transaction T aborts, we must determine which transactions have read data written by T, abort them, and recursively abort any transactions that have read data written by an aborted transaction. To cancel the effect of an aborted transaction, we can use the log, if it is one of the types (undo or undo/redo) that provides former values. We may also be able to restore the data from the disk copy of the database, if the effect of the dirty data has not migrated to disk.

As we have noted, a timestamp-based scheduler with a commit bit prevents a transaction that may have read dirty data from proceeding, so there is no possibility of cascading rollback with such a scheduler. A validation-based scheduler avoids cascading rollback, because writing to the database (even in buffers) occurs only after it is determined that the transaction will commit.

19.1.3 Recoverable Schedules

For any of the logging methods we have discussed in Chapter 17 to allow recovery, the set of transactions that are regarded as committed *after* recovery must be consistent. That is, if a transaction T_1 is, after recovery, regarded as committed, and T_1 used a value written by T_2, then T_2 must also remain committed, after recovery. Thus, we define:

- A schedule is *recoverable* if each transaction commits only after each transaction from which it has read has committed.

Example 19.3: In this and several subsequent examples of schedules with read- and write-actions, we shall use c_i for the action "transaction T_i commits." Here is an example of a recoverable schedule:

$$S_1\colon\ w_1(A);\ w_1(B);\ w_2(A);\ r_2(B);\ c_1;\ c_2;$$

Note that T_2 reads a value (B) written by T_1, so T_2 must commit after T_1 for the schedule to be recoverable.

Schedule S_1 above is evidently serial (and therefore serializable) as well as recoverable, but the two concepts are orthogonal. For instance, the following variation on S_1 is still recoverable, but not serializable.

$$S_2\colon\ w_2(A);\ w_1(B);\ w_1(A);\ r_2(B);\ c_1;\ c_2;$$

In schedule S_2, T_2 must precede T_1 in a serial order because of the writing of A, but T_1 must precede T_2 because of the writing and reading of B.

Finally, observe the following variation on S_1, which is serializable but not recoverable:

$$S_3\colon\ w_1(A);\ w_1(B);\ w_2(A);\ r_2(B);\ c_2;\ c_1;$$

In schedule S_3, T_1 precedes T_2, but their commitments occur in the wrong order. If before a crash, the commit record for T_2 reached disk, but the commit record for T_1 did not, then regardless of whether undo, redo, or undo/redo logging were used, T_2 would be committed after recovery, but T_1 would not. □

In order for recoverable schedules to be truly recoverable under any of the three logging methods, there is one additional assumption we must make regarding schedules:

- The log's commit records reach disk in the order in which they are written.

As we observed in Example 19.3 concerning schedule S_3, should it be possible for commit records to reach disk in the wrong order, then consistent recovery might be impossible. We shall return to and exploit this principle in Section 19.1.6.

19.1.4 Schedules That Avoid Cascading Rollback

Recoverable schedules sometimes require cascading rollback. For instance, if after the first four steps of schedule S_1 in Example 19.3 T_1 had to roll back, it would be necessary to roll back T_2 as well. To guarantee the absence of cascading rollback, we need a stronger condition than recoverability. We say that:

- A schedule *avoids cascading rollback* (or "is an *ACR schedule*") if transactions may read only values written by committed transactions.

Put another way, an ACR schedule forbids the reading of dirty data. As for recoverable schedules, we assume that "committed" means that the log's commit record has reached disk.

Example 19.4: The schedules of Example 19.3 are not ACR. In each case, T_2 reads B from the uncommitted transaction T_1. However, consider:

$$S_4\text{: } w_1(A);\ w_1(B);\ w_2(A);\ c_1;\ r_2(B);\ c_2;$$

Now, T_2 reads B only after T_1, the transaction that last wrote B, has committed, and its log record written to disk. Thus, schedule S_4 is ACR, as well as recoverable. □

Notice that should a transaction such as T_2 read a value written by T_1 after T_1 commits, then surely T_2 either commits or aborts after T_1 commits. Thus:

- Every ACR schedule is recoverable.

19.1.5 Managing Rollbacks Using Locking

Our prior discussion applies to schedules that are generated by any kind of scheduler. In the common case that the scheduler is lock-based, there is a simple and commonly used way to guarantee that there are no cascading rollbacks:

- *Strict Locking*: A transaction must not release any exclusive locks (or other locks, such as increment locks that allow values to be changed) until the transaction has either committed or aborted, and the commit or abort log record has been flushed to disk.

A schedule of transactions that follow the strict-locking rule is called a *strict schedule*. Two important properties of these schedules are:

1. *Every strict schedule is ACR.* The reason is that a transaction T_2 cannot read a value of element X written by T_1 until T_1 releases any exclusive lock (or similar lock that allows X to be changed). Under strict locking, the release does not occur until after commit.

2. *Every strict schedule is serializable.* To see why, observe that a strict schedule is equivalent to the serial schedule in which each transaction runs instantaneously at the time it commits.

With these observations, we can now picture the relationships among the different kinds of schedules we have seen so far. The containments are suggested in Fig. 19.3.

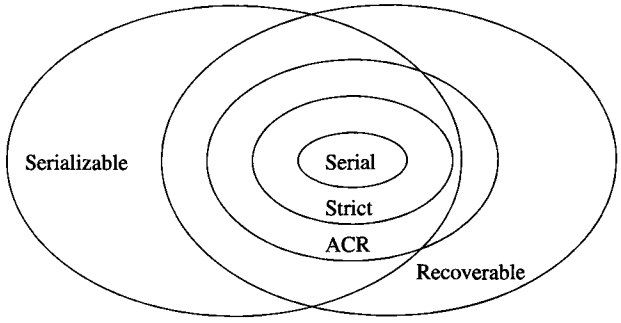

Figure 19.3: Containments and noncontainments among classes of schedules

Clearly, in a strict schedule, it is not possible for a transaction to read dirty data, since data written to a buffer by an uncommitted transaction remains locked until the transaction commits. However, we still have the problem of fixing the data in buffers when a transaction aborts, since these changes must have their effects cancelled. How difficult it is to fix buffered data depends on whether database elements are blocks or something smaller. We shall consider each.

Rollback for Blocks

If the lockable database elements are blocks, then there is a simple rollback method that never requires us to use the log. Suppose that a transaction T has obtained an exclusive lock on block A, written a new value for A in a buffer, and then had to abort. Since A has been locked since T wrote its value, no other transaction has read A. It is easy to restore the old value of A provided the following rule is followed:

- Blocks written by uncommitted transactions are pinned in main memory; that is, their buffers are not allowed to be written to disk.

In this case, we "roll back" T when it aborts by telling the buffer manager to ignore the value of A. That is, the buffer occupied by A is not written anywhere, and its buffer is added to the pool of available buffers. We can be sure that the value of A on disk is the most recent value written by a committed transaction, which is exactly the value we want A to have.

There is also a simple rollback method if we are using a multiversion system as in Sections 18.8.5 and 18.8.6. We must again assume that blocks written by uncommitted transactions are pinned in memory. Then, we simply remove the value of A that was written by T from the list of available values of A. Note that because T was a writing transaction, its value of A was locked from the time the value was written to the time it aborted (assuming the timestamp/lock scheme of Section 18.8.6 is used).

