Chapter 20

Parallel and Distributed Databases

While many databases sit at a single machine, a database can also be distributed over many machines. There are other databases that reside at a single highly parallel machine. When computation is either parallel or distributed, there are many database-implementation issues that need to be reconsidered.

In this chapter, we first look at the different kinds of parallel architectures that have been used. On a parallel machine it is important that the most expensive operations take advantage of parallelism, and for databases, these operations are the full-relation operations such as join. We then discuss the map-reduce paradigm for expressing large-scale computations. This formulation of algorithms is especially amenable to execution on large-scale parallel machines, and it is simple to express important database processes in this manner.

We then turn to distributed architectures. These include grids and networks of workstations, as well as corporate databases that are distributed around the world. Now, we must worry not only about exploiting the many available processors for query execution, but some database operations become much harder to perform correctly in a distributed environment. Notable among these are distributed commitment of transactions and distributed locking.

The extreme case of a distributed architecture is a collection of independent machines, often called "peer-to-peer" networks. In these networks, even data lookup becomes problematic. We shall therefore discuss distributed hash tables and distributed search in peer-to-peer networks.

20.1 Parallel Algorithms on Relations

Database operations, frequently being time-consuming and involving a lot of data, can generally profit from parallel processing. In this section, we shall review the principal architectures for parallel machines. We then concentrate on the "shared-nothing" architecture, which appears to be the most cost effective for database operations, although it may not be superior for other parallel applications. There are simple modifications of the standard algorithms for most relational operations that will exploit parallelism almost perfectly. That is, the time to complete an operation on a p-processor machine is about $1/p$ of the time it takes to complete the operation on a uniprocessor.

20.1.1 Models of Parallelism

At the heart of all parallel machines is a collection of processors. Often the number of processors p is large, in the hundreds or thousands. We shall assume that each processor has its own local cache, which we do not show explicitly in our diagrams. In most organizations, each processor also has local memory, which we do show. Of great importance to database processing is the fact that along with these processors are many disks, perhaps one or more per processor, or in some architectures a large collection of disks accessible to all processors directly.

Additionally, parallel computers all have some communications facility for passing information among processors. In our diagrams, we show the communication as if there were a shared bus for all the elements of the machine. However, in practice a bus cannot interconnect as many processors or other elements as are found in the largest machines, so the interconnection system in many architectures is a powerful switch, perhaps augmented by busses that connect subsets of the processors in local clusters. For example, the processors in a single rack are typically connected.

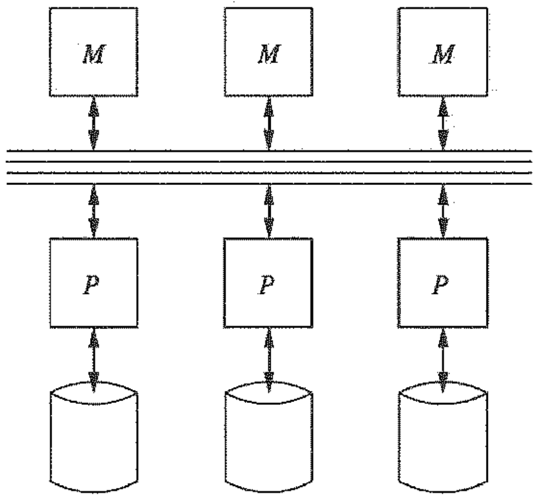

Figure 20.1: A shared-memory machine

We can classify parallel architectures into three broad groups. The most tightly coupled architectures share their main memory. A less tightly coupled architecture shares disk but not memory. Architectures that are often used for databases do not even share disk; these are called "shared nothing" architectures, although the processors are in fact interconnected and share data through message passing.

Shared-Memory Machines

In this architecture, illustrated in Fig. 20.1, each processor has access to all the memory of all the processors. That is, there is a single physical address space for the entire machine, rather than one address space for each processor. The diagram of Fig. 20.1 is actually too extreme, suggesting that processors have no private memory at all. Rather, each processor has some local main memory, which it typically uses whenever it can. However, it has direct access to the memory of other processors when it needs to. Large machines of this class are of the *NUMA* (nonuniform memory access) type, meaning that it takes somewhat more time for a processor to access data in a memory that "belongs" to some other processor than it does to access its "own" memory, or the memory of processors in its local cluster. However, the difference in memory-access times are not great in current architectures. Rather, all memory accesses, no matter where the data is, take much more time than a cache access, so the critical issue is whether or not the data a processor needs is in its own cache.

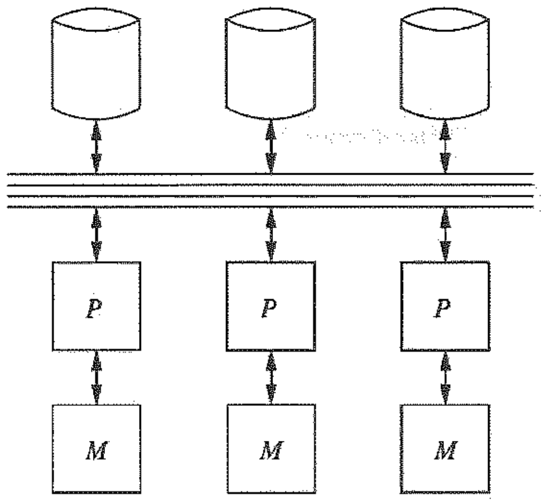

Figure 20.2: A shared-disk machine

Shared-Disk Machines

In this architecture, suggested by Fig. 20.2, every processor has its own memory, which is not accessible directly from other processors. However, the disks are accessible from any of the processors through the communication network. Disk controllers manage the potentially competing requests from different processors.

The number of disks and processors need not be identical, as it might appear from Fig. 20.2.

This architecture today appears in two forms, depending on the units of transfer between the disks and processors. Disk farms called *network attached storage* (NAS) store and transfer files. The alternative, *storage area networks* (SAN) transfer disk blocks to and from the processors.

Shared-Nothing Machines

Here, all processors have their own memory and their own disk or disks, as in Fig. 20.3. All communication is via the network, from processor to processor. For example, if one processor P wants to read tuples from the disk of another processor Q, then processor P sends a message to Q asking for the data. Q obtains the tuples from its disk and ships them over the network in another message, which is received by P.

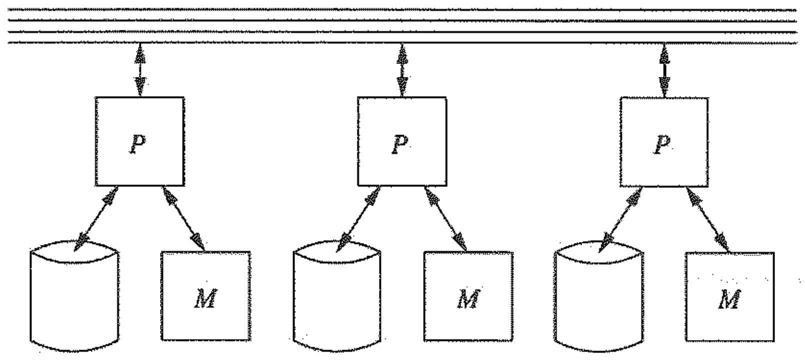

Figure 20.3: A shared-nothing machine

As we mentioned, the shared-nothing architecture is the most commonly used architecture for database systems. Shared-nothing machines are relatively inexpensive to build; one buys racks of commodity machines and connects them with the network connection that is typically built into the rack. Multiple racks can be connected by an external network.

But when we design algorithms for these machines we must be aware that it is costly to send data from one processor to another. Normally, data must be sent between processors in a message, which has considerable overhead associated with it. Both processors must execute a program that supports the message transfer, and there may be contention or delays associated with the communication network as well. Typically, the cost of a message can be broken into a large fixed overhead plus a small amount of time per byte transmitted. Thus, there is a significant advantage to designing a parallel algorithm so that communications between processors involve large amounts of data sent at once. For instance, we might buffer several blocks of data at processor P, all bound for processor Q. If Q does not need the data immediately, it may be much more efficient to wait until we have a long message at P and then send it to

Q. Fortunately, the best known parallel algorithms for database operations can use long messages effectively.

20.1.2 Tuple-at-a-Time Operations in Parallel

Let us begin our discussion of parallel algorithms for a shared-nothing machine by considering the selection operator. First, we must consider how data is best stored. As first suggested by Section 13.3.3, it is useful to distribute our data across as many disks as possible. For convenience, we shall assume there is one disk per processor. Then if there are p processors, divide any relation R's tuples evenly among the p processor's disks.

To compute $\sigma_C(R)$, we may use each processor to examine the tuples of R on its own disk. For each, it finds those tuples satisfying condition C and copies those to the output. To avoid communication among processors, we store those tuples t in $\sigma_C(R)$ at the same processor that has t on its disk. Thus, the result relation $\sigma_C(R)$ is divided among the processors, just like R is.

Since $\sigma_C(R)$ may be the input relation to another operation, and since we want to minimize the elapsed time and keep all the processors busy all the time, we would like $\sigma_C(R)$ to be divided evenly among the processors. If we were doing a projection, rather than a selection, then the number of tuples in $\pi_L(R)$ at each processor would be the same as the number of tuples of R at that processor. Thus, if R is distributed evenly, so would be its projection. However, a selection could radically change the distribution of tuples in the result, compared to the distribution of R.

Example 20.1: Suppose the selection is $\sigma_{a=10}(R)$, that is, find all the tuples of R whose value in the attribute a is 10. Suppose also that we have divided R according to the value of the attribute a. Then all the tuples of R with $a = 10$ are at one processor, and the entire relation $\sigma_{a=10}(R)$ is at one processor. □

To avoid the problem suggested by Example 20.1, we need to think carefully about the policy for partitioning our stored relations among the processors. Probably the best we can do is to use a hash function h that involves all the components of a tuple in such a way that changing one component of a tuple t can change $h(t)$ to be any possible bucket number. For example, if we want B buckets, we might convert each component somehow to an integer between 0 and $B - 1$, add the integers for each component, divide the result by B, and take the remainder as the bucket number. If B is also the number of processors, then we can associate each processor with a bucket and give that processor the contents of its bucket.

20.1.3 Parallel Algorithms for Full-Relation Operations

First, let us consider the operation $\delta(R)$. If we use a hash function to distribute the tuples of R as in Section 20.1.2, then we shall place duplicate tuples of R at the same processor. We can produce $\delta(R)$ in parallel by applying a standard, uniprocessor algorithm (as in Section 15.4.2 or 15.5.2, e.g.) to the portion of R at each processor. Likewise, if we use the same hash function to distribute the tuples of both R and S, then we can take the union, intersection, or difference of R and S by working in parallel on the portions of R and S at each processor.

However, suppose that R and S are not distributed using the same hash function, and we wish to take their union.[1] In this case, we first must make copies of all the tuples of R and S and distribute them according to a single hash function h.[2]

In parallel, we hash the tuples of R and S at each processor, using hash function h. The hashing proceeds as described in Section 15.5.1, but when the buffer corresponding to a bucket i at one processor j is filled, instead of moving it to the disk at j, we ship the contents of the buffer to processor i. If we have room for several blocks per bucket in main memory, then we may wait to fill several buffers with tuples of bucket i before shipping them to processor i.

Thus, processor i receives all the tuples of R and S that belong in bucket i. In the second stage, each processor performs the union of the tuples from R and S belonging to its bucket. As a result, the relation $R \cup S$ will be distributed over all the processors. If hash function h truly randomizes the placement of tuples in buckets, then we expect approximately the same number of tuples of $R \cup S$ to be at each processor.

The operations of intersection and difference may be performed just like a union; it does not matter whether these are set or bag versions of these operations. Moreover:

- To take a join $R(X,Y) \bowtie S(Y,Z)$, we hash the tuples of R and S to a number of buckets equal to the number of processors. However, the hash function h we use must depend only on the attributes of Y, not all the attributes, so that joining tuples are always sent to the same bucket. As with union, we ship tuples of bucket i to processor i. We may then perform the join at each processor using any uniprocessor join algorithm.

- To perform grouping and aggregation $\gamma_L(R)$, we distribute the tuples of R using a hash function h that depends only on the grouping attributes in list L. If each processor has all the tuples corresponding to one of the buckets of h, then we can perform the γ_L operation on these tuples locally, using any uniprocessor γ algorithm.

20.1.4 Performance of Parallel Algorithms

Now, let us consider how the running time of a parallel algorithm on a p-processor machine compares with the time to execute an algorithm for the

[1]In principle, this union could be either a set- or bag-union. But the simple bag-union technique from Section 15.2.3 of copying all the tuples from both arguments works in parallel, so we probably would not want to use the algorithm described here for a bag-union.

[2]If the hash function used to distribute tuples of R or S is known, we can use that hash function for the other and not distribute both relations.

same operation on the same data, using a uniprocessor. The total work — disk I/O's and processor cycles — cannot be smaller for a parallel machine than for a uniprocessor. However, because there are p processors working with p disks, we can expect the elapsed, or wall-clock, time to be much smaller for the multiprocessor than for the uniprocessor.

