce tasks, 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 map-reduce operation.

2.6 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.

- *Map-Reduce*: 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. Key values are not necessarily unique.

- *The Reduce Function*: A map-reduce 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 map-reduce process.

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

- *Managing Compute-Node Failures*: Map-reduce 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, 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 Map-Reduce*: While not all parallel algorithms are suitable for implementation in the map-reduce framework, there are simple implementations of matrix-vector and matrix-matrix multiplication. Also, the principal operators of relational algebra are easily implemented in map-reduce.

- *Workflow Systems*: Map-reduce 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. Clustera and Hyracks are examples of such systems.

- *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.

- *The Communication-Cost Model*: Many applications of map-reduce 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 an 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 map-reduce 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.

2.7 References for Chapter 2

GFS, the Google File System, was described in [10]. The paper on Google's map-reduce is [8]. Information about Hadoop and HDFS can be found at [11]. More detail on relations and relational algebra can be found in [16].

Clustera is covered in [9]. Hyracks (previously called Hyrax) is from [4]. The Dryad system [13] has similar capabilities, but requires user creation of parallel tasks. That responsibility was automated through the introduction of DryadLINQ [17]. For a discussion of cluster implementation of recursion, see [1]. Pregel is from [14].

A different approach to recursion was taken in Haloop [5]. There, recursion is seen as an iteration, with the output of one round being input to the next round. Efficiency is obtained by managing the location of the intermediate data and the tasks that implement each round.

The communication-cost model for algorithms comes from [2]. [3] discusses optimal implementations of multiway joins using a map-reduce system.

There are a number of other systems built on a distributed file system and/or map-reduce, which have not been covered here, but may be worth knowing about. [6] describes *BigTable*, a Google implementation of an object store of very large size. A somewhat different direction was taken at Yahoo! with *Pnuts* [7]. The latter supports a limited form of transaction processing, for example.

PIG [15] is an implementation of relational algebra on top of Hadoop. Similarly, *Hive* [12] implements a restricted form of SQL on top of Hadoop.

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 and J.D. Ullman, "A new computation model for cluster computing," `http://ilpubs.stanford.edu:8090/953`, Stanford Dept. of CS Technical Report, 2009.

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

4. V. Borkar and M. Carey, "Hyrax: demonstrating a new foundation for data-parallel computation,"

   `http://asterix.ics.uci.edu/pub/hyraxdemo.pdf`

   Univ. of California, Irvine, 2010.

5. 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.

6. 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.

7. 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.

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

9. 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.

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

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

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

13. 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.

14. 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.

15. 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.

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

17. 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.