Rollback for Small Database Elements

When lockable database elements are fractions of a block (e.g., tuples or objects), then the simple approach to restoring buffers that have been modified by aborted transactions will not work. The problem is that a buffer may contain data changed by two or more transactions; if one of them aborts, we still must preserve the changes made by the other. We have several choices when we must restore the old value of a small database element A that was written by the transaction that has aborted:

1. We can read the original value of A from the database stored on disk and modify the buffer contents appropriately.
2. If the log is an undo or undo/redo log, then we can obtain the former value from the log itself. The same code used to recover from crashes may be used for "voluntary" rollbacks as well.
3. We can keep a separate main-memory log of the changes made by each transaction, preserved for only the time that transaction is active. The old value can be found from this "log."

None of these approaches is ideal. The first surely involves a disk access. The second (examining the log) might not involve a disk access, if the relevant portion of the log is still in a buffer. However, it could also involve extensive examination of portions of the log on disk, searching for the update record that tells the correct former value. The last approach does not require disk accesses, but may consume a large fraction of memory for the main-memory "logs."

19.1.6 Group Commit

Under some circumstances, we can avoid reading dirty data even if we do not flush every commit record on the log to disk immediately. As long as we flush

log records in the order that they are written, we can release locks as soon as the commit record is written to the log in a buffer.

Example 19.5: Suppose transaction T_1 writes X, finishes, writes its `COMMIT` record on the log, but the log record remains in a buffer. Even though T_1 has not committed in the sense that its commit record can survive a crash, we shall release T_1's locks. Then T_2 reads X and "commits," but its commit record, which follows that of T_1, also remains in a buffer. Since we are flushing log records in the order written, T_2 cannot be perceived as committed by a recovery manager (because its commit record reached disk) unless T_1 is also perceived as committed. Thus, the recovery manager will find either one of two things:

1. T_1 is committed on disk. Then regardless of whether or not T_2 is committed on disk, we know T_2 did not read X from an uncommitted transaction.

2. T_1 is not committed on disk. Then neither is T_2, and both are aborted by the recovery manager. In this case, the fact that T_2 read X from an uncommitted transaction has no effect on the database.

On the other hand, suppose that the buffer containing T_2's commit record got flushed to disk (say because the buffer manager decided to use the buffer for something else), but the buffer containing T_1's commit record did not. If there is a crash at that point, it will look to the recovery manager that T_1 did not commit, but T_2 did. The effect of T_2 will be permanently reflected in the database, but this effect was based on the dirty read of X by T_2. □

Our conclusion from Example 19.5 is that we can release locks earlier than the time that the transaction's commit record is flushed to disk. This policy, often called *group commit*, is:

- Do not release locks until the transaction finishes, and the commit log record at least appears in a buffer.

- Flush log blocks in the order that they were created.

Group commit, like the policy of requiring "recoverable schedules" as discussed in Section 19.1.3, guarantees that there is never a read of dirty data.

19.1.7 Logical Logging

We saw in Section 19.1.5 that dirty reads are easier to fix up when the unit of locking is the block or page. However, there are at least two problems presented when database elements are blocks.

1. All logging methods require either the old or new value of a database element, or both, to be recorded in the log. When the change to a block

When is a Transaction Really Committed?

The subtlety of group commit reminds us that a completed transaction can be in several different states between when it finishes its work and when it is truly "committed," in the sense that under no circumstances, including the occurrence of a system failure, will the effect of that transaction be lost. As we noted in Chapter 17, it is possible for a transaction to finish its work and even write its `COMMIT` record to the log in a main-memory buffer, yet have the effect of that transaction lost if there is a system crash and the `COMMIT` record has not yet reached disk. Moreover, we saw in Section 17.5 that even if the `COMMIT` record is on disk but not yet backed up in the archive, a media failure can cause the transaction to be undone and its effect to be lost.

In the absence of failure, all these states are equivalent, in the sense that each transaction will surely advance from being finished to having its effects survive even a media failure. However, when we need to take failures and recovery into account, it is important to recognize the differences among these states, which otherwise could all be referred to informally as "committed."

is small, e.g., a rewritten attribute of one tuple, or an inserted or deleted tuple, then there is a great deal of redundant information written on the log.

2. The requirement that the schedule be recoverable, releasing its locks only after commit, can inhibit concurrency severely. For example, recall our discussion in Section 18.7.1 of the advantage of early lock release as we access data through a B-tree index. If we require that locks be held until commit, then this advantage cannot be obtained, and we effectively allow only one writing transaction to access a B-tree at any time.

Both these concerns motivate the use of *logical logging*, where only the changes to the blocks are described. There are several degrees of complexity, depending on the nature of the change.

1. A small number of bytes of the database element are changed, e.g., the update of a fixed-length field. This situation can be handled in a straightforward way, where we record only the changed bytes and their positions. Example 19.6 will show this situation and an appropriate form of update record.

2. The change to the database element is simply described, and easily restored, but it has the effect of changing most or all of the bytes in the database element. One common situation, discussed in Example 19.7, is

when a variable-length field is changed and much of its record, and even other records, must slide within the block. The new and old values of the block look very different unless we realize and indicate the simple cause of the change.

3. The change affects many bytes of a database element, and further changes can prevent this change from ever being undone. This situation is true "logical" logging, since we cannot even see the undo/redo process as occurring on the database elements themselves, but rather on some higher-level "logical" structure that the database elements represent. We shall, in Example 19.8, take up the matter of B-trees, a logical structure represented by database elements that are disk blocks, to illustrate this complex form of logical logging.

Example 19.6: Suppose database elements are blocks that each contain a set of tuples from some relation. We can express the update of an attribute by a log record that says something like "tuple t had its attribute a changed from value v_1 to v_2." An insertion of a new tuple into empty space on the block can be expressed as "a tuple t with value $(a_1, a_2, \ldots, a_k)$ was inserted beginning at offset position p." Unless the attribute changed or the tuple inserted are comparable in size to a block, the amount of space taken by these records will be much smaller than the entire block. Moreover, they serve for both undo and redo operations.

Notice that both these operations are idempotent; if you perform them several times on a block, the result is the same as performing them once. Likewise, their implied inverses, where the value of $t[a]$ is restored from v_2 back to v_1, or the tuple t is removed, are also idempotent. Thus, records of these types can be used for recovery in exactly the same way that update log records were used throughout Chapter 17. □

Example 19.7: Again assume database elements are blocks holding tuples, but the tuples have some variable-length fields. If a change to a field such as was described in Example 19.6 occurs, we may have to slide large portions of the block to make room for a longer field, or to preserve space if a field becomes smaller. In extreme cases, we could have to create an overflow block (recall Section 13.8) to hold part of the contents of the original block, or we could remove an overflow block if a shorter field allows us to combine the contents of two blocks into one.