A unary operation such as $\sigma_C(R)$ can be completed in $1/p$th of the time it would take to perform the operation at a single processor, provided relation R is distributed evenly, as was supposed in Section 20.1.2. The number of disk I/O's is essentially the same as for a uniprocessor selection. The only difference is that there will, on average, be p half-full blocks of R, one at each processor, rather than a single half-full block of R had we stored all of R on one processor's disk.

Now, consider a binary operation, such as join. We use a hash function on the join attributes that sends each tuple to one of p buckets, where p is the number of processors. To distribute the tuples belonging to one processor, we must read each tuple from disk to memory, compute the hash function, and ship all tuples except the one out of p tuples that happens to belong to the bucket at its own processor.

If we are computing $R(X,Y) \bowtie S(Y,Z)$, then we need to do $B(R) + B(S)$ disk I/O's to read all the tuples of R and S and determine their buckets. We then must ship $\big((p-1)/p\big)\big(B(R) + B(S)\big)$ blocks of data across the machine's internal interconnection network to their proper processors; only the $(1/p)$th of the tuples already at the right processor need not be shipped. The cost of shipment can be greater or less than the cost of the same number of disk I/O's, depending on the architecture of the machine. However, we shall assume that shipment across the internal network is significantly cheaper than movement of data between disk and memory, because no physical motion is involved in shipment among processors, while it is for disk I/O.

In principle, we might suppose that the receiving processor has to store the data on its own disk, then execute a local join on the tuples received. For example, if we used a two-pass sort-join at each processor, a naive parallel algorithm would use $3\big(B(R) + B(S)\big)/p$ disk I/O's at each processor, since the sizes of the relations in each bucket would be approximately $B(R)/p$ and $B(S)/p$, and this type of join takes three disk I/O's per block occupied by each of the argument relations. To this cost we would add another $2\big(B(R) + B(S)\big)/p$ disk I/O's per processor, to account for the first read of each tuple and the storing away of each tuple by the processor receiving the tuple during the hash and distribution of tuples. We should also add the cost of shipping the data, but we have elected to consider that cost negligible compared with the cost of disk I/O for the same data.

The above comparison demonstrates the value of the multiprocessor. While we do more disk I/O in total — five disk I/O's per block of data, rather than three — the elapsed time, as measured by the number of disk I/O's performed at each processor has gone down from $3\big(B(R) + B(S)\big)$ to $5\big(B(R) + B(S)\big)/p$, a significant win for large p.

Biiig Mistake

When using hash-based algorithms to distribute relations among processors and to execute operations, as in Example 20.2, we must be careful not to overuse one hash function. For instance, suppose we used a hash function h to hash the tuples of relations R and S among processors, in order to take their join. We might be tempted to use h to hash the tuples of S locally into buckets as we perform a one-pass hash-join at each processor. But if we do so, all those tuples will go to the same bucket, and the main-memory join suggested in Example 20.2 will be extremely inefficient.

Moreover, there are ways to improve the speed of the parallel algorithm so that the total number of disk I/O's is not greater than what is required for a uniprocessor algorithm. In fact, since we operate on smaller relations at each processor, we may be able to use a local join algorithm that uses fewer disk I/O's per block of data. For instance, even if R and S were so large that we need a two-pass algorithm on a uniprocessor, we may be able to use a one-pass algorithm on $(1/p)$th of the data.

We can avoid two disk I/O's per block if, when we ship a block to the processor of its bucket, that processor can use the block immediately as part of its join algorithm. Many algorithms known for join and the other relational operators allow this use, in which case the parallel algorithm looks just like a multipass algorithm in which the first pass uses the hashing technique of Section 15.8.3.

Example 20.2: Consider our running example from Chapter 15 of the join $R(X,Y) \bowtie S(Y,Z)$, where R and S occupy 1000 and 500 blocks, respectively. Now, let there be 101 buffers at each processor of a 10-processor machine. Also, assume that R and S are distributed uniformly among these 10 processors.

We begin by hashing each tuple of R and S to one of 10 "buckets," using a hash function h that depends only on the join attributes Y. These 10 "buckets" represent the 10 processors, and tuples are shipped to the processor corresponding to their "bucket." The total number of disk I/O's needed to read the tuples of R and S is 1500, or 150 per processor. Each processor will have about 15 blocks worth of data for each other processor, so it ships 135 blocks to the other nine processors. The total communication is thus 1350 blocks.

We shall arrange that the processors ship the tuples of S before the tuples of R. Since each processor receives about 50 blocks of tuples from S, it can store those tuples in a main-memory data structure, using 50 of its 101 buffers. Then, when processors start sending R-tuples, each one is compared with the local S-tuples, and any resulting joined tuples are output.

In this way, the only cost of the join is 1500 disk I/O's. Moreover, the elapsed time is primarily the 150 disk I/O's performed at each processor, plus the time to ship tuples between processors and perform the main-memory computations. Note that 150 disk I/O's is less than 1/10th of the time to perform the same algorithm on a uniprocessor; we have not only gained because we had 10 processors working for us, but the fact that there are a total of 1010 buffers among those 10 processors gives us additional efficiency. □

20.1.5 Exercises for Section 20.1

Exercise 20.1.1: Suppose that a disk I/O takes 100 milliseconds. Let $B(R) = 100$, so the disk I/O's for computing $\sigma_C(R)$ on a uniprocessor machine will take about 10 seconds. What is the speedup if this selection is executed on a parallel machine with p processors, where: (a) $p = 8$ (b) $p = 100$ (c) $p = 1000$.

! **Exercise 20.1.2:** In Example 20.2 we described an algorithm that computed the join $R \bowtie S$ in parallel by first hash-distributing the tuples among the processors and then performing a one-pass join at the processors. In terms of $B(R)$ and $B(S)$, the sizes of the relations involved, p (the number of processors), and M (the number of blocks of main memory at each processor), give the condition under which this algorithm can be executed successfully.

20.2 The Map-Reduce Parallelism Framework

Map-reduce is a high-level programming system that allows many important database processes to be written simply. The user writes code for two functions, map and reduce. A master controller divides the input data into chunks, and assigns different processors to execute the map function on each chunk. Other processors, perhaps the same ones, are then assigned to perform the reduce function on pieces of the output from the map function.

20.2.1 The Storage Model

For the map-reduce framework to make sense, we should assume a massively parallel machine, most likely shared-nothing. Typically, the processors are commodity computers, mounted in racks with a simple communication network among the processors on a rank. If there is more than one rack, the racks are also connected by a simple network.

Data is assumed stored in files. Typically, the files are very large compared with the files found in conventional systems. For example, one file might be all the tuples of a very large relation. Or, the file might be a terabyte of "market-baskets," as discussed in Section 22.1.4. For another example of a single file, we shall talk in Section 23.2.2 of the "transition matrix of the Web," which is a representation of the graph with all Web pages as nodes and hyperlinks as edges.

Files are divided into *chunks*, which might be complete cylinders of a disk, and are typically many megabytes. For resiliency, each chunk is replicated several times, so it will not be lost if the disk holding it crashes.

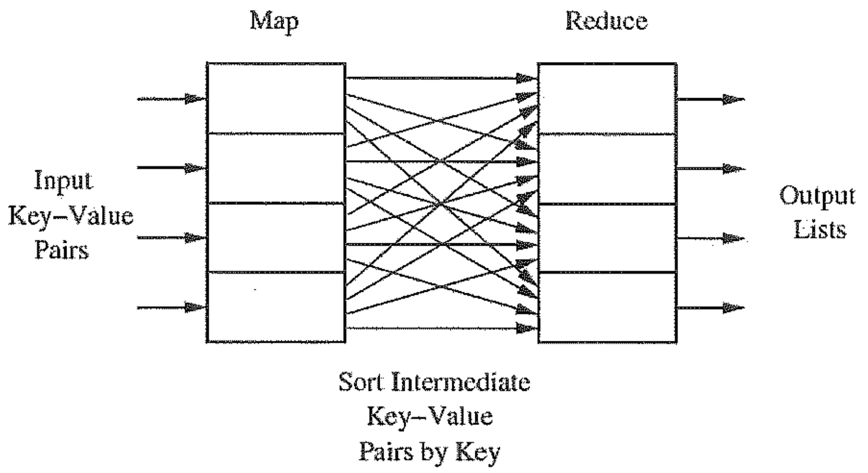

Figure 20.4: Execution of map and reduce functions

20.2.2 The Map Function

The outline of what user-defined map and reduce functions do is suggested in Fig. 20.4. The input is generally thought of as a set of key-value records, although in fact the input could be objects of any type.[3] The function *map* is executed by one or more processes, located at any number of processors. Each map process is given a chunk of the entire input data on which to work.

The map function is designed to take one key-value pair as input and to produce a list of key-value pairs as output. However:

- The types of keys and values for the output of the map function need not be the same as the types of input keys and values.
- The "keys" that are output from the map function are not true keys in the database sense. That is, there can be many pairs with the same key value. However, the key field of output pairs plays a special role in the reduce process to be explained next.

The result of executing all the *map* processes is a collection of key-value pairs called the *intermediate result*. These key-value pairs are the outputs of the map function applied to every input pair. Each pair appears at the processor that generated it. Remember that there may be many *map* processes executing the same algorithm on a different part of the input file at different processors.

[3]As we shall see, the output of a map-reduce algorithm is always a set of key-value pairs. Since it is useful in some applications to compose two or more map-reduce operations, it is conventional to assume that both input and output are sets of key-value pairs.

Example 20.3: We shall consider as an example, constructing an inverted index for words in documents, as was discussed in Section 14.1.8. That is, our input is a collection of documents, and we desire to construct as the final output (not as the output of *map*) a list for each word of the documents that contain that word at least once. The input is a set of pairs each of whose keys are document ID's and whose values are the corresponding documents.

The map function takes a pair consisting of a document ID i and a document d. This function scans d character by character, and for each word w it finds, it emits the pair (w, i). Notice that in the output, the word is the key and the document ID is the associated value. The output of *map* for a single ID-document pair is a list of word-ID pairs. It is not necessary to catch duplicate words in the document; the elimination of duplicates can be done later, at the reduce phase. The intermediate result is the collection of all word-ID pairs created from all the documents in the input database. □

20.2.3 The Reduce Function

The second user-defined function, *reduce*, is also executed by one or more processes, located at any number of processors. The input to *reduce* is a single key value from the intermediate result, together with the list of all values that appear with this key in the intermediate result. Duplicate values are not eliminated.

In Fig. 20.4, we suggest that the output of *map* at each of four processors is distributed to four processors, each of which will execute *reduce* for a subset of the intermediate keys. However, there are a number of ways in which this distribution could be managed. For example, Each *map* process could leave its output on its local disk, and a *reduce* process could retrieve the portion of the intermediate result that it needed, over whatever network or bus interconnects the processors.

The reduce function itself combines the list of values associated with a given key k. The result is k paired with a value of some type. In many simple cases, the reduce function is associative and commutative, and the entire list of values is reduced to a single value of the same type as the list elements. For instance, if *reduce* is addition, the result is the some of a list of numbers.

When *reduce* is associative and commutative, it is possible to speed up the execution of *reduce* by starting to apply its operation to the pairs produced by the *map* processes, even before they finish. Moreover, if a given *map* process produces more than one intermediate pair with the same key, then the reduce operation can be applied on the spot to combine the pairs, without waiting for them to be passed to the *reduce* process for that key.

Example 20.4: Let us consider the reduce function that lets us complete Example 20.3 to produce inverted indexes. The intermediate result consists of pairs of the form $(w, [i_1, i_2, \ldots, i_n])$, where the i's are a list of document ID's, one for each occurrence of word w. The reduce function we need takes a list of ID's, eliminates duplicates, and sorts the list of unique ID's.

Notice how this organization of the computation makes excellent use of whatever parallelism is available. The map function works on a single document, so we could have as many processes and processors as there are documents in the database. The reduce function works on a single word, so we could have as many processes and processors as there are words in the database. Of course, it is unlikely that we would use so many processors in practice. □

Example 20.5: Suppose rather than constructing an inverted index, we want to construct a word count. That is, for each word w that appears at least once in our database of documents, we want our output to have the pair (w, c), where c is the number of times w appears among all the documents. The map function takes an input document, goes through the document character by character, and each time it encounters another word w, it emits the pair $(w, 1)$. The intermediate result is a list of pairs $(w_1, 1), (w_2, 1), \ldots$.

In this example, the reduce function is addition of integers. That is, the input to *reduce* is a pair $(w, [1, 1, \ldots, 1])$, with a 1 for each occurrence of the word w. The reduce function sums the 1's, producing the count. □

Example 20.6: It is a little trickier to express the join of relations in the map-reduce framework. In this simple special case, we shall take the natural join of relations $R(A, B)$ and $S(B, C)$. First, the input to the map function is key-value pairs (x, t), where x is either R or S, and t is a tuple of the relation named by x. The output is a single pair consisting of the join value B taken from the tuple t and a pair consisting of x (to let us remember which relation this tuple came from) and the other component of t, either A (if $x = R$) or C (if $x = S$). All these records of the form $\big(b, (R, a)\big)$ or $\big(b, (S, c)\big)$ form the intermediate result.

