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 $B(R) = 200$, so the disk I/O's for computing $\sigma_C(R)$ on a uniprocessor machine will take about 20 seconds. What is the speedup if this selection is executed on a parallel machine with p processors, where: (a) $p = 1000$ (b) $p = 12$ (c) $p = 100$.

! **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.

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

Exercise 20.2.2: Modify Example 20.5 to count the number of documents in which each word w appears.

20.3 Distr

We shall now con
tributed system,
participate in dat
tem and a shared
of communication
passing cost that
distributed syster
in the same roon
capacity than the

Distributed d
distributed syster
to queries. Furth
resilience in the f

On the other
aspect of a datal
components of a
dominate the cos
messages are sent
issues, while the
lems that come u
locking.

20.3.1 Dist

One important
tributed among
marily to that si

1. A bank ma
in a given c
(or city). C
bank at "t
may also h
records an
the records
a branch c

2. A chain o
store (or a
store and
with data
card custo
and what

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 $8c$. The cost of executing an update is d for the copy of R at the issuing site and $12d$ 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.

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

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

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

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

Exercise 20.4.2: For those hypergraphs of Exercise 20.4.1 that are acyclic, construct a full reducer.

! **Exercise 20.4.3:** 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 20.4.4:** 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 20.4.5:** 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 20.4.6:** 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) intersection (b) difference (c) union?

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.

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.

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 update locks

b) Global shared, exclusive, and increment 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) Primary-copy locking, with the primary copy at P.

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

c) Majority locking.

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 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) 35 (b) 20 (c) 60?

Exercise 20.7.2: Given the circle of nodes of Fig. 20.14, construct the finger tables for: (a) N_{14} (b) N_{51}

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

a) N_{14} searches for a key that hashes to 27.

b) N_8 searches for a key that hashes to 5.

c) N_{56} searches for a key that hashes to 54.

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) 16 (b) 45.

! **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.