As long as the block and its overflow block(s) are considered part of one database element, then it is straightforward to use the old and/or new value of the changed field to undo or redo the change. However, the block-plus-overflow-block(s) must be thought of as holding certain tuples at a "logical" level. We may not even be able to restore the bytes of these blocks to their original state after an undo or redo, because there may have been reorganization of the blocks due to other changes that varied the length of other fields. Yet if we think of a database element as being a collection of blocks that together represent certain

tuples, then a redo or undo can indeed restore the logical "state" of the element. □

However, it may not be possible, as we suggested in Example 19.7, to treat blocks as expandable through the mechanism of overflow blocks. We may thus be able to undo or redo actions only at a level higher than blocks. The next example discusses the important case of B-tree indexes, where the management of blocks does not permit overflow blocks, and we must think of undo and redo as occurring at the "logical" level of the B-tree itself, rather than the blocks.

Example 19.8: Let us consider the problem of logical logging for B-tree nodes. Instead of writing the old and/or new value of an entire node (block) on the log, we write a short record that describes the change. These changes include:

1. Insertion or deletion of a key/pointer pair for a child.
2. Change of the key associated with a pointer.
3. Splitting or merging of nodes.

Each of these changes can be indicated with a short log record. Even the splitting operation requires only telling where the split occurs, and where the new nodes are. Likewise, merging requires only a reference to the nodes involved, since the manner of merging is determined by the B-tree management algorithms used.

Using logical update records of these types allows us to release locks earlier than would otherwise be required for a recoverable schedule. The reason is that dirty reads of B-tree blocks are never a problem for the transaction that reads them, provided its only purpose is to use the B-tree to locate the data the transaction needs to access.

For instance, suppose that transaction T reads a leaf node N, but the transaction U that last wrote N later aborts, and some change made to N (e.g., the insertion of a new key/pointer pair into N due to an insertion of a tuple by U) needs to be undone. If T has also inserted a key/pointer pair into N, then it is not possible to restore N to the way it was before U modified it. However, the effect of U on N can be undone; in this example we would delete the key/pointer pair that U had inserted. The resulting N is not the same as that which existed before U operated; it has the insertion made by T. However, there is no database inconsistency, since the B-tree as a whole continues to reflect only the changes made by committed transactions. That is, we have restored the B-tree at a logical level, but not at the physical level. □

19.1.8 Recovery From Logical Logs

If the logical actions are idempotent — i.e., they can be repeated any number of times without harm — then we can recover easily using a logical log. For

instance, we discussed in Example 19.6 how a tuple insertion could be represented in the logical log by the tuple and the place within a block where the tuple was placed. If we write that tuple in the same place two or more times, then it is as if we had written it once. Thus, when recovering, should we need to redo a transaction that inserted a tuple, we can repeat the insertion into the proper block at the proper place, without worrying whether we had already inserted that tuple.

In contrast, consider a situation where tuples can move around within blocks or between blocks, as in Examples 19.7 and 19.8. Now, we cannot associate a particular place into which a tuple is to be inserted; the best we can do is place in the log an action such as "the tuple t was inserted somewhere on block B." If we need to redo the insertion of t during recovery, we may wind up with two copies of t in block B. Worse, we may not know whether the block B with the first copy of t made it to disk. Another transaction writing to another database element on block B may have caused a copy of B to be written to disk, for example.

To disambiguate situations such as this when we recover using a logical log, a technique called *log sequence numbers* has been developed.

- Each log record is given a number one greater than that of the previous log record.[1] Thus, a typical logical log record has the form $<L, T, A, B>$, where:
  - L is the log sequence number, an integer.
  - T is the transaction involved.
  - A is the action performed by T, e.g., "insert of tuple t."
  - B is the block on which the action was performed.
- For each action, there is a *compensating action* that logically undoes the action. As discussed in Example 19.8, the compensating action may not restore the database to exactly the same state S it would have been in had the action never occurred, but it restores the database to a state that is logically equivalent to S. For instance, the compensating action for "insert tuple t" is "delete tuple t."
- If a transaction T aborts, then for each action performed on the database by T, the compensating action is performed, and the fact that this action was performed is also recorded in the log.
- Each block maintains, in its header, the log sequence number of the last action that affected that block.

Suppose now that we need to use the logical log to recover after a crash. Here is an outline of the steps to take.

[1] Eventually the log sequence numbers must restart at 0, but the time between restarts of the sequence is so large that no ambiguity can occur.

1. Our first step is to reconstruct the state of the database at the time of the crash, including blocks whose current values were in buffers and therefore got lost. To do so:
   (a) Find the most recent checkpoint on the log, and determine from it the set of transactions that were active at that time.
   (b) For each log entry $<L, T, A, B>$, compare the log sequence number N on block B with the log sequence number L for this log record. If $N < L$, then redo action A; that action was never performed on block B. However, if $N \geq L$, then do nothing; the effect of A was already felt by B.
   (c) For each log entry that informs us that a transaction T started, committed, or aborted, adjust the set of active transactions accordingly.
2. The set of transactions that remain active when we reach the end of the log must be aborted. To do so:
   (a) Scan the log again, this time from the end back to the previous checkpoint. Each time we encounter a record $<L, T, A, B>$ for a transaction T that must be aborted, perform the compensating action for A on block B and record in the log the fact that that compensating action was performed.
   (b) If we must abort a transaction that began prior to the most recent checkpoint (i.e., that transaction was on the active list for the checkpoint), then continue back in the log until the start-records for all such transactions have been found.
   (c) Write abort-records in the log for each of the transactions we had to abort.

19.1.9 Exercises for Section 19.1

Exercise 19.1.1: What are all the ways to insert locks (of a single type only, as in Section 18.3) into the sequence of actions

$$r_1(A);\ r_1(B);\ w_1(A);\ w_1(B);$$

so that the transaction T_1 is:

a) Two-phase locked, and strict.

b) Two-phase locked, but not strict.

Exercise 19.1.2: Suppose that each of the sequences of actions below is followed by an abort action for transaction T_1. Tell which transactions need to be rolled back.

a) $r_1(A);\ r_2(B);\ w_1(B);\ w_2(C);\ r_3(B);\ r_3(C);\ w_3(D);$

b) $r_1(A)$; $w_1(B)$; $r_2(B)$; $w_2(C)$; $r_3(C)$; $w_3(D)$;

c) $r_2(A)$; $r_3(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$; $r_3(B)$; $w_2(C)$; $r_3(C)$;

d) $r_2(A)$; $r_3(A)$; $r_1(A)$; $w_1(B)$; $r_3(B)$; $w_2(C)$; $r_3(C)$;

Exercise 19.1.3: Consider each of the sequences of actions in Exercise 19.1.2, but now suppose that all three transactions commit and write their commit record on the log immediately after their last action. However, a crash occurs, and a tail of the log was not written to disk before the crash and is therefore lost. Tell, depending on where the lost tail of the log begins:

i. What transactions could be considered uncommitted?

ii. Are any dirty reads created during the recovery process? If so, what transactions need to be rolled back?

iii. What additional dirty reads could have been created if the portion of the log lost was not a tail, but rather some portions in the middle?

! **Exercise 19.1.4:** Consider the following two transactions:

$$T_1\text{: } w_1(A);\ w_1(B);\ r_1(C);\ c_1;$$
$$T_2\text{: } w_2(A);\ r_2(B);\ w_2(C);\ c_2;$$

a) How many schedules of T_1 and T_2 are recoverable?

b) Of these, how many are ACR schedules?

c) How many are both recoverable and serializable?

d) How many are both ACR and serializable?