The reduce function takes a B-value b, the key, together with a list that consists of pairs of the form (R, a) or (S, c). The result of the join will have as many tuples with B-value b as we can form by pairing an a from an (R, a) element on the list with a c from an (S, c) element on the list. Thus, *reduce* must extract from the list all the A-values associated with R and the list of all C-values associated with S. These are paired in all possible ways, with the b in the middle to form a tuple of the result. □

20.2.4 Exercises for Section 20.2

Exercise 20.2.1: Modify Example 20.5 to count the number of documents in which each word w appears.

Exercise 20.2.2: Express, in the map-reduce framework, the following operations on relations: (a) σ_C (b) π_L (c) $R \bowtie_C S$ (d) $R \cup S$ (e) $R \cap S$.

20.3 Distributed Databases

We shall now consider the elements of distributed database systems. In a distributed system, there are many, relatively autonomous processors that may participate in database operations. The difference between a distributed system and a shared-nothing parallel system is in the assumption about the cost of communication. Shared-nothing parallel systems usually have a message-passing cost that is small compared with disk accesses and other costs. In a distributed system, the processors are typically physically distant, rather than in the same room. The network connecting processors may have much less capacity than the network in a shared-nothing system.

Distributed databases offer significant advantages. Like parallel systems, a distributed system can use many processors and thereby accelerate the response to queries. Further, since the processors are widely separated, we can increase resilience in the face of failures by replicating data at several sites.

On the other hand, distributed processing increases the complexity of every aspect of a database system, so we need to rethink how even the most basic components of a DBMS are designed. Since the cost of communicating may dominate the cost of processing in main memory, a critical issue is how many messages are sent between sites. In this section we shall introduce the principal issues, while the next sections concentrate on solutions to two important problems that come up in distributed databases: distributed commit and distributed locking.

20.3.1 Distribution of Data

One important reason to distribute data is that the organization is itself distributed among many sites, and the sites each have data that is germane primarily to that site. Some examples are:

1. A bank may have many branches. Each branch (or the group of branches in a given city) will keep a database of accounts maintained at that branch (or city). Customers can choose to bank at any branch, but will normally bank at "their" branch, where their account data is stored. The bank may also have data that is kept in the central office, such as employee records and policies such as current interest rates. Of course, a backup of the records at each branch is also stored, probably in a site that is neither a branch office nor the central office.

2. A chain of department stores may have many individual stores. Each store (or a group of stores in one city) has a database of sales at that store and inventory at that store. There may also be a central office with data about employees, a chain-wide inventory, data about credit-card customers, and information about suppliers such as unfilled orders, and what each is owed. In addition, there may be a copy of all the stores'

sales data in a data warehouse that is used to analyze and predict sales through ad-hoc queries issued by analysts.

3. A digital library may consist of a consortium of universities that each hold on-line books and other documents. Search at any site will examine the catalog of documents available at all sites and deliver an electronic copy of the document to the user if any site holds it.

In some cases, what we might think of logically as a single relation has been partitioned among many sites. For example, the chain of stores might be imagined to have a single sales relation, such as

```
Sales(item, date, price, purchaser)
```

However, this relation does not exist physically. Rather, it is the union of a number of relations with the same schema, one at each of the stores in the chain. These local relations are called *fragments*, and the partitioning of a logical relation into physical fragments is called *horizontal decomposition* of the relation `Sales`. We regard the partition as "horizontal" because we may visualize a single `Sales` relation with its tuples separated, by horizontal lines, into the sets of tuples at each store.

In other situations, a distributed database appears to have partitioned a relation "vertically," by decomposing what might be one logical relation into two or more, each with a subset of the attributes, and with each relation at a different site. For instance, if we want to find out which sales at the Boston store were made to customers who are more than 90 days in arrears on their credit-card payments, it would be useful to have a relation (or view) that included the item, date, and purchaser information from `Sales`, along with the date of the last credit-card payment by that purchaser. However, in the scenario we are describing, this relation is decomposed vertically, and we would have to join the credit-card-customer relation at the central headquarters with the fragment of `Sales` at the Boston store.

20.3.2 Distributed Transactions

A consequence of the distribution of data is that a transaction may involve processes at several sites. Thus, our model of what a transaction is must change. No longer is a transaction a piece of code executed by a single processor communicating with a single scheduler and a single log manager at a single site. Rather, a transaction consists of communicating *transaction components*, each at a different site and communicating with the local scheduler and logger. Two important issues that must thus be looked at anew are:

1. How do we manage the commit/abort decision when a transaction is distributed? What happens if one component of the transaction wants to abort the whole transaction, while others encountered no problem and

want to commit? We discuss a technique called "two-phase commit" in Section 20.5; it allows the decision to be made properly and also frequently allows sites that are up to operate even if some other site(s) have failed.

2. How do we assure serializability of transactions that involve components at several sites? We look at locking in particular, in Section 20.6 and see how local lock tables can be used to support global locks on database elements and thus support serializability of transactions in a distributed environment.

20.3.3 Data Replication

One important advantage of a distributed system is the ability to *replicate* data, that is, to make copies of the data at different sites. One motivation is that if a site fails, there may be other sites that can provide the same data that was at the failed site. A second use is in improving the speed of query answering by making a copy of needed data available at the sites where queries are initiated. For example:

1. A bank may make copies of current interest-rate policy available at each branch, so a query about rates does not have to be sent to the central office.

2. A chain store may keep copies of information about suppliers at each store, so local requests for information about suppliers (e.g., the manager needs the phone number of a supplier to check on a shipment) can be handled without sending messages to the central office.

3. A digital library may temporarily cache a copy of a popular document at a school where students have been assigned to read the document.

However, there are several problems that must be faced when data is replicated.

a) How do we keep copies identical? In essence, an update to a replicated data element becomes a distributed transaction that updates all copies.

b) How do we decide where and how many copies to keep? The more copies, the more effort is required to update, but the easier queries become. For example, a relation that is rarely updated might have copies everywhere for maximum efficiency, while a frequently updated relation might have only one copy and a backup.

c) What happens when there is a communication failure in the network, and different copies of the same data have the opportunity to evolve separately and must then be reconciled when the network reconnects?

20.3.4 Exercises for Section 20.3

!! **Exercise 20.3.1:** The following exercise will allow you to address some of the problems that come up when deciding on a replication strategy for data. Suppose there is a relation R that is accessed from n sites. The ith site issues q_i queries about R and u_i updates to R per second, for $i = 1, 2, \ldots, n$. The cost of executing a query if there is a copy of R at the site issuing the query is c, while if there is no copy there, and the query must be sent to some remote site, then the cost is $10c$. The cost of executing an update is d for the copy of R at the issuing site and $10d$ for every copy of R that is not at the issuing site. As a function of these parameters, how would you choose, for large n, a set of sites at which to replicate R.

20.4 Distributed Query Processing

We now turn to optimizing queries on a network of distributed machines. When communication among processors is a significant cost, there are some query plans that can be more efficient than the ones we developed in Section 20.1 for processors that could communicate locally. Our principal objective is a new way of computing joins, using the semijoin operator that was introduced in Exercise 2.4.8.

20.4.1 The Distributed Join Problem

Suppose we want to compute $R(A, B) \bowtie S(B, C)$. However, R and S reside at different nodes of a network, as suggested in Fig. 20.5. There are two obvious ways to compute the join.

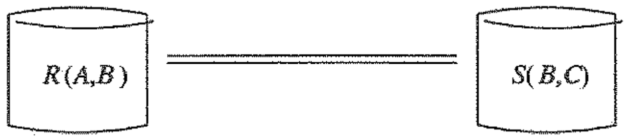

Figure 20.5: Joining relations at different nodes of a network

1. Send a copy of R to the site of S, and compute the join there.
2. Send a copy of S to the site of R and compute the join there.

In many situations, either of these methods is fine. However, problems can arise, such as:

a) What happens if the channel between the sites has low-capacity, e.g., a phone line or wireless link? Then, the cost of the join is primarily the time it takes to copy one of the relations, so we need to design our query plan to minimize communication.

b) Even if communication is fast, there may be a better query plan if the shared attribute B has values that are much smaller than the values of A and C. For example, B could be an identifier for documents or videos, while A and C are the documents or videos themselves.

20.4.2 Semijoin Reductions

Both these problems can be dealt with using the same type of query plan, in which only the relevant part of each relation is shipped to the site of the other. Recall that the semijoin of relations $R(X,Y)$ and $S(Y,Z)$, where X, Y, and Z are sets of attributes, is $R \ltimes S = R \bowtie \big(\pi_Y(S)\big)$. That is, we project S onto the common attributes, and then take the natural join of that projection with R. $\pi_Y(S)$ is a set-projection, so duplicates are eliminated. It is unusual to take a natural join where the attributes of one argument are a subset of the attributes of the other, but the definition of the join covers this case. The effect is that $R \ltimes S$ contains all those tuples of R that join with at least one tuple of S. Put another way, the semijoin $R \ltimes S$ eliminates the dangling tuples of R.

Having sent $\pi_Y(S)$ to the site of R, we can compute $R \ltimes S$ there. We know those tuples of R that are not in $R \ltimes S$ cannot participate in $R \bowtie S$. Therefore it is sufficient to send $R \ltimes S$, rather than all of R, to the site of S and to compute the join there. This plan is suggested by Fig. 20.6 for the relations $R(A,B)$ and $S(B,C)$. Of course there is a symmetric plan where the roles of R and S are interchanged.

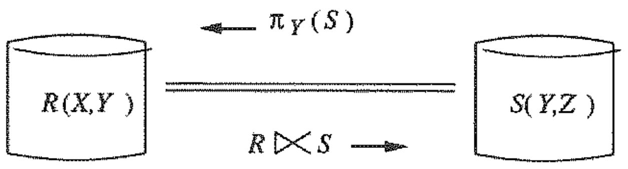

Figure 20.6: Exploiting the semijoin to minimize communication

Whether this semijoin plan, or the plan with R and S interchanged is more efficient than one of the obvious plans depends on several factors. First, if the projection of S onto Y results in a relation much smaller than S, then it is cheaper to send $\pi_Y(S)$ to the site of R, rather than S itself. $\pi_Y(S)$ will be small compared with S if either or both of the following hold:

1. There are many duplicates to be eliminated; i.e., many tuples of S share Y-values.
2. The components for the attributes of Z are large compared with the components of Y; e.g., Z includes attributes whose values are audios, videos, or documents.

In order for the semijoin plan to be superior, we also need to know that the size of $R \ltimes S$ is smaller than R. That is, R must contain many dangling tuples in its join with S.

20.4.3 Joins of Many Relations

When we want to take the natural join of two relations, only one semijoin is useful. The same holds for an equijoin, since we can act as if the equated pairs of attributes had the same name and treat the equijoin as if it were a natural join. However, when we take the natural join or equijoin of three or more relations at different sites, several surprising things happen.

- We may need several semijoins to eliminate all the dangling tuples from the relations before shipping them to other sites for joining.
- There are sets of relation schemas such that no finite sequence of semijoins eliminates all dangling tuples.
- It is possible to identify those sets of relation schemas such that there is a finite way to eliminate dangling tuples by semijoins.

Example 20.7: To see what can go wrong when we take the natural join of more than two relations, consider $R(A, B)$, $S(B, C)$, and $T(C, A)$. Suppose R and S have exactly the same n tuples: $\{(1,1),\ (2,2), \ldots, (n,n)\}$. T has $n-1$ tuples: $\{(1,2),\ (2,3), \ldots, (n-1,n)\}$. The relations are shown in Fig. 20.7.

A	B	B	C	C	A
1	1	1	1	1	2
2	2	2	2	2	3
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
n	n	n	n	$n-1$	n
R		S		T	

Figure 20.7: Three relations for which elimination of dangling tuples by semijoins is very slow

Notice that while R and S join to produce the n tuples

$$\{(1,1,1),\ (2,2,2), \ldots, (n,n,n)\}$$

none of these tuples can join with any tuple of T. The reason is that all tuples of $R \bowtie S$ agree in their A and C components, while the tuples of T disagree in their A and C components. That is, $R \bowtie S \bowtie T$ is empty, and *all* tuples of each relation are dangling.

However, no one semijoin can eliminate more than one tuple from any relation. For example, $S \ltimes T$ eliminates only (n,n) from S, because $\pi_C(T) = \{1, 2, \ldots, n-1\}$. Similarly, $R \ltimes T$ eliminates only $(1,1)$ from R, because $\pi_A(T) = \{2, 3, \ldots, n\}$. We can then continue, say, with $R \ltimes S$, which eliminates (n,n) from R, and $T \ltimes R$, which eliminates $(n-1,n)$ from T. Now we can compute $S \ltimes T$ again and eliminate $(n-1, n-1)$ from S, and so on. While we shall not prove it, we in fact need $3n-1$ semijoins to make all three relations empty. □

Since n in Example 20.7 is arbitrary, we see that for the particular relations discussed there, no fixed, finite sequence of semijoins is guaranteed to eliminate all dangling tuples, regardless of the data currently held in the relations. On the other hand, as we shall see, many typical joins of three or more relations do have fixed, finite sequences of semijoins that are guaranteed to eliminate all the dangling tuples. We call such a sequence of semijoins a ***full reducer*** for the relations in question.