Exercise 19.1.5: Give an example of an ACR schedule with shared and exclusive locks that is not strict.

19.2 Deadlocks

Several times we have observed that concurrently executing transactions can compete for resources and thereby reach a state where there is a *deadlock*: each of several transactions is waiting for a resource held by one of the others, and none can make progress.

- In Section 18.3.4 we saw how ordinary operation of two-phase-locked transactions can still lead to a deadlock, because each has locked something that another transaction also needs to lock.
- In Section 18.4.3 we saw how the ability to upgrade locks from shared to exclusive can cause a deadlock because each transaction holds a shared lock on the same element and wants to upgrade the lock.

There are two broad approaches to dealing with deadlock. We can detect deadlocks and fix them, or we can manage transactions in such a way that deadlocks are never able to form.

19.2.1 Deadlock Detection by Timeout

When a deadlock exists, it is generally impossible to repair the situation so that all transactions involved can proceed. Thus, at least one of the transactions will have to be aborted and restarted.

The simplest way to detect and resolve deadlocks is with a *timeout*. Put a limit on how long a transaction may be active, and if a transaction exceeds this time, roll it back. For example, in a simple transaction system, where typical transactions execute in milliseconds, a timeout of one minute would affect only transactions that are caught in a deadlock.

Notice that when one deadlocked transaction times out and rolls back, it releases its locks or other resources. Thus, there is a chance that the other transactions involved in the deadlock will complete before reaching their timeout limits. However, since transactions involved in a deadlock are likely to have started at approximately the same time (or else, one would have completed before another started), it is also possible that spurious timeouts of transactions that are no longer involved in a deadlock will occur.

19.2.2 The Waits-For Graph

Deadlocks that are caused by transactions waiting for locks held by another can be detected by a *waits-for graph*, indicating which transactions are waiting for locks held by another transaction. This graph can be used either to detect deadlocks after they have formed or to prevent deadlocks from ever forming. We shall assume the latter, which requires us to maintain the waits-for graph at all times, refusing to allow an action that creates a cycle in the graph.

Recall from Section 18.5.2 that a lock table maintains for each database element X a list of the transactions that are waiting for locks on X, as well as transactions that currently hold locks on X. The waits-for graph has a node for each transaction that currently holds any lock or is waiting for one. There is an arc from node (transaction) T to node U if there is some database element A such that:

1. U holds a lock on A,
2. T is waiting for a lock on A, and
3. T cannot get a lock on A in its desired mode unless U first releases its lock on A.[2]

[2]In common situations, such as shared and exclusive locks, every waiting transaction will have to wait until *all* current lock holders release their locks, but there are examples of systems of lock modes where a transaction can get its lock after only some of the current locks are released; see Exercise 19.2.6.

If there are no cycles in the waits-for graph, then each transaction can complete eventually. There will be at least one transaction waiting for no other transaction, and this transaction surely can complete. At that time, there will be at least one other transaction that is not waiting, which can complete, and so on.

However, if there is a cycle, then no transaction in the cycle can ever make progress, so there is a deadlock. Thus, a strategy for deadlock avoidance is to roll back any transaction that makes a request that would cause a cycle in the waits-for graph.

Example 19.9: Suppose we have the following four transactions, each of which reads one element and writes another:

T_1: $l_1(A)$; $r_1(A)$; $l_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;

T_2: $l_2(C)$; $r_2(C)$; $l_2(A)$; $w_2(A)$; $u_2(C)$; $u_2(A)$;

T_3: $l_3(B)$; $r_3(B)$; $l_3(C)$; $w_3(C)$; $u_3(B)$; $u_3(C)$;

T_4: $l_4(D)$; $r_4(D)$; $l_4(A)$; $w_4(A)$; $u_4(D)$; $u_4(A)$;

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$;			
2)		$l_2(C)$; $r_2(C)$;		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(D)$; $r_4(D)$;
5)		$l_2(A)$; **Denied**		
6)			$l_3(C)$; **Denied**	
7)				$l_4(A)$; **Denied**
8)	$l_1(B)$; **Denied**			

Figure 19.4: Beginning of a schedule with a deadlock

We use a simple locking system with only one lock mode, although the same effect would be noted if we were to use a shared/exclusive system. In Fig. 19.4 is the beginning of a schedule of these four transactions. In the first four steps, each transaction obtains a lock on the element it wants to read. At step (5), T_2 tries to lock A, but the request is denied because T_1 already has a lock on A. Thus, T_2 waits for T_1, and we draw an arc from the node for T_2 to the node for T_1.

Similarly, at step (6) T_3 is denied a lock on C because of T_2, and at step (7), T_4 is denied a lock on A because of T_1. The waits-for graph at this point is as shown in Fig. 19.5. There is no cycle in this graph.

At step (8), T_1 must wait for the lock on B held by T_3. If we allow T_1 to wait, there is a cycle in the waits-for graph involving T_1, T_2, and T_3, as seen

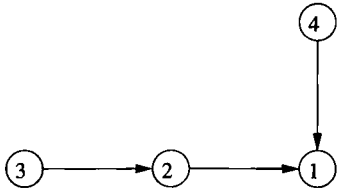

Figure 19.5: Waits-for graph after step (7) of Fig. 19.4

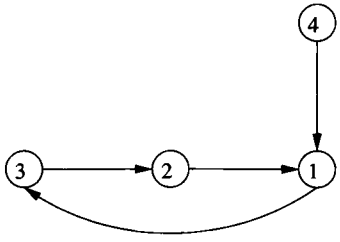

Figure 19.6: Waits-for graph with a cycle caused by step (8) of Fig. 19.4

in Fig. 19.6. Since each of these transactions is waiting for another to finish, none can make progress, and therefore there is a deadlock involving these three transactions. Incidentally, T_4 can not finish either, although it is not in the cycle, because T_4's progress depends on T_1 making progress.

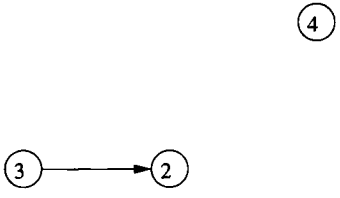

Figure 19.7: Waits-for graph after T_1 is rolled back

Since we roll back any transaction that causes a cycle, T_1 must be rolled back, yielding the waits-for graph of Fig. 19.7. T_1 relinquishes its lock on A, which may be given to either T_2 or T_4. Suppose it is given to T_2. Then T_2 can complete, whereupon it relinquishes its locks on A and C. Now T_3, which needs a lock on C, and T_4, which needs a lock on A, can both complete. At some time, T_1 is restarted, but it cannot get locks on A and B until T_2, T_3, and T_4 have completed. □

19.2.3 Deadlock Prevention by Ordering Elements

Now, let us consider several more methods for deadlock prevention. The first requires us to order database elements in some arbitrary but fixed order. For instance, if database elements are blocks, we could order them lexicographically by their physical address.