20.4.4 Acyclic Hypergraphs

Let us assume that we are taking a natural join of several relations, although as mentioned, we can also handle equijoins by pretending the names of equated attributes from different relations are the same, and renaming attributes to make that pretense a reality. If we do, then we can draw a useful picture of every natural join as a ***hypergraph***, that is a set of nodes with ***hyperedges*** that are sets of nodes. A traditional graph is then a hypergraph all of whose hyperedges are sets of size two.

The hypergraph for a natural join is formed by creating one node for each attribute name. Each relation is represented by a hyperedge containing all of its attributes.

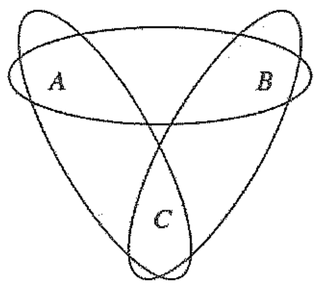

Figure 20.8: The hypergraph for Example 20.7

Example 20.8: Figure 20.8 is the hypergraph for the three relations from Example 20.7. The relation $R(A, B)$ is represented by the hyperedge $\{A, B\}$; S is represented by the hyperedge $\{B, C\}$, and T is the hyperedge $\{A, C\}$. Notice that this hypergraph is actually a graph, since the hyperedges are each pairs of nodes. Also observe that the three hyperedges form a cycle in the graph. As we shall see, it is this cyclicity that causes there to be no full reducer.

However, the question of when a hypergraph is cyclic has a somewhat unintuitive answer. In Fig. 20.9 is another hypergraph, which could be used, for instance, to represent the join of the relations $R(A, E, F)$, $S(A, B, C)$, $T(C, D, E)$, and $U(A, C, E)$. This hypergraph is a true hypergraph, since it has hyperedges with more than two nodes. It also happens to be an "acyclic" hypergraph, even though it appears to have cycles. □

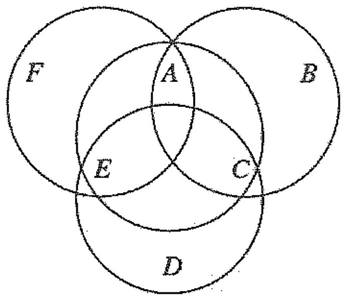

Figure 20.9: An acyclic hypergraph

To define acyclic hypergraphs correctly, and thus get the condition under which a full reducer exists, we first need the notion of an "ear" in a hypergraph. A hyperedge H is an *ear* if there is some other hyperedge G in the same hypergraph such that every node of H is either:

1. Found only in H, or

2. Also found in G.

We shall say that G *consumes* H, for a reason that will become apparent when we discuss reduction of the hypergraph.

Example 20.9: In Fig. 20.9, hyperedge $H = \{A, E, F\}$ is an ear. The role of G is played by $\{A, C, E\}$. Node F is unique to H; it appears in no other hyperedge. The other two nodes of H (A and E) are also members of G. □

A hypergraph is *acyclic* if it can be reduced to a single hyperedge by a sequence of *ear reductions*. An ear reduction is simply the elimination of one ear from the hypergraph, along with any nodes that appear only in that ear. Note that an ear, if not eliminated at one step, remains an ear after another ear is eliminated. However, it is possible that a hyperedge that was not an ear, becomes an ear after another hyperedge is eliminated.

Example 20.10: Figure 20.8 is not acyclic. No hyperedge is an ear, so we cannot get started with any ear reduction. For example, $\{A, B\}$ is not an ear because neither A nor B is unique to this hyperedge, and no other hyperedge contains both A and B.

On the other hand, Fig. 20.9 is acyclic. As we mentioned in Example 20.9, $\{A, E, F\}$ is an ear; so are $\{A, B, C\}$ and $\{C, D, E\}$. We can therefore eliminate hyperedge $\{A, E, F\}$ from the hypergraph. When we eliminate this ear, node F

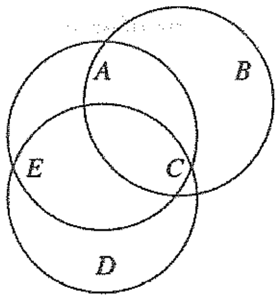

Figure 20.10: After one ear reduction

disappears, but the other five nodes and three hyperedges remain, as suggested in Fig. 20.10.

Since $\{A, B, C\}$ is an ear in Fig. 20.10, we may eliminate it and node B in a second ear reduction. Now, we are left with only hyperedges $\{A, C, E\}$ and $\{C, D, E\}$. Each is now an ear; notice that $\{A, C, E\}$ was not an ear until now. We can eliminate either, leaving a single hyperedge and proving that Fig. 20.9 is an acyclic hypergraph. □

20.4.5 Full Reducers for Acyclic Hypergraphs

We can construct a full reducer for any acyclic hypergraph by following the sequence of ear reductions. We construct the sequence of semijoins as follows, by induction on the number of hyperedges in an acyclic hypergraph.

BASIS: If there is only one hyperedge, do nothing. The "join" of one relation is the relation itself, and there are surely no dangling tuples.

INDUCTION: If the acyclic hypergraph has more than one hyperedge, then it must have at least one ear. Pick one, say H, and suppose it is consumed by hyperedge G.

1. Execute the semijoin $G := G \ltimes H$; that is, eliminate from G any of its tuples that do not join with H.[4]

2. Recursively, find a semijoin sequence for the hypergraph with ear H eliminated.

3. Execute the semijoin $H := H \ltimes G$.

Example 20.11 : Let us construct the full reducer for the relations $R(A, E, F)$, $S(A, B, C)$, $T(C, D, E)$, and $U(A, C, E)$, whose hypergraph we saw in Fig. 20.9.

[4]We are identifying hyperedges with the relations that they represent for convenience in notation. Moreover, if the sets of tuples corresponding to a hyperedge are stored tables, rather than temporary relations, we do not actually replace a relation by a semijoin, as would be suggested by a step like $G := G \ltimes H$, but instead we store the result in a new temporary, G'.

We shall use the sequence of ears R, then S, then U, as in Example 20.10. Since U consumes R, we begin with the semijoin $U := U \ltimes R$.

Recursively, we reduce the remaining three hyperedges. That reduction starts with U consuming S, so the next step is $U := U \ltimes S$. Another level of recursion has T consuming U, so we add the step $T := T \ltimes U$. With only T remaining, we have the basis case and do nothing.

Finally, we complete the elimination of ear U by adding $U := U \ltimes T$. Then, we complete the elimination of S by adding $S := S \ltimes U$, and we complete the elimination of R with $R := R \ltimes U$. The entire sequence of semijoins that forms a full reducer for Fig. 20.9 is shown in Fig. 20.11. □

$$
\begin{aligned}
U &:= U \ltimes R \\
U &:= U \ltimes S \\
T &:= T \ltimes U \\
U &:= U \ltimes T \\
S &:= S \ltimes U \\
R &:= R \ltimes U
\end{aligned}
$$

Figure 20.11: A full reducer for Fig. 20.9

Once we have executed all the semijoins in the full reducer, we can copy all the reduced relations to the site of one of them, knowing that the relations to be shipped contain no dangling tuples and therefore are as small as can be. In fact, if we know at which site the join will be performed, then we do not have to eliminate all dangling tuples for relations at that site. We can stop applying semijoins to a relation as soon as that relation will no longer be used to reduce other relations.

Example 20.12: If the full reducer of Fig. 20.11 will be followed by a join at the site of S, then we do not have to do the step $S := S \ltimes U$. However, if the join is to be conducted at the site of T, then we still have to do the reduction $T := T \ltimes U$, because T is used to reduce other relations at later steps. □

20.4.6 Why the Full-Reducer Algorithm Works

We can show that the algorithm produces a full reducer for any acyclic hypergraph by induction on the number of hyperedges.

BASIS: One hyperedge. There are no dangling tuples, so nothing needs to be done.

INDUCTION: When we eliminate the ear H, we eliminate, from the hyperedge G that consumes H, all tuples that will not join with at least one tuple of H. Thus, whatever further reductions are done, the join of the relations for all the hyperedges besides H cannot contain a tuple that will not join with H.

Note that this statement is true because G is the only link between H and the remaining relations.

By induction, all tuples that are dangling in the join of the remaining relations are eliminated. When we do the final semijoin $H := H \ltimes G$ to eliminate dangling tuples from H, we know that no relation has dangling tuples.

20.4.7 Exercises for Section 20.4

! **Exercise 20.4.1:** Suppose we want to take the natural join of $R(A,B)$ and $S(B,C)$, where R and S are at different sites, and the size of the data communicated is the dominant cost of the join. Suppose the sizes of R and S are s_R and s_S, respectively. Suppose that the size of $\pi_B(R)$ is fraction p_R of the size of R and $\pi_B(S)$ is fraction p_S of the size of S. Finally, suppose that fractions d_R and d_S of relations R and S, respectively, are dangling. Write expressions, in terms of these six parameters, for the costs of the four strategies for evaluating $R \bowtie S$, and determine the conditions under which each is the best strategy. The four strategies are:

i) Ship R to the site of S.

ii) Ship S to the site of R.

iii) Ship $\pi_B(S)$ to the site of R, and then $R \ltimes S$ to the site of S.

iv) Ship $\pi_B(R)$ to the site of S, and then $S \ltimes R$ to the site of R.

Exercise 20.4.2: Determine which of the following hypergraphs are acyclic. Each hypergraph is represented by a list of its hyperedges.

a) $\{A,B\}$, $\{B,C,D\}$, $\{B,E,F\}$, $\{F,G,H\}$, $\{G,I\}$, $\{H,J\}$.

b) $\{A,B\}$, $\{B,C,D\}$, $\{B,E,F\}$, $\{F,G,H\}$, $\{G,I\}$, $\{B,H\}$.

c) $\{A,B,C,D\}$, $\{A,B,E\}$, $\{B,D,F\}$, $\{C,D,G\}$, $\{A,C,H\}$.

Exercise 20.4.3: For those hypergraphs of Exercise 20.4.2 that are acyclic, construct a full reducer.

! **Exercise 20.4.4:** Besides the full reducer of Example 20.11, how many other full reducers of six steps can be constructed for the hypergraph of Fig. 20.9 by choosing other orders for the elimination of ears?

! **Exercise 20.4.5:** A well known property of acyclic graphs is that if you delete an edge from an acyclic graph it remains acyclic. Is the analogous statement true for hypergraphs? That is, if you eliminate a hyperedge from an acyclic hypergraph, is the remaining hypergraph always acyclic? *Hint*: consider the acyclic hypergraph of Fig. 20.9.

!! **Exercise 20.4.6:** Not all binary operations on relations located at different nodes of a network can have their execution time reduced by preliminary operations like the semijoin. Is it possible to improve on the obvious algorithm (ship one of the relations to the other site) when the operation is (a) union (b) intersection (c) difference?

20.5 Distributed Commit

In this section, we shall address the problem of how a distributed transaction that has components at several sites can execute atomically. The next section discusses another important property of distributed transactions: executing them serializably.

20.5.1 Supporting Distributed Atomicity

We shall begin with an example that illustrates the problems that might arise.

Example 20.13: Consider our example of a chain of stores mentioned in Section 20.3. Suppose a manager of the chain wants to query all the stores, find the inventory of toothbrushes at each, and issue instructions to move toothbrushes from store to store in order to balance the inventory. The operation is done by a single global transaction T that has component T_i at the ith store and a component T_0 at the office where the manager is located. The sequence of activities performed by T are summarized below:

1. Component T_0 is created at the site of the manager.
2. T_0 sends messages to all the stores instructing them to create components T_i.
3. Each T_i executes a query at store i to discover the number of toothbrushes in inventory and reports this number to T_0.
4. T_0 takes these numbers and determines, by some algorithm we do not need to discuss, what shipments of toothbrushes are desired. T_0 then sends messages such as "store 10 should ship 500 toothbrushes to store 7" to the appropriate stores (stores 7 and 10 in this instance).
5. Stores receiving instructions update their inventory and perform the shipments.

□

There are a number of things that could go wrong in Example 20.13, and many of these result in violations of the atomicity of T. That is, some of the actions comprising T get executed, but others do not. Mechanisms such as logging and recovery, which we assume are present at each site, will assure that each T_i is executed atomically, but do not assure that T itself is atomic.

Example 20.14: Suppose a bug in the algorithm to redistribute toothbrushes might cause store 10 to be instructed to ship more toothbrushes than it has. T_{10} will therefore abort, and no toothbrushes will be shipped from store 10; neither will the inventory at store 10 be changed. However, T_7 detects no problems and commits at store 7, updating its inventory to reflect the supposedly shipped toothbrushes. Now, not only has T failed to execute atomically (since T_{10} never completes), but it has left the distributed database in an inconsistent state. □

Another source of problems is the possibility that a site will fail or be disconnected from the network while the distributed transaction is running.

Example 20.15: Suppose T_{10} replies to T_0's first message by telling its inventory of toothbrushes. However, the machine at store 10 then crashes, and the instructions from T_0 are never received by T_{10}. Can distributed transaction T ever commit? What should T_{10} do when its site recovers? □

20.5.2 Two-Phase Commit

In order to avoid the problems suggested in Section 20.5.1, distributed DBMS's use a complex protocol for deciding whether or not to commit a distributed transaction. In this section, we shall describe the basic idea behind these protocols, called *two-phase commit*.[5] By making a global decision about committing, each component of the transaction will commit, or none will. As usual, we assume that the atomicity mechanisms at each site assure that either the local component commits or it has no effect on the database state at that site; i.e., components of the transaction are atomic. Thus, by enforcing the rule that either all components of a distributed transaction commit or none does, we make the distributed transaction itself atomic.

Several salient points about the two-phase commit protocol follow:

- In a two-phase commit, we assume that each site logs actions at that site, but there is no global log.

- We also assume that one site, called the *coordinator*, plays a special role in deciding whether or not the distributed transaction can commit. For example, the coordinator might be the site at which the transaction originates, such as the site of T_0 in the examples of Section 20.5.1.

- The two-phase commit protocol involves sending certain messages between the coordinator and the other sites. As each message is sent, it is logged at the sending site, to aid in recovery should it be necessary.

With these points in mind, we can describe the two phases in terms of the messages sent between sites.

[5]Do not confuse two-phase commit with two-phase locking. They are independent ideas, designed to solve different problems.

Phase I

In phase 1 of the two-phase commit, the coordinator for a distributed transaction T decides when to attempt to commit T. Presumably the attempt to commit occurs after the component of T at the coordinator site is ready to commit, but in principle the steps must be carried out even if the coordinator's component wants to abort (but with obvious simplifications as we shall see). The coordinator polls the sites of all components of the transaction T to determine their wishes regarding the commit/abort decision, as follows:

1. The coordinator places a log record `<Prepare` T`>` on the log at its site.

2. The coordinator sends to each component's site (in principle including itself) the message `prepare` T.

3. Each site receiving the message `prepare` T decides whether to commit or abort its component of T. The site can delay if the component has not yet completed its activity, but must eventually send a response.

4. If a site wants to commit its component, it must enter a state called *precommitted*. Once in the precommitted state, the site cannot abort its component of T without a directive to do so from the coordinator. The following steps are done to become precommitted:

   (a) Perform whatever steps are necessary to be sure the local component of T will not have to abort, even if there is a system failure followed by recovery at the site. Thus, not only must all actions associated with the local T be performed, but the appropriate actions regarding the log must be taken so that T will be redone rather than undone in a recovery. The actions depend on the logging method, but surely the log records associated with actions of the local T must be flushed to disk.

   (b) Place the record `<Ready` T`>` on the local log and flush the log to disk.

   (c) Send to the coordinator the message `ready` T.

   However, the site does not commit its component of T at this time; it must wait for phase 2.

5. If, instead, the site wants to abort its component of T, then it logs the record `<Don't commit` T`>` and sends the message `don't commit` T to the coordinator. It is safe to abort the component at this time, since T will surely abort if even one component wants to abort.

The messages of phase 1 are summarized in Fig. 20.12.

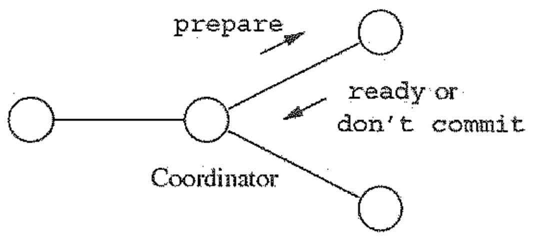

Figure 20.12: Messages in phase 1 of two-phase commit

Phase II

The second phase begins when responses `ready` or `don't commit` are received from each site by the coordinator. However, it is possible that some site fails to respond; it may be down, or it has been disconnected by the network. In that case, after a suitable timeout period, the coordinator will treat the site as if it had sent `don't commit`.

1. If the coordinator has received `ready` T from all components of T, then it decides to commit T. The coordinator logs <`Commit` T> at its site and then sends message `commit` T to all sites involved in T.
2. However, if the coordinator has received `don't commit` T from one or more sites, it logs <`Abort` T> at its site and then sends `abort` T messages to all sites involved in T.
3. If a site receives a `commit` T message, it commits the component of T at that site, logging <`Commit` T> as it does.
4. If a site receives the message `abort` T, it aborts T and writes the log record <`Abort` T>.

The messages of phase 2 are summarized in Fig. 20.13.

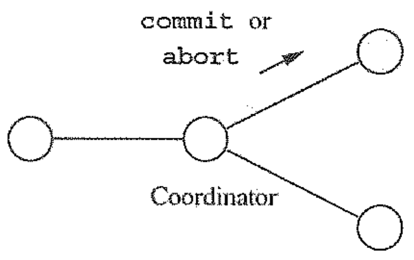

Figure 20.13: Messages in phase 2 of two-phase commit

20.5.3 Recovery of Distributed Transactions

At any time during the two-phase commit process, a site may fail. We need to make sure that what happens when the site recovers is consistent with the global decision that was made about a distributed transaction T. There are several cases to consider, depending on the last log entry for T.

1. If the last log record for T was `<Commit` T`>`, then T must have been committed by the coordinator. Depending on the log method used, it may be necessary to redo the component of T at the recovering site.

2. If the last log record is `<Abort` T`>`, then similarly we know that the global decision was to abort T. If the log method requires it, we undo the component of T at the recovering site.

3. If the last log record is `<Don't commit` T`>`, then the site knows that the global decision must have been to abort T. If necessary, effects of T on the local database are undone.

4. The hard case is when the last log record for T is `<Ready` T`>`. Now, the recovering site does not know whether the global decision was to commit or abort T. This site must communicate with at least one other site to find out the global decision for T. If the coordinator is up, the site can ask the coordinator. If the coordinator is not up at this time, some other site may be asked to consult its log to find out what happened to T. In the worst case, no other site can be contacted, and the local component of T must be kept active until the commit/abort decision is determined.

5. It may also be the case that the local log has no records about T that come from the actions of the two-phase commit protocol. If so, then the recovering site may unilaterally decide to abort its component of T, which is consistent with all logging methods. It is possible that the coordinator already detected a timeout from the failed site and decided to abort T. If the failure was brief, T may still be active at other sites, but it will never be inconsistent if the recovering site decides to abort its component of T and responds with `don't commit` T if later polled in phase 1.

The above analysis assumes that the failed site is not the coordinator. When the coordinator fails during a two-phase commit, new problems arise. First, the surviving participant sites must either wait for the coordinator to recover or elect a new coordinator. Since the coordinator could be down for an indefinite period, there is good motivation to elect a new leader, at least after a brief waiting period to see if the coordinator comes back up.

The matter of *leader election* is in its own right a complex problem of distributed systems, beyond the scope of this book. However, a simple method will work in most situations. For instance, we may assume that all participant sites have unique identifying numbers, e.g., IP addresses. Each participant sends messages announcing its availability as leader to all the other sites, giving its identifying number. After a suitable length of time, each participant acknowledges as the new coordinator the lowest-numbered site from which it has heard, and sends messages to that effect to all the other sites. If all sites receive consistent messages, then there is a unique choice for new coordinator, and everyone knows about it. If there is inconsistency, or a surviving site has failed to respond, that too will be universally known, and the election starts over.

Now, the new leader polls the sites for information about each distributed transaction T. Each site reports the last record on its log concerning T, if there is one. The possible cases are:

1. Some site has `<Commit` T`>` on its log. Then the original coordinator must have wanted to send `commit` T messages everywhere, and it is safe to commit T.

2. Similarly, if some site has `<Abort` T`>` on its log, then the original coordinator must have decided to abort T, and it is safe for the new coordinator to order that action.

3. Suppose now that no site has `<Commit` T`>` or `<Abort` T`>` on its log, but at least one site does *not* have `<Ready` T`>` on its log. Then since actions are logged before the corresponding messages are sent, we know that the old coordinator never received `ready` T from this site and therefore could not have decided to commit. It is safe for the new coordinator to decide to abort T.

4. The most problematic situation is when there is no `<Commit` T`>` or `<Abort` T`>` to be found, but every surviving site has `<Ready` T`>`. Now, we cannot be sure whether the old coordinator found some reason to abort T or not; it could have decided to do so because of actions at its own site, or because of a `don't commit` T message from another failed site, for example. Or the old coordinator may have decided to commit T and already committed its local component of T. Thus, the new coordinator is not able to decide whether to commit or abort T and must wait until the original coordinator recovers. In real systems, the database administrator has the ability to intervene and manually force the waiting transaction components to finish. The result is a possible loss of atomicity, but the person executing the blocked transaction will be notified to take some appropriate compensating action.

20.5.4 Exercises for Section 20.5

! **Exercise 20.5.1:** Consider a transaction T initiated at a home computer that asks bank B to transfer \$10,000 from an account at B to an account at another bank C.

a) What are the components of distributed transaction T? What should the components at B and C do?

b) What can go wrong if there is not \$10,000 in the account at B?

c) What can go wrong if one or both banks' computers crash, or if the network is disconnected?

d) If one of the problems suggested in (c) occurs, how could the transaction resume correctly when the computers and network resume operation?

Exercise 20.5.2: In this exercise, we need a notation for describing sequences of messages that can take place during a two-phase commit. Let (i, j, M) mean that site i sends the message M to site j, where the value of M and its meaning can be P (prepare), R (ready), D (don't commit), C (commit), or A (abort). We shall discuss a simple situation in which site 0 is the coordinator, but not otherwise part of the transaction, and sites 1 and 2 are the components. For instance, the following is one possible sequence of messages that could take place during a successful commit of the transaction:

$$(0,1,P),\ (0,2,P),\ (2,0,R),\ (1,0,R),\ (0,2,C),\ (0,1,C)$$

a) Give an example of a sequence of messages that could occur if site 1 wants to commit and site 2 wants to abort.

! b) How many possible sequences of messages such as the above are there, if the transaction successfully commits?

! c) If site 1 wants to commit, but site 2 does not, how many sequences of messages are there, assuming no failures occur?

! d) If site 1 wants to commit, but site 2 is down and does not respond to messages, how many sequences are there?

!! **Exercise 20.5.3:** Using the notation of Exercise 20.5.2, suppose the sites are a coordinator and n other sites that are the transaction components. As a function of n, how many sequences of messages are there if the transaction successfully commits?

20.6 Distributed Locking

In this section we shall see how to extend a locking scheduler to an environment where transactions are distributed and consist of components at several sites. We assume that lock tables are managed by individual sites, and that the component of a transaction at a site can request locks on the data elements only at that site.

When data is replicated, we must arrange that the copies of a single element X are changed in the same way by each transaction. This requirement introduces a distinction between locking the *logical* database element X and locking one or more of the copies of X. In this section, we shall offer a cost model for distributed locking algorithms that applies to both replicated and nonreplicated data. However, before introducing the model, let us consider an obvious (and sometimes adequate) solution to the problem of maintaining locks in a distributed database — centralized locking.

20.6.1 Centralized Lock Systems

Perhaps the simplest approach is to designate one site, the *lock site*, to maintain a lock table for logical elements, whether or not they have copies at that site. When a transaction wants a lock on logical element X, it sends a request to the lock site, which grants or denies the lock, as appropriate. Since obtaining a global lock on X is the same as obtaining a local lock on X at the lock site, we can be sure that global locks behave correctly as long as the lock site administers locks conventionally. The usual cost is three messages per lock (request, grant, and release), unless the transaction happens to be running at the lock site.

The use of a single lock site can be adequate in some situations, but if there are many sites and many simultaneous transactions, the lock site could become a bottleneck. Further, if the lock site crashes, no transaction at any site can obtain locks. Because of these problems with centralized locking, there are a number of other approaches to maintaining distributed locks, which we shall introduce after discussing how to estimate the cost of locking.

20.6.2 A Cost Model for Distributed Locking Algorithms

Suppose that each data element exists at exactly one site (i.e., there is no data replication) and that the lock manager at each site stores locks and lock requests for the elements at its site. Transactions may be distributed, and each transaction consists of components at one or more sites.

While there are several costs associated with managing locks, many of them are fixed, independent of the way transactions request locks over a network. The one cost factor over which we have control is the number of messages sent between sites when a transaction obtains and releases its locks. We shall thus count the number of messages required for various locking schemes on the assumption that all locks are granted when requested. Of course, a lock request may be denied, resulting in an additional message to deny the request and a later message when the lock is granted. However, since we cannot predict the rate of lock denials, and this rate is not something we can control anyway, we shall ignore this additional requirement for messages in our comparisons.