If every transaction is required to request locks on elements in order, then there can be no deadlock due to transactions waiting for locks. For suppose T_2 is waiting for a lock on A_1 held by T_1; T_3 is waiting for a lock on A_2 held by T_2, and so on, while T_n is waiting for a lock on A_{n-1} held by T_{n-1}, and T_1 is waiting for a lock on A_n held by T_n. Since T_2 has a lock on A_2 but is waiting for A_1, it must be that $A_2 < A_1$ in the order of elements. Similarly, $A_i < A_{i-1}$ for $i = 3, 4, \ldots, n$. But since T_1 has a lock on A_1 while it is waiting for A_n, it also follows that $A_1 < A_n$. We now have $A_1 < A_n < A_{n-1} < \cdots < A_2 < A_1$, which is impossible, since it implies $A_1 < A_1$.

Example 19.10: Let us suppose elements are ordered alphabetically. Then if the four transactions of Example19.9 are to lock elements in alphabetical order, T_2 and T_4 must be rewritten to lock elements in the opposite order. Thus, the four transactions are now:

T_1: $l_1(A)$; $r_1(A)$; $l_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;

T_2: $l_2(A)$; $l_2(C)$; $r_2(C)$; $w_2(A)$; $u_2(C)$; $u_2(A)$;

T_3: $l_3(B)$; $r_3(B)$; $l_3(C)$; $w_3(C)$; $u_3(B)$; $u_3(C)$;

T_4: $l_4(A)$; $l_4(D)$; $r_4(D)$; $w_4(A)$; $u_4(D)$; $u_4(A)$;

Figure 19.8 shows what happens if the transactions execute with the same timing as Fig. 19.4. T_1 begins and gets a lock on A. T_2 tries to begin next by getting a lock on A, but must wait for T_1. Then, T_3 begins by getting a lock on B, but T_4 is unable to begin because it too needs a lock on A, for which it must wait.

Since T_2 is stalled, it cannot proceed, and following the order of events in Fig. 19.4, T_3 gets a turn next. It is able to get its lock on C, whereupon it completes at step (6). Now, with T_3's locks on B and C released, T_1 is able to complete, which it does at step (8). At this point, the lock on A becomes available, and we suppose that it is given on a first-come-first-served basis to T_2. Then, T_2 can get both locks that it needs and completes at step (11). Finally, T_4 can get its locks and completes. □

19.2.4 Detecting Deadlocks by Timestamps

We can detect deadlocks by maintaining the waits-for graph, as we discussed in Section 19.2.2. However, this graph can be large, and analyzing it for cycles each time a transaction has to wait for a lock can be time-consuming. An

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$;			
2)		$l_2(A)$; **Denied**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Denied**
5)			$l_3(C)$; $w_3(C)$;	
6)			$u_3(B)$; $u_3(C)$;	
7)	$l_1(B)$; $w_1(B)$;			
8)	$u_1(A)$; $u_1(B)$;			
9)		$l_2(A)$; $l_2(C)$;		
10)		$r_2(C)$; $w_2(A)$;		
11)		$u_2(A)$; $u_2(C)$;		
12)				$l_4(A)$; $l_4(D)$;
13)				$r_4(D)$; $w_4(A)$;
14)				$u_4(A)$; $u_4(D)$;

Figure 19.8: Locking elements in alphabetical order prevents deadlock

alternative to maintaining the waits-for graph is to associate with each transaction a timestamp. This timestamp is for deadlock detection only; it is not the same as the timestamp used for concurrency control in Section 18.8, even if timestamp-based concurrency control is in use. In particular, if a transaction is rolled back, it restarts with a new, later concurrency timestamp, but its timestamp for deadlock detection never changes.

The timestamp is used when a transaction T has to wait for a lock that is held by another transaction U. Two different things happen, depending on whether T or U is *older* (has the earlier timestamp). There are two different policies that can be used to manage transactions and detect deadlocks.

1. The *Wait-Die Scheme*:

   (a) If T is older than U (i.e., the timestamp of T is smaller than U's timestamp), then T is allowed to wait for the lock(s) held by U.

   (b) If U is older than T, then T "dies"; it is rolled back.

2. The *Wound-Wait Scheme*:

   (a) If T is older than U, it "wounds" U. Usually, the "wound" is fatal: U must roll back and relinquish to T the lock(s) that T needs from U. There is an exception if, by the time the "wound" takes effect, U has already finished and released its locks. In that case, U survives and need not be rolled back.

   (b) If U is older than T, then T waits for the lock(s) held by U.

Example 19.11: Let us consider the wait-die scheme, using the transactions of Example 19.10. We shall assume that T_1, T_2, T_3, T_4 is the order of times; i.e., T_1 is the oldest transaction. We also assume that when a transaction rolls back, it does not restart soon enough to become active before the other transactions finish.

Figure 19.9 shows a possible sequence of events under the wait-die scheme. T_1 gets the lock on A first. When T_2 asks for a lock on A, it dies, because T_1 is older than T_2. In step (3), T_3 gets a lock on B, but in step (4), T_4 asks for a lock on A and dies because T_1, the holder of the lock on A, is older than T_4. Next, T_3 gets its lock on C and completes. When T_1 continues, it finds the lock on B available and also completes at step (8).

Now, the two transactions that rolled back — T_2 and T_4 — start again. Their timestamps, as far as deadlock is concerned, do not change; T_2 is still older than T_4. However, we assume that T_4 restarts first, at step (9), and when the older transaction T_2 requests a lock on A at step (10), it is forced to wait, but does not abort. T_4 completes at step (12), and then T_2 is allowed to run to completion, as shown in the last three steps. □

Example 19.12: Next, let us consider the same transactions running under the wound-wait policy, as shown in Fig. 19.10. As in Fig. 19.9, T_1 begins by locking A. When T_2 requests a lock on A at step (2), it waits, since T_1 is older than T_2. After T_3 gets its lock on B at step (3), T_4 is also made to wait for the lock on A.

Then, suppose that T_1 continues at step (5) with its request for the lock on B. That lock is already held by T_3, but T_1 is older than T_3. Thus, T_1 "wounds" T_3. Since T_3 is not yet finished, the wound is fatal: T_3 relinquishes its lock and rolls back. Thus, T_1 is able to complete.

When T_1 makes the lock on A available, suppose it is given to T_2, which is then able to proceed. After T_2, the lock is given to T_4, which proceeds to completion. Finally, T_3 restarts and completes without interference. □

19.2.5 Comparison of Deadlock-Management Methods

In both the wait-die and wound-wait schemes, older transactions kill off newer transactions. Since transactions restart with their old timestamp, eventually each transaction becomes the oldest in the system and is sure to complete. This guarantee, that every transaction eventually completes, is called *no starvation.* Notice that other schemes described in this section do not necessarily prevent starvation; if extra measures are not taken, a transaction could repeatedly start, get involved in a deadlock, and be rolled back. (see Exercise 19.2.7).

There is, however, a subtle difference in the way wait-die and wound-wait behave. In wound-wait, a newer transaction is killed whenever an old transaction asks for a lock held by the newer transaction. If we assume that transactions take their locks near the time that they begin, it will be rare that an old transaction was beaten to a lock by a new transaction. Thus, we expect rollback to be rare in wound-wait.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$;			
2)		$l_2(A)$; **Dies**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Dies**
5)			$l_3(C)$; $w_3(C)$;	
6)			$u_3(B)$; $u_3(C)$;	
7)	$l_1(B)$; $w_1(B)$;			
8)	$u_1(A)$; $u_1(B)$;			
9)				$l_4(A)$; $l_4(D)$;
10)		$l_2(A)$; **Waits**		
11)				$r_4(D)$; $w_4(A)$;
12)				$u_4(A)$; $u_4(D)$;
13)		$l_2(A)$; $l_2(C)$;		
14)		$r_2(C)$; $w_2(A)$;		
15)		$u_2(A)$; $u_2(C)$;		

Figure 19.9: Actions of transactions detecting deadlock under the wait-die scheme

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$;			
2)		$l_2(A)$; **Waits**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Waits**
5)	$l_1(B)$; $w_1(B)$;		**Wounded**	
6)	$u_1(A)$; $u_1(B)$;			
7)		$l_2(A)$; $l_2(C)$;		
8)		$r_2(C)$; $w_2(A)$;		
9)		$u_2(A)$; $u_2(C)$;		
10)				$l_4(A)$; $l_4(D)$;
11)				$r_4(D)$; $w_4(A)$;
12)				$u_4(A)$; $u_4(D)$;
13)			$l_3(B)$; $r_3(B)$;	
14)			$l_3(C)$; $w_3(C)$;	
15)			$u_3(B)$; $u_3(C)$;	

Figure 19.10: Actions of transactions detecting deadlock under the wound-wait scheme

Why Timestamp-Based Deadlock Detection Works

We claim that in either the wait-die or wound-wait scheme, there can be no cycle in the waits-for graph, and hence no deadlock. Suppose there is a cycle such as $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$. One of the transactions is the oldest, say T_2.

In the wait-die scheme, you can only wait for younger transactions. Thus, it is not possible that T_1 is waiting for T_2, since T_2 is surely older than T_1. In the wound-wait scheme, you can only wait for older transactions. Thus, there is no way T_2 could be waiting for the younger T_3. We conclude that the cycle cannot exist, and therefore there is no deadlock.

On the other hand, when a rollback does occur, wait-die rolls back a transaction that is still in the stage of gathering locks, presumably the earliest phase of the transaction. Thus, although wait-die may roll back more transactions than wound-wait, these transactions tend to have done little work. In contrast, when wound-wait does roll back a transaction, it is likely to have acquired its locks and for substantial processor time to have been invested in its activity. Thus, either scheme may turn out to cause more wasted work, depending on the population of transactions processed.

We should also consider the advantages and disadvantages of both wound-wait and wait-die when compared with a straightforward construction and use of the waits-for graph. The important points are:

- Both wound-wait and wait-die are easier to implement than a system that maintains or periodically constructs the waits-for graph.
- Using the waits-for graph minimizes the number of times we must abort a transaction because of deadlock. If we abort a transaction, there really is a deadlock. On the other hand, either wound-wait or wait-die will sometimes roll back a transaction when there really is no deadlock.

19.2.6 Exercises for Section 19.2

Exercise 19.2.1: For each of the sequences of actions below, assume that shared locks are requested immediately before each read action, and exclusive locks are requested immediately before every write action. Also, unlocks occur immediately after the final action that a transaction executes. Tell what actions are denied, and whether deadlock occurs. Also tell how the waits-for graph evolves during the execution of the actions. If there are deadlocks, pick a transaction to abort, and show how the sequence of actions continues.

a) $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(D)$; $r_4(E)$; $w_3(B)$; $w_2(C)$; $w_4(A)$; $w_1(D)$;

b) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(D)$;

c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(A)$;

d) $r_1(A)$; $r_2(B)$; $w_1(C)$; $w_2(D)$; $r_3(C)$; $w_1(B)$; $w_4(D)$; $w_2(A)$;

Exercise 19.2.2: For each of the action sequences in Exercise 19.2.1, tell what happens under the wound-wait deadlock avoidance system. Assume the order of deadlock-timestamps is the same as the order of subscripts for the transactions, that is, T_1, T_2, T_3, T_4. Also assume that transactions that need to restart do so in the order that they were rolled back.

Exercise 19.2.3: For each of the action sequences in Exercise 19.2.1, tell what happens under the wait-die deadlock avoidance system. Make the same assumptions as in Exercise 19.2.2.

! **Exercise 19.2.4:** Can one have a waits-for graph with a cycle of length n, but no smaller cycle, for any integer $n > 1$? What about $n = 1$, i.e., a loop on a node?

!! **Exercise 19.2.5:** One approach to avoiding deadlocks is to require each transaction to announce all the locks it wants at the beginning, and to either grant all those locks or deny them all and make the transaction wait. Does this approach avoid deadlocks due to locking? Either explain why, or give an example of a deadlock that can arise.

! **Exercise 19.2.6:** Consider the intention-locking system of Section 18.6. Describe how to construct the waits-for graph for this system of lock modes. Especially, consider the possibility that a database element A is locked by different transactions in modes IS and also either S or IX. If a request for a lock on A has to wait, what arcs do we draw?

! **Exercise 19.2.7:** In Section 19.2.5 we pointed out that deadlock-detection methods other than wound-wait and wait-die do not necessarily prevent starvation, where a transaction is repeatedly rolled back and never gets to finish. Give an example of how using the policy of rolling back any transaction that would cause a cycle can lead to starvation. Does requiring that transactions request locks on elements in a fixed order necessarily prevent starvation? What about timeouts as a deadlock-resolution mechanism?

19.3 Long-Duration Transactions

There is a family of applications for which a database system is suitable for maintaining data, but the model of many short transactions on which database concurrency-control mechanisms are predicated, is inappropriate. In this section we shall examine some examples of these applications and the problems that arise. We then discuss a solution based on "compensating transactions" that negate the effects of transactions that were committed, but shouldn't have been.

19.3.1 Problems of Long Transactions

Roughly, a *long transaction* is one that takes too long to be allowed to hold locks that another transaction needs. Depending on the environment, "too long" could mean seconds, minutes, or hours. Three broad classes of applications that involve long transactions are:

1. *Conventional DBMS Applications.* While common database applications run mostly short transactions, many applications require occasional long transactions. For example, one transaction might examine all of a bank's accounts to verify that the total balance is correct. Another application may require that an index be reconstructed occasionally to keep performance at its peak.

2. *Design Systems.* Whether the thing being designed is mechanical like an automobile, electronic like a microprocessor, or a software system, the common element of design systems is that the design is broken into a set of components (e.g., files of a software project), and different designers work on different components simultaneously. We do not want two designers taking a copy of a file, editing it to make design changes, and then writing the new file versions back, because then one set of changes would overwrite the other. Thus, a *check-out-check-in* system allows a designer to "check out" a file and check it in when the changes are finished, perhaps hours or days later. Even if the first designer is changing the file, another designer might want to look at the file to learn something about its contents. If the check-out operation were tantamount to an exclusive lock, then some reasonable and sensible actions would be delayed, possibly for days.