Example 20.16: As we mentioned in Section 20.6.1, in the central locking method, the typical lock request uses three messages, one to request the lock, one from the central site to grant the lock, and a third to release the lock. The exceptions are:

1. The messages are unnecessary when the requesting site is the central lock site, and

2. Additional messages must be sent when the initial request cannot be granted.

However, we assume that both these situations are relatively rare; i.e., most lock requests are from sites other than the central lock site, and most lock requests can be granted. Thus, three messages per lock is a good estimate of the cost of the centralized lock method. □

Now, consider a situation more flexible than central locking, where there is no replication, but each database element X can maintain its locks at its own site. It might seem that, since a transaction wanting to lock X will have a component at the site of X, there are no messages between sites needed. The local component simply negotiates with the lock manager at that site for the lock on X. However, if the distributed transaction needs locks on several elements, say X, Y, and Z, then the transaction cannot complete its computation until it has locks on all three elements. If X, Y, and Z are at different sites, then the components of the transactions at those sites must at least exchange synchronization messages to prevent the transaction from proceeding before it has all the locks it needs.

Rather than deal with all the possible variations, we shall take a simple model of how transactions gather locks. We assume that one component of each transaction, the *lock coordinator* for that transaction, has the responsibility to gather all the locks that all components of the transaction require. The lock coordinator locks elements at its own site without messages, but locking an element X at any other site requires three messages:

1. A message to the site of X requesting the lock.

2. A reply message granting the lock (recall we assume all locks are granted immediately; if not, a denial message followed by a granting message later will be sent).

3. A message to the site of X releasing the lock.

If we pick as the lock coordinator the site where the most locks are needed by the transaction, then we minimize the requirement for messages. The number of messages required is three times the number of database elements at the other sites.

20.6.3 Locking Replicated Elements

When an element X has replicas at several sites, we must be careful how we interpret the locking of X.

Example 20.17: Suppose there are two copies, X_1 and X_2, of a database element X. Suppose also that a transaction T gets a shared lock on the copy X_1 at the site of that copy, while transaction U gets an exclusive lock on the copy X_2 at its site. Now, U can change X_2 but cannot change X_1, resulting in the two copies of the element X becoming different. Moreover, since T and U may lock other elements as well, and the order in which they read and write X is not forced by the locks they hold on the copies of X, there is also an opportunity for T and U to engage in unserializable behavior. □

The problem illustrated by Example 20.17 is that when data is replicated, we must distinguish between getting a shared or exclusive lock on the logical element X and getting a local lock on a copy of X. That is, in order to assure serializability, we need for transactions to take global locks on the logical elements. But the logical elements don't exist physically — only their copies do — and there is no global lock table. Thus, the only way that a transaction can obtain a global lock on X is to obtain local locks on one or more copies of X at the site(s) of those copies. We shall now consider methods for turning local locks into global locks that have the required property:

- A logical element X can have either one exclusive lock and no shared lock, or any number of shared locks and no exclusive locks.

20.6.4 Primary-Copy Locking

An improvement on the centralized locking approach, one which also allows replicated data, is to distribute the function of the lock site, but still maintain the principle that each logical element has a single site responsible for its global lock. This distributed-lock method, called *primary copy*, avoids the possibility that the central lock site will become a bottleneck, while still maintaining the simplicity of the centralized method.

In the primary copy lock method, each logical element X has one of its copies designated the "primary copy." In order to get a lock on logical element X, a transaction sends a request to the site of the primary copy of X. The site of the primary copy maintains an entry for X in its lock table and grants or denies the request as appropriate. Again, global (logical) locks will be administered correctly as long as each site administers the locks for the primary copies correctly.

Also as with a centralized lock site, most lock requests require three messages, except for those where the transaction and the primary copy are at the same site. However, if we choose primary copies wisely, then we expect that these sites will frequently be the same.

Example 20.18: In the chain-of-stores example, we should make each store's sales data have its primary copy at the store. Other copies of this data, such as at the central office or at a data warehouse used by sales analysts, are not primary copies. Probably, the typical transaction is executed at a store and updates only sales data for that store. No messages are needed when this type of transaction takes its locks. Only if the transaction examined or modified data at another store would lock-related messages be sent. □

20.6.5 Global Locks From Local Locks

Another approach is to synthesize global locks from collections of local locks. In these schemes, no copy of a database element X is "primary"; rather they are symmetric, and local shared or exclusive locks can be requested on any of these

Distributed Deadlocks

There are many opportunities for transactions to get deadlocked as they try to acquire global locks on replicated data. There are also many ways to construct a global waits-for graph and thus detect deadlocks. However, in a distributed environment, it is often simplest and also most effective to use a timeout. Any transaction that has not completed after an appropriate amount of time is assumed to have gotten deadlocked and is rolled back.

copies. The key to a successful global locking scheme is to require transactions to obtain a certain number of local locks on copies of X before the transaction can assume it has a global lock on X.

Suppose database element A has n copies. We pick two numbers:

1. s is the number of copies of A that must be locked in shared mode in order for a transaction to have a global shared lock on A.

2. x is the number of copies of A that must be locked in exclusive mode in order for a transaction to have an exclusive lock on A.

As long as $2x > n$ and $s + x > n$, we have the desired properties: there can be only one global exclusive lock on A, and there cannot be both a global shared and global exclusive lock on A. The explanation is as follows. Since $2x > n$, if two transactions had global exclusive locks on A, there would be at least one copy that had granted local exclusive locks to both (because there are more local exclusive locks granted than there are copies of A). However, then the local locking method would be incorrect. Similarly, since $s + x > n$, if one transaction had a global shared lock on A and another had a global exclusive lock on A, then some copy granted both local shared and exclusive locks at the same time.

In general, the number of messages needed to obtain a global shared lock is $3s$, and the number to obtain a global exclusive lock is $3x$. That number seems excessive, compared with centralized methods that require 3 or fewer messages per lock on the average. However, there are compensating arguments, as the following two examples of specific (s, x) choices shows.

Read-Locks-One; Write-Locks-All

Here, $s = 1$ and $x = n$. Obtaining a global exclusive lock is very expensive, but a global shared lock requires three messages at the most. Moreover, this scheme has an advantage over the primary-copy method: while the latter allows us to avoid messages when we read the primary copy, the read-locks-one scheme allows us to avoid messages whenever the transaction is at the site of *any copy* of the database element we desire to read. Thus, this scheme can be superior when most transactions are read-only, but transactions to read an element X initiate at different sites. An example would be a distributed digital library that caches copies of documents where they are most frequently read.

Majority Locking

Here, $s = x = \lceil (n+1)/2 \rceil$. It seems that this system requires many messages no matter where the transaction is. However, there are several other factors that may make this scheme acceptable. First, many network systems support *broadcast*, where it is possible for a transaction to send out one general request for local locks on an element X, which will be received by all sites. Similarly, the release of locks may be achieved by a single message.

Moreover, this selection of s and x provides an advantage others do not: it allows partial operation even when the network is disconnected. As long as there is one component of the network that contains a majority of the sites with copies of X, then it is possible for a transaction to obtain a lock on X. Even if other sites are active while disconnected, we know that they cannot even get a shared lock on X, and thus there is no risk that transactions running in different components of the network will engage in behavior that is not serializable.

20.6.6 Exercises for Section 20.6

! **Exercise 20.6.1 :** We showed how to create global shared and exclusive locks from local locks of that type. How would you create:

a) Global shared, exclusive, and increment locks

b) Global shared, exclusive, and update locks

!! c) Global shared, exclusive, and intention locks for each type

from local locks of the same types?

Exercise 20.6.2 : Suppose there are five sites, each with a copy of a database element X. One of these sites P is the dominant site for X and will be used as X's primary site in a primary-copy distributed-lock system. The statistics regarding accesses to X are:

i. 50% of all accesses are read-only accesses originating at P.

ii. Each of the other four sites originates 10% of the accesses, and these are read-only.

iii. The remaining 10% of accesses require exclusive access and may originate at any of the five sites with equal probability (i.e., 2% originate at each).

For each of the lock methods below, give the average number of messages needed to obtain a lock. Assume that all requests are granted, so no denial messages are needed.

Grid Computing

Grid computing is a term that means almost the same as peer-to-peer computing. However, the applications of grids usually involve sharing of computing resources rather than data, and there is often a master node that controls what the others do. Popular examples include SETI, which attempts to distribute the analysis of signals for signs of extraterrestrial intelligence among participating nodes, and Folding-at-Home, which attempts to do the same for protein-folding.

a) Read-locks-one; write-locks-all.

b) Majority locking.

c) Primary-copy locking, with the primary copy at P.

20.7 Peer-to-Peer Distributed Search

In this section, we examine peer-to-peer distributed systems. When these systems are used to store and deliver data, the problem of search becomes surprisingly hard. That is, each node in the peer-to-peer network has a subset of the data elements, but there is no centralized index that says where something is located. The method called "distributed hashing" allows peer-to-peer networks to grow and shrink, yet allows us to find available data much more efficiently than sending messages to every node.

20.7.1 Peer-to-Peer Networks

A *peer-to-peer* network is a collection of *nodes* or *peers* (participating machines) that:

1. Are *autonomous*: participants do not respect any central control and can join or leave the network at will.

2. Are *loosely coupled*; they communicate over a general-purpose network such as the Internet, rather than being hard-wired together like the processors in a parallel machine.

3. Are equal in functionality; there is no leader or controlling node.

4. Share resources with one another.

Peer-to-peer networks initially received a bad name, because their first popular use was in sharing copyrighted files such as music. However, they have

Copyright Issues in Digital Libraries

In order for a distributed world-wide digital library to become a reality, there will have to be some resolution of the severe copyright issues that arise. Current, small-scale versions of this network have partial solutions. For example, on-line university libraries often pass accesses to the ACM digital library only from IP addresses in the university's domain. Other arrangements are based on the idea that only one user at a time can access a particular copyrighted document. The digital library can "loan" the right to another library, but then users of the first library cannot access the document. The world awaits a solution that is easily implementable and fair to all interests.

many legitimate uses. For example, as libraries replace books by digital images, it becomes feasible for all the world's libraries to share what they have. It should not be necessary for each library to store a copy of every book or document in the world. But then, when you request a book from your local library, that library's node needs to find a peer library that does have a copy of what you want.

As another example, we might imagine a peer-to-peer network for the sharing of personal collections of photographs or videos, that is, a peer-to-peer version of Flickr or YouTube. The images are housed on participants' personal computers, so they will be turned on and off periodically. There can be millions of participants, and each has only a small fraction of the resources of the entire network.

20.7.2 The Distributed-Hashing Problem

Early peer-to-peer networks such as Napster used a centralized table that told where data elements could be found. Later systems distributed the function of locating elements, either by replication or division of the task among the peers. When the database is truly large, such as a shared worldwide library or photo-sharing network, there is no choice but to share the task in some way.

We shall abstract the problem to one of lookup of records in a (very large) set of key-value pairs. Associated with each key K is a value V. For example, K might be the identifier of a document. V could be the document itself, or it could be the set of nodes at which the document can be found.

If the size of the key-value data is small, there are several simple solutions. We could use a central node that holds the entire key-value table. All nodes would query the central node when they wanted the value V associated with a given key K. In that case, a pair of query-response messages would answer any lookup question for any node. Alternatively, we could replicate the entire table at each node, so there would be no messages needed at all.

The problem becomes more interesting when the key-value table is too large to be handled by a single node. We shall consider this problem, using the following constraints:

1. At any time, only one node among the peers knows the value associated with any given key K.
2. The key-value pairs are distributed roughly equally among the peers.
3. Any node can ask the peers for the value V associated with a chosen key K. The value of V should be obtained in a way such that the number of messages sent among the peers grows much more slowly than the number of peers.
4. The amount of routing information needed at each node to help locate keys must also grow much more slowly than the number of nodes.

20.7.3 Centralized Solutions for Distributed Hashing

If the set of participants in the network is fixed once and for all, or the set of participants changes slowly, then there are straightforward ways to manage lookup of keys. For example, we could use a hash function h that hashes keys into node numbers. We place the key-value pair (K, V) at the node $h(K)$.

In fact, Google and similar search engines effectively maintain a centralized index of the entire Web and manage huge numbers of requests. They do so by behaving logically as if there were a centralized index, when in fact the index is replicated at a very large number of nodes. Each node consists of many machines that together share the index of the Web.

However, machines at Google are not really "peers." They cannot decide to leave the network, and they each have a specific function to perform. While machines can fail, their load is simply assumed by a node of similar machines until the failed machine is replaced. In the balance of this section, we shall consider the more complex solution that is needed when the data is maintained by a true collection of peer nodes.

20.7.4 Chord Circles

We shall now describe one of several possible algorithms for distributed hashing, an algorithm with the desirable property that it uses a number of messages that is logarithmic in the number of peers. In addition, the amount of information other than key-value peers needed at each node grows logarithmically in the number of nodes.

In this algorithm, we arrange the peers in a "chord circle." Each node knows its predecessor and successor around the circle, and nodes also have links to nodes located at an exponentially growing set of distances around the circle (these links are the "chords"). Figure 20.14 suggests what the chord circle looks like.

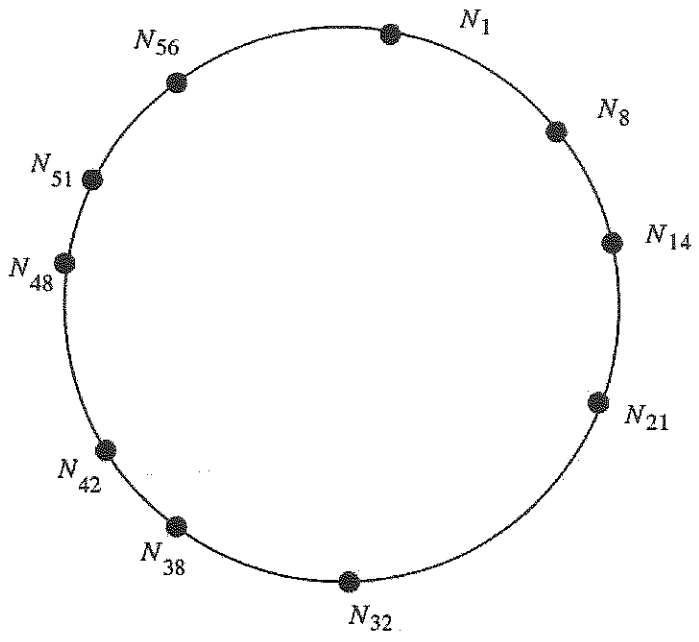

Figure 20.14: A chord circle

To place a node in the circle, we hash its ID i, and place it at position $h(i)$. We shall henceforth refer to this node as $N_{h(i)}$. Thus, for example, in Fig. 20.14, N_{21} is a node whose ID i has $h(i) = 21$. The successor of each node is the next higher one clockwise around the circle. For example, the successor of N_{21} is N_{32}, and N_1 is the successor of N_{56}. Likewise, N_{21} is the predecessor of N_{32}, and N_{56} is the predecessor of N_1.

The nodes are located around the circle using a hash function h that is capable of mapping both keys and node ID's (e.g., IP-addresses) to m-bit numbers, for some m. In Fig. 20.14, we suppose that $m = 6$, so there are 64 different possible locations for nodes around the circle. In a real application, m would be much larger.

Key-value pairs are also distributed around the circle using the hash function h. If (K, V) is a key-value pair, then we compute $h(K)$ and place (K, V) at the lowest numbered node N_j such that $h(K) \leq j$. As a special case, if $h(K)$ is above the highest-numbered node, then it is assigned to the lowest-numbered node. That is, key K goes to the first node at or clockwise of the position $h(K)$ in the circle.

Example 20.19: In Fig. 20.14, any (K, V) pair such that $42 < h(K) \leq 48$ would be stored at N_{48}. If $h(K)$ is any of $57, 58, \ldots, 63, 0, 1$, then (K, V) would be placed at N_1. □

20.7.5 Links in Chord Circles

Each node around the circle stores links to its predecessor and successor. Thus, for example, in Fig. 20.14, N_1 has successor N_8 and predecessor N_{56}. These links are sufficient to send messages around the circle to look up the value associated with any key. For instance, if N_8 wants to find the value associated with a key K such that $h(K) = 54$, it can send the request forward around the circle until a node N_j is found such that $j \geq 54$; it would be node N_{56} in Fig. 20.14.

However, linear search is much too inefficient if the circle is large. To speed up the search, each node has a *finger table* that gives the first nodes found at distances around the circle that are a power of two. That is, suppose that the hash function h produces m-bit numbers. Node N_i has entries in its finger table for distances $1, 2, 4, 8, \ldots, 2^{m-1}$. The entry for 2^j is the first node we meet after going distance 2^j clockwise around the circle. Notice that some entries may be the same node, and there are only $m - 1$ entries, even though the number of nodes could be as high as 2^m.

Distance	1	2	4	8	16	32
Node	N_{14}	N_{14}	N_{14}	N_{21}	N_{32}	N_{42}

Figure 20.15: Finger table for N_8

Example 20.20: Referring to Fig. 20.14, let us construct the finger table for N_8; this table is shown in Fig. 20.15. For distance 1, we ask what is the lowest numbered node whose number is at least $8 + 1 = 9$. That node is N_{14}, since there are no nodes numbered $9, 10, \ldots, 13$. For distance 2, we ask for the lowest node that is at least $8 + 2 = 10$; the answer is N_{14} again. Likewise, for distance 4, N_{14} is is lowest-numbered node that is at least $8 + 4 = 12$.

For distance 8, we look for the lowest-numbered node that is at least $8 + 8 = 16$. Now, N_{14} is too low. The lowest-numbered node that is at least 16 is N_{21}, so that is the entry in the finger table for 8. For 16, we need a node numbered at least 24, so the entry for 16 is N_{32}. For 32, we need a node numbered at least 40, and the proper entry is N_{42}. Figure 20.16 shows the four links that are in the finger table for N_8. □

20.7.6 Search Using Finger Tables

Suppose we are at node N_i and we want to find the key-value pair (K, V) where $h(K) = j$. We know that (K, V), if it exists, will be at the lowest-numbered node that is at least j.[6] We can use the finger table and knowledge of successors

[6]As always, "lowest" must be taken in the circular sense, as the first node you meet traveling clockwise around the circle, after reaching the point j.

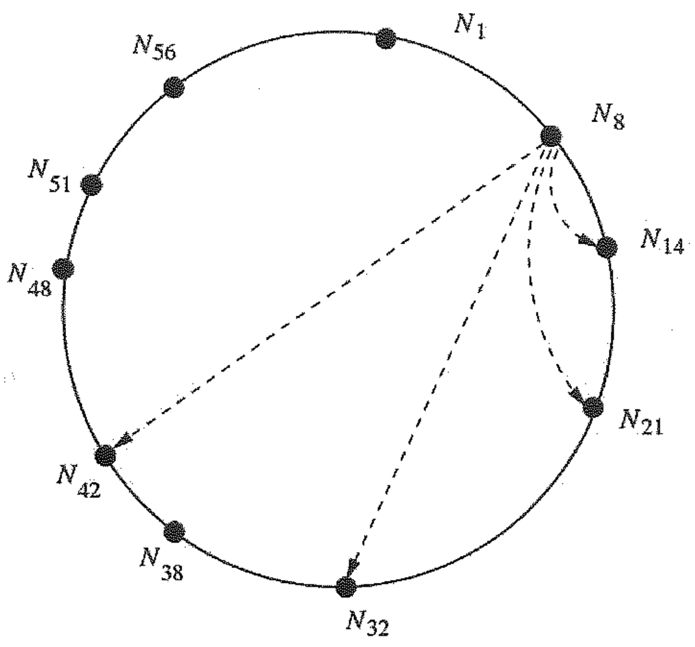

Figure 20.16: Links in the finger table for N_8

to find (K, V), if it exists, using at most $m+1$ messages, where m is the number of bits in the hash values produced by hash function h. Note that messages do not have to follow the entries of the finger table, which is needed only to help each node find out what other nodes exist.

Algorithm 20.21 : Lookup in a Chord Circle.

INPUT: An initial request by a node N_i for the value associated with key value K, where $h(K) = j$.

OUTPUT: A sequence of messages sent by various nodes, resulting in a message to N_i with either the value of V in the key-value pair (K, V), or a statement that such a pair does not exist.

METHOD: The steps of the algorithm are actually executed by different nodes. At any time, activity is at some "current" node N_c, and initially N_c is N_i. Steps (1) and (2) below are done repeatedly. Note that N_i is a part of each request message, so the current node always knows that N_i is the node to which the answer must be sent.

1. End the search if $c < j \leq s$, where N_s is the successor of N_c, around the circle. Then, N_c sends a message to N_s asking for (K, V) and informing N_s that the originator of the request is N_i. N_s will send a message to N_i with either the value V or a statement that (K, V) does not exist.

2. Otherwise, N_c consults its finger table to find the highest-numbered node N_h that is less than j. N_c sends N_h a message asking it to search for

(K, V) on behalf of N_i. N_h becomes the current node N_c, and steps (1) and (2) are repeated with the new N_c.

□

Example 20.22: Suppose N_8 wants to find the value V for key K, where $h(K) = 54$. Since the successor of N_8 is N_{14}, and 54 is not in the range $9, 10, \ldots, 14$, N_8 knows (K, V) is not at N_{14}. N_8 thus examines its finger table, and finds that all the entries are below 54. Thus it takes the largest, N_{42}, and sends a message to N_{42} asking it to look for key K and have the result sent to N_8.

N_{42} finds that 54 is not in the range $43, 44, \ldots, 48$ between N_{42} and its successor N_{48}. Thus, N_{42} examines its own finger table, which is:

Distance	1	2	4	8	16	32
Node	N_{48}	N_{48}	N_{48}	N_{51}	N_1	N_{14}

The last node (in the circular sense) that is less than 54 is N_{51}, so N_{42} sends a message to N_{51}, asking it to search for (K, V) on behalf of N_8.

N_{51} finds that 54 is no greater than its successor, N_{56}. Thus, if (K, V) exists, it is at N_{56}. N_{51} sends a request to N_{56}, which replies to N_8. The sequence of messages is shown in Fig. 20.17. □

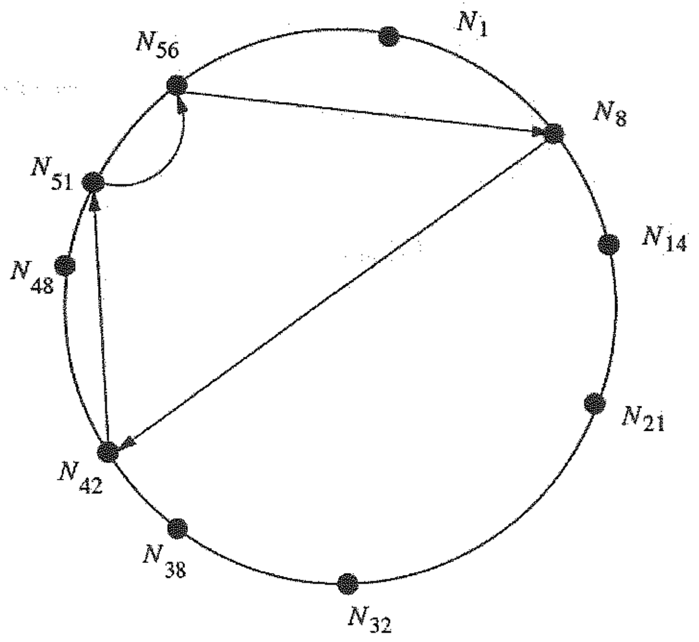

Figure 20.17: Message sequence in the search for (K, V)

In general, this recursive algorithm sends no more than m request messages. The reason is that whenever a node N_c has to consult its finger table, it messages

Dealing with Hash Collisions

Occasionally, when we insert a node, the hash value of its ID will be the same as that of some node already in the circle. The actual position of a particular node doesn't matter, as long as it knows its position and acts as if that position was the hash value of its ID. Thus, we can adjust the position of the new node up or down, until we find a position around the circle that is unoccupied.

a node that is no more than half the distance (measured clockwise around the circle) from the node holding (K, V) as N_c is. One response message is sent in all cases.

20.7.7 Adding New Nodes

Suppose a new node N_i (i.e., a node whose ID hashes to i) wants to join the network of peers. If N_i does not know how to communicate with any peer, it is not possible for N_i to join. However, if N_i knows even one peer, N_i can ask that peer what node would be N_i's successor around the circle. To answer, the known peer performs Algorithm 20.21 as if it were looking for a key that hashed to i. The node at which this hypothetical key would reside is the successor of N_i. Suppose that the successor of N_i is N_j.

We need to do two things:

1. Change predecessor and successor links, so N_i is properly linked into the circle.

2. Rearrange data so N_i gets all the data at N_j that belongs to N_i, that is, key-value pairs whose key hashes to something i or less.

We could link N into the circle at once, although it is difficult to do so correctly, because of concurrency problems. That is, several nodes whose successor would be N_j may be adding themselves at once. To avoid concurrency problems, we proceed in two steps. The first step is to set the successor of N to N_j and its predecessor to nil. N has no data at this time, and it has an empty finger table.

Example 20.23: Suppose we add to the circle of Fig. 20.14 a node N_{26}, i.e., a node whose ID hashes to 26. Whatever peer N_{26} contacted will be told that N_{26}'s successor is N_{32}. N_{26} sets its successor to N_{32} and its predecessor to nil. The predecessor of N_{32} remains N_{21} for the moment. The situation is suggested by Fig. 20.18. There, solid lines are successor links and dashed lines are predecessor links. □

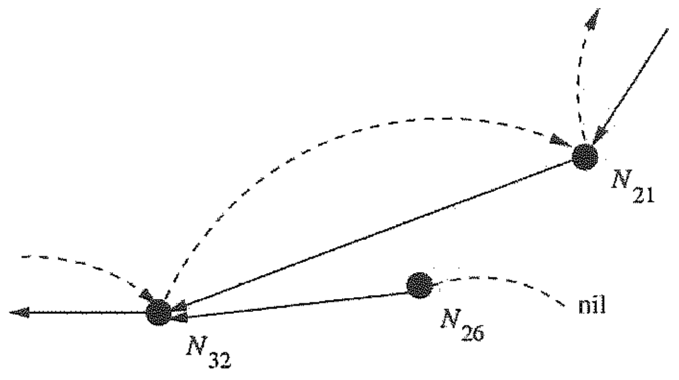

Figure 20.18: Adding node N_{26} to the network of peers

The second step is done automatically by all nodes, and is not a direct response to the insertion of N_i. All nodes must periodically perform a ***stabilization*** check, during which time predecessors and successors are updated, and if necessary, data is shared between a new node and its successor. Surely, N_{26} in Fig. 20.18 will have to perform a stabilization to get N_{32} to accept N_{26} as its predecessor, but N_{21} also needs to perform a stabilization in order to realize that N_{26} is its new successor. Note that N_{21} has not been informed of the existence of N_{26}, and will not be informed until N_{21} discovers this fact for itself during its own stabilization. The stabilization process at any node N is as follows.