3. *Workflow Systems.* These systems involve collections of processes, some executed by software alone, some involving human interaction, and perhaps some involving human action alone. We shall give shortly an example of office paperwork involving the payment of a bill. Such applications may take days to perform, and during that entire time, some database elements may be subject to change. Were the system to grant an exclusive lock on data involved in a transaction, other transactions could be locked out for days.

Example 19.13: Consider the problem of an employee vouchering travel expenses. The intent of the traveler is to be reimbursed from account A123, and the process whereby the payment is made is shown in Fig. 19.11. The process begins with action A_1, where the traveler's secretary fills out an on-line form describing the travel, the account to be charged, and the amount. We assume in this example that the account is A123, and the amount is \$1000.

The traveler's receipts are sent physically to the departmental authorization office, while the form is sent on-line to an automated action A_2. This process checks that there is enough money in the charged account (A123) and reserves the money for expenditure; i.e., it tentatively deducts \$1000 from the account

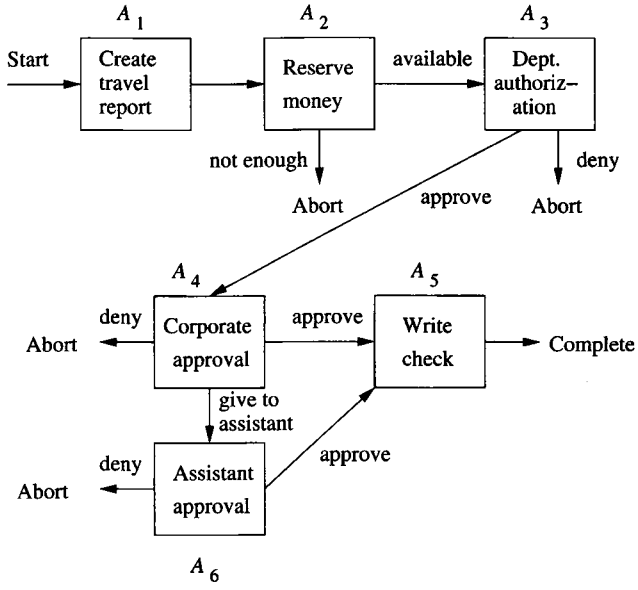

Figure 19.11: Workflow for a traveler requesting expense reimbursement

but does not issue a check for that amount. If there is not enough money in the account, the transaction aborts, and presumably it will restart when either enough money is in the account or after changing the account to be charged.

Action A_3 is performed by the departmental administrator, who examines the receipts and the on-line form. This action might take place the next day. If everything is in order, the form is approved and sent to the corporate administrator, along with the physical receipts. If not, the transaction is aborted. Presumably the traveler will be required to modify the request in some way and resubmit the form.

In action A_4, which may take place several days later, the corporate administrator either approves or denies the request, or passes the form to an assistant, who will then make the decision in action A_5. If the form is denied, the transaction again aborts and the form must be resubmitted. If the form is approved, then at action A_6 the check is written, and the deduction of \$1000 from account A123 is finalized.

However, suppose that the only way we could implement this workflow is by conventional locking. In particular, since the balance of account A123 may be changed by the complete transaction, it has to be locked exclusively at action A_2 and not unlocked until either the transaction aborts or action A_6 completes. This lock may have to be held for days, while the people charged with authorizing the payment get a chance to look at the matter. If so, then there can be no other charges made to account A123, even tentatively. On

the other hand, if there are no controls at all over how account A123 can be accessed, then it is possible that several transactions will reserve or deduct money from the account simultaneously, leading to an overdraft. Thus, some compromise between rigid, long-term locks on one hand, and anarchy on the other, is required. □

19.3.2 Sagas

A *saga* is a collection of actions, such as those of Example 19.13, that together form a long-duration "transaction." That is, a saga consists of:

1. A collection of actions.
2. A directed graph whose nodes are either actions or the *terminal* nodes *Abort* and *Complete*. No arcs leave the terminal nodes.
3. An indication of the node at which the action starts, called the *start node*.

The paths through the graph, from the start node to either of the terminal nodes, represent possible sequences of actions. Those paths that lead to the *Abort* node represent sequences of actions that cause the overall transaction to be rolled back, and these sequences of actions should leave the database unchanged. Paths to the *Complete* node represent successful sequences of actions, and all the changes to the database system that these actions perform will remain in the database.

Example 19.14: The paths in the graph of Fig. 19.11 that lead to the *Abort* node are: A_1A_2, $A_1A_2A_3$, $A_1A_2A_3A_4$, and $A_1A_2A_3A_4A_5$. The paths that lead to the *Complete* node are $A_1A_2A_3A_4A_6$, and $A_1A_2A_3A_4A_5A_6$. Notice that in this case the graph has no cycles, so there are a finite number of paths leading to a terminal node. However, in general, a graph can have cycles and an infinite number of paths. □

Concurrency control for sagas is managed by two facilities:

1. Each action may be considered itself a (short) transaction, that when executed uses a conventional concurrency-control mechanism, such as locking. For instance, A_2 may be implemented to (briefly) obtain a lock on account A123, decrement the amount indicated on the travel voucher, and release the lock. This locking prevents two transactions from trying to write new values of the account balance at the same time, thereby losing the effect of the first to write and making money "appear by magic."
2. The overall transaction, which can be any of the paths to a terminal node, is managed through the mechanism of "compensating transactions," which are inverses to the transactions at the nodes of the saga. Their job is to roll back the effect of a committed action in a way that does not depend on what has happened to the database between the time the action was executed and the time the compensating transaction is executed.

When are Database States "The Same"?

When discussing compensating transactions, we should be careful about what it means to return the database to "the same" state that it had before. We had a taste of the problem when we discussed logical logging for B-trees in Example 19.8. There we saw that if we "undid" an operation, the state of the B-tree might not be identical to the state before the operation, but would be equivalent to it as far as access operations on the B-tree were concerned. More generally, executing an action and its compensating transaction might not restore the database to a state literally identical to what existed before, but the differences must not be detectable by whatever application programs the database supports.

19.3.3 Compensating Transactions

In a saga, each action A has a *compensating transaction*, which we denote A^{-1}. Intuitively, if we execute A, and later execute A^{-1}, then the resulting database state is the same as if neither A nor A^{-1} had executed. More formally:

- If D is any database state, and $B_1B_2\cdots B_n$ is any sequence of actions and compensating transactions (whether from the saga in question or any other saga or transaction that may legally execute on the database) then the same database states result from running the sequences $B_1B_2\cdots B_n$ and $AB_1B_2\cdots B_nA^{-1}$ starting in database state D.

If a saga execution leads to the *Abort* node, then we roll back the saga by executing the compensating transactions for each executed action, in the reverse order of those actions. By the property of compensating transactions stated above, the effect of the saga is negated, and the database state is the same as if it had never happened. An explanation of why the effect is guaranteed to be negated is given in Section 19.3.4

Example 19.15: Let us consider the actions in Fig. 19.11 and see what the compensating transactions for A_1 through A_6 might be. First, A_1 creates an online document. If the document is stored in the database, then A_1^{-1} must remove it from the database. Notice that this compensation obeys the fundamental property for compensating transactions: If we create the document, do any sequence of actions α (including deletion of the document if we wish), then the effect of $A_1\alpha A_1^{-1}$ is the same as the effect of α.

A_2 must be implemented carefully. We "reserve" the money by deducting it from the account. The money will stay removed unless restored by the compensating transaction A_2^{-1}. We claim that this A_2^{-1} is a correct compensating transaction if the usual rules for how accounts may be managed are followed. To appreciate the point, it is useful to consider a similar transaction where the

obvious compensation will not work; we consider such a case in Example 19.16, next.

The actions A_3, A_4, and A_6 each involve adding an approval to a form. Thus, their compensating transactions can remove that approval.[3]

Finally, A_5, which writes the check, does not have an obvious compensating transaction. In practice none is needed, because once A_5 is executed, this saga cannot be rolled back. However, technically A_5 does not affect the database anyway, since the money for the check was deducted by A_2. Should we need to consider the "database" as the larger world, where effects such as cashing a check affected the database, then we would have to design A_5^{-1} to first try to cancel the check, next write a letter to the payee demanding the money back, and if all remedies failed, restoring the money to the account by declaring a loss due to a bad debt. □

Next, let us take up the example, alluded to in Example 19.15, where a change to an account cannot be compensated by an inverse change. The problem is that accounts normally are not allowed to go negative.

Example 19.16: Suppose B is a transaction that adds \$1000 to an account that has \$2000 in it initially, and B^{-1} is the compensating transaction that removes the same amount of money. Also, it is reasonable to assume that transactions may fail if they try to delete money from an account and the balance would thereby become negative. Let C be a transaction that deletes \$2500 from the same account. Then $BCB^{-1} \not\equiv C$. The reason is that C by itself fails, and leaves the account with \$2000, while if we execute B then C, the account is left with \$500, whereupon B^{-1} fails.

Our conclusion that a saga with arbitrary transfers among accounts and a rule about accounts never being allowed to go negative cannot be supported simply by compensating transactions. Some modification to the system must be done, e.g., allowing negative balances in accounts. □

19.3.4 Why Compensating Transactions Work

Let us say that two sequences of actions are *equivalent* ($\equiv$) if they take any database state D to the same state. The fundamental assumption about compensating transactions can be stated:

- If A is any action and α is any sequence of legal actions and compensating transactions, then $A\alpha A^{-1} \equiv \alpha$.

Now, we need to show that if a saga execution $A_1A_2\cdots A_n$ is followed by its compensating transactions in reverse order, $A_n^{-1}\cdots A_2^{-1}A_1^{-1}$, with any intervening actions whatsoever, then the effect is as if neither the actions nor the compensating transactions executed. The proof is an induction on n.

[3] In the saga of Fig. 19.11, the only time these actions are compensated is when we are going to delete the form anyway, but the definition of compensating transactions require that they work in isolation, regardless of whether some other compensating transaction was going to make their changes irrelevant.

BASIS: If $n = 1$, then the sequence of all actions between A_1 and its compensating transaction A_1^{-1} looks like $A_1\alpha A_1^{-1}$. By the fundamental assumption about compensating transactions, $A_1\alpha A_1^{-1} \equiv \alpha$.

INDUCTION: Assume the statement for paths of up to $n-1$ actions, and consider a path of n actions, followed by its compensating transactions in reverse order, with any other transactions intervening. The sequence looks like

$$A_1\alpha_1 A_2\alpha_2 \cdots \alpha_{n-1} A_n \beta A_n^{-1} \gamma_{n-1} \cdots \gamma_2 A_2^{-1} \gamma_1 A_1^{-1} \tag{19.1}$$

where all Greek letters represent sequences of zero or more actions. By the definition of compensating transaction, $A_n\beta A_n^{-1} \equiv \beta$. Thus, (19.1) is equivalent to

$$A_1\alpha_1 A_2\alpha_2 \cdots A_{n-1}\alpha_{n-1}\beta\gamma_{n-1} A_{n-1}^{-1} \gamma_{n-2} \cdots \gamma_2 A_2^{-1} \gamma_1 A_1^{-1} \tag{19.2}$$

By the inductive hypothesis, expression (19.2) is equivalent to

$$\alpha_1\alpha_2 \cdots \alpha_{n-1}\beta\gamma_{n-1} \cdots \gamma_2\gamma_1$$

since there are only $n-1$ actions in (19.2). That is, the saga and its compensation leave the database state the same as if the saga had never occurred.

19.3.5 Exercises for Section 19.3

! **Exercise 19.3.1:** The process of "uninstalling" software can be thought of as a compensating transaction for the action of installing the same software. In a simple model of installing and uninstalling, suppose that an action consists of *loading* one or more files from the source (e.g., a CD-ROM) onto the hard disk of the machine. To load a file f, we copy f from CD-ROM. If there was a file f' with the same path name, we back up f' before replacement. To distinguish files with the same path name, we may assume each file has a timestamp.

a) What is the compensating transaction for the action that loads file f? Consider both the case where no file with that path name existed, and where there was a file f' with the same path name.

b) Explain why your answer to (a) is guaranteed to compensate. *Hint*: Consider carefully the case where after replacing f' by f, a later action replaces f by another file with the same path name.

! **Exercise 19.3.2:** Describe the process of booking an airline seat as a saga. Consider the possibility that the customer will query about a seat but not book it. The customer may book the seat, but cancel it, or not pay for the seat within the required time limit. The customer may or may not show up for the flight. For each action, describe the corresponding compensating transaction.

19.4 Summary of Chapter 19

- *Dirty Data*: Data that has been written, either into main-memory buffers or on disk, by a transaction that has not yet committed is called "dirty."
- *Cascading Rollback*: A combination of logging and concurrency control that allows a transaction to read dirty data may have to roll back transactions that read such data from a transaction that later aborts.
- *Strict Locking*: The strict locking policy requires transactions to hold their locks (except for shared-locks) until not only have they committed, but the commit record on the log has been flushed to disk. Strict locking guarantees that no transaction can read dirty data, even retrospectively after a crash and recovery.
- *Group Commit*: We can relax the strict-locking condition that requires commit records to reach disk if we assure that log records are written to disk in the order that they are written. There is still then a guarantee of no dirty reads, even if a crash and recovery occurs.
- *Restoring Database State After an Abort*: If a transaction aborts but has written values to buffers, then we can restore old values either from the log or from the disk copy of the database. If the new values have reached disk, then the log may still be used to restore the old value.
- *Logical Logging*: For large database elements such as disk blocks, it saves much space if we record old and new values on the log incrementally, that is, by indicating only the changes. In some cases, recording changes logically, that is, in terms of an abstraction of what blocks contain, allows us to restore state logically after a transaction abort, even if it is impossible to restore the state literally.
- *Deadlocks*: These occur when each of a set of transactions is waiting for a resource, such as a lock, currently held by another transaction in the set.
- *Waits-For Graphs*: Create a node for each waiting transaction, with an arc to the transaction it is waiting for. The existence of a deadlock is the same as the existence of one or more cycles in the waits-for graph. We can avoid deadlocks if we maintain the waits-for graph and abort any transaction whose waiting would cause a cycle.
- *Deadlock Avoidance by