1. Let S be the successor of N. N sends a message to S asking for P, the predecessor of S, and S replies. In normal cases, $P = N$, and if so, skip to step (4).

2. If P lies strictly between N and S, then N records that P is its successor.

3. Let S' be the current successor of N; S' could be either S or P, depending on what step (2) decided. If the predecessor of S' is nil or N lies strictly between S' and its predecessor, then N sends a message to S' telling S' that N is the predecessor of S'. S' sets its predecessor to N.

4. S' shares its data with N. That is, all (K, V) pairs at S' such that $h(K) \leq N$ are moved to N.

Example 20.24: Following the events of Example 20.23, with the predecessor and successor links in the state of Fig. 20.18, node N_{26} will perform a stabilization. For this stabilization, $N = N_{26}$, $S = N_{32}$, and $P = N_{21}$. Since P does not lie between N and S, step (2) makes no change, so $S' = S = N_{32}$ at step (3). Since $N = N_{26}$ lies strictly between $S' = N_{32}$ and its predecessor N_{21}, we make N_{26} the predecessor of N_{32}. The state of the links is shown in Fig. 20.19. At step (4), all key-value pairs whose keys hash to 22 through 26 are moved from N_{32} to N_{26}.

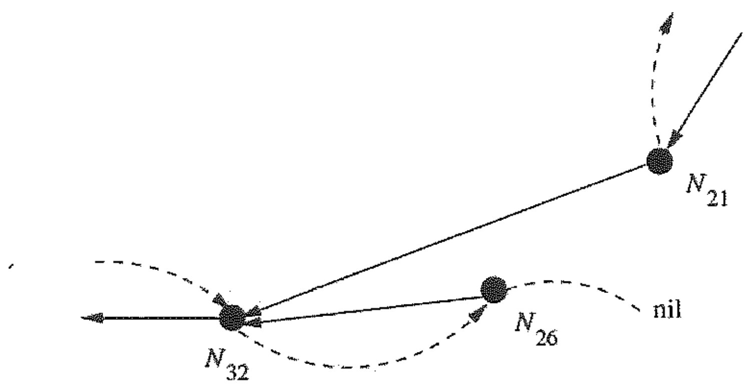

Figure 20.19: After making N_{26} the predecessor of N_{32}

The circle has still not stabilized, since N_{21} and many other nodes do not know about N_{26}. Searches for keys in the 22–26 range will still wind up at N_{32}. However, N_{32} knows that it no longer has keys in this range. N_{32}, which is N_c in Algorithm 20.21, simply continues the search according to this algorithm, which in effect causes the search to go around the circle again, possibly several times.

Eventually, N_{21} runs the stabilization operation, which it, like all nodes, does periodically. Now, $N = N_{21}$, $S = N_{32}$, and $P = N_{26}$. The test of step (2) is satisfied, so N_{26} becomes the successor of N_{21}. At step (3), $S' = N_{26}$. Since the predecessor of N_{26} is nil, we make N_{21} the predecessor of N_{26}. No data is shared at step (4), since all data at N_{26} belongs there. The final state of the predecessor and successor links is shown in Fig. 20.20.

At this time, the search for a key in the range 22–26 will reach N_{26} and be answered properly. It is possible, under rare circumstances, that insertion of many new nodes will keep the network from becoming completely stable for a long time. In that case, the search for a key in the range 22–26 could continue running until the network finally does stabilize. However, as soon as the network does stablize, the search comes to an end. □

There is still more to do, however. In terms of the running example, the finger table for N_{26} needs to be constructed, and other finger tables may now be wrong because they will link to N_{32} in some cases when they should link to N_{26}. Thus, it is necessary that every node N periodically checks its finger table. For each $i = 1, 2, 4, 8, \ldots$, node N must execute Algorithm 20.21 with $j = N + i \mod 2^m$. When it gets back the node at which the network thinks such a key would be located, N sets its finger-table entry for distance i to that value.

Notice that a new node, such as N_{26} in our running example, can construct its initial finger table this way, since the construction of any entry requires only entries that have already been constructed. That is, the entry for distance 1 is always the successor. For distance $2i$, either the successor is the correct entry, or we can find the correct entry by calling upon whatever node is the finger-table

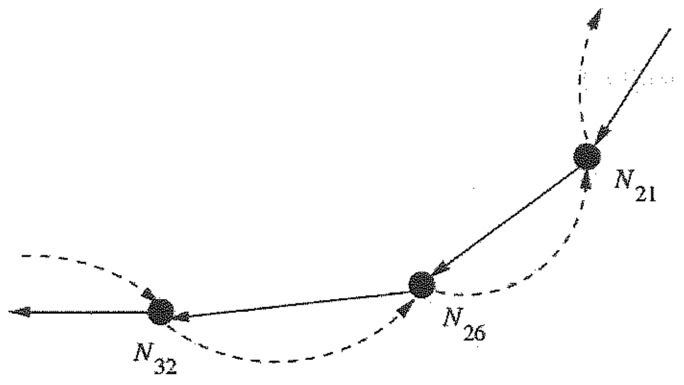

Figure 20.20: After N_{21} runs the stabilization algorithm

entry for distance i.

20.7.8 When a Peer Leaves the Network

A central tenet of peer-to-peer systems is that a node cannot be compelled to participate. Thus, a node can leave the circle at any time. The simple case is when a node leaves "gracefully," that is, cooperating with other nodes to keep the data available. To leave gracefully, a node:

1. Notifies its predecessor and successor that it is leaving, so they can become each other's predecessor and successor.
2. Transfers its data to its successor.

The network is still in a state that has errors; in particular the node that left may still appear in the finger tables of some nodes. These nodes will discover the error, either when they periodically update their finger tables, as discussed in Section 20.7.7, or when they try to communicate with the node that has disappeared. In the latter case, they can recompute the erroneous finger-table entry exactly as they would during periodic update.

20.7.9 When a Peer Fails

A harder problem occurs when a node fails, is turned off, or decides to leave without doing the "graceful" steps of Section 20.7.8. If the data is not replicated, then data at the failed node is now unavailable to the network. To avoid total unavailability of data, we can replicate it at several nodes. For example, we can place each (K, V) pair at three nodes: the correct node, its predecessor in the circle, and its successor.

To reestablish the circle when a node leaves, we can have each node record not only its predecessor and successor, but the predecessor of its predecessor and the successor of its successor. An alternative approach is to cluster nodes into groups of (say) three or more. Nodes in a cluster replicate their data and can substitute for one another, if one leaves or fails. When clusters get too large, they can be split into two clusters that are adjacent on the circle, using an algorithm similar to that described in Section 20.7.7 for node insertion. Similarly, clusters that get too small can be combined with a neighbor, a process similar to graceful leaving as in Section 20.7.8. Insertion of a new node is executed by having the node join its nearest cluster.

20.7.10 Exercises for Section 20.7

Exercise 20.7.1: Given the circle of nodes of Fig. 20.14, where do key-value pairs reside if the key hashes to: (a) 24 (b) 60?

Exercise 20.7.2: Given the circle of nodes of Fig. 20.14, construct the finger tables for: (a) N_1 (b) N_{48} (c) N_{56}.

Exercise 20.7.3: Given the circle of nodes of Fig. 20.14, what is the sequence of messages sent if:

a) N_1 searches for a key that hashes to 27.

b) N_1 searches for a key that hashes to 0.

c) N_{51} searches for a key that hashes to 45.

Exercise 20.7.4: Show the sequence of steps that adjust successor and predecessor pointers and share data, for the circle of Fig. 20.14 when nodes are added that hash to: (a) 41 (b) 62.

! **Exercise 20.7.5:** Suppose we want to guard against node failures by having each node maintain the predecessor information, successor information, and data of its predecessor and successor, as well as its own, as discussed in Section 20.7.9. How would you modify the node-insertion algorithm described in Section 20.7.7?

20.8 Summary of Chapter 20

✦ *Parallel Machines*: Parallel machines can be characterized as shared-memory, shared-disk, or shared-nothing. For database applications, the shared-nothing architecture is generally the most cost-effective.

✦ *Parallel Algorithms*: The operations of relational algebra can generally be sped up on a parallel machine by a factor close to the number of processors. The preferred algorithms start by hashing the data to buckets that correspond to the processors, and shipping data to the appropriate processor. Each processor then performs the operation on its local data.

- *The Map-Reduce Framework*: Often, highly parallel algorithms on massive files can be expressed by a map function and a reduce function. Many *map* processes execute on parts of the file in parallel, to produce key-value pairs. These pairs are then distributed so each key's pairs can be handled by one *reduce* process.

- *Distributed Data*: In a distributed database, data may be partitioned horizontally (one relation has its tuples spread over several sites) or vertically (a relation's schema is decomposed into several schemas whose relations are at different sites). It is also possible to replicate data, so presumably identical copies of a relation exist at several sites.

- *Distributed Joins*: In an environment with expensive communication, semijoins can speed up the join of two relations that are located at different sites. We project one relation onto the join attributes, send it to the other site, and return only the tuples of the second relation that are not dangling tuples.

- *Full Reducers*: When joining more than two relations at different sites, it may or may not be possible to eliminate all dangling tuples by performing semijoins. A finite sequence of semijoins that is guaranteed to eliminate all dangling tuples, no matter how large the relations are, is called a full reducer.

- *Hypergraphs*: A natural join of several relations can be represented by a hypergraph, which has a node for each attribute name and a hyperedge for each relation, which contains the nodes for all the attributes of that relation.

- *Acyclic Hypergraphs*: These are the hypergraphs that can be reduced to a single hyperedge by a series of ear-reductions — elimination of hyperedges all of whose nodes are either in no other hyperedge, or in one particular other hyperedge. Full reducers exist for all and only the hypergraphs that are acyclic.

- *Distributed Transactions*: In a distributed database, one logical transaction may consist of components, each executing at a different site. To preserve consistency, these components must all agree on whether to commit or abort the logical transaction.

- *Two-Phase Commit*: This algorithm enables transaction components to decide whether to commit or abort, often allowing a resolution even in the face of a system crash. In the first phase, a coordinator component polls the components whether they want to commit or abort. In the second phase, the coordinator tells the components to commit if and only if all have expressed a willingness to commit.

- *Distributed Locks*: If transactions must lock database elements found at several sites, a method must be found to coordinate these locks. In the centralized-site method, one site maintains locks on all elements. In the primary-copy method, the home site for an element maintains its locks.

- *Locking Replicated Data*: When database elements are replicated at several sites, global locks on an element must be obtained through locks on one or more replicas. The majority locking method requires a read- or write-lock on a majority of the replicas to obtain a global lock. Alternatively, we may allow a global read lock by obtaining a read lock on any copy, while allowing a global write lock only through write locks on every copy.

- *Peer-to-Peer Networks*: These networks consist of independent, autonomous nodes that all play the same role in the network. Such networks are generally used to share data among the peer nodes.

- *Distributed Hashing*: Distributed hashing is a central database problem in peer-to-peer networks. We are given a set of key-value pairs to distribute among the peers, and we must find the value associated with a given key without sending messages to all, or a large fraction of the peers, and without relying on any one peer that has all the key-value pairs.

- *Chord Circles*: A solution to the distributed hashing problem begins by using a hash function that hashes both node ID's and keys into the same m-bit values, which we perceive as forming a circle with 2^m positions. Keys are placed at the node at the position immediately clockwise of the position to which the key hashes. By use of a finger-table, which gives the nodes at distances $1, 2, 4, 8, \ldots$ around the circle from a given node, key lookup can be accomplished in time that is logarithmic in the number of nodes.

20.9 References for Chapter 20

The use of hashing in parallel join and other operations has been proposed several times. The earliest source we know of is [8]. The map-reduce framework for parallelism was expressed in [2]. There is an open-souce implementation available [6].

The relationship between full reducers and acyclic hypergraphs is from [1]. The test for whether a hypergraph is acyclic was discovered by [5] and [13].

The two-phase commit protocol was proposed in [7]. A more powerful scheme (not covered here) called three-phase commit is from [9]. The leader-election aspect of recovery was examined in [4].

Distributed locking methods have been proposed by [3] (the centralized locking method) [11] (primary-copy) and [12] (global locks from locks on copies).

The chord algorithm for distributed hashing is from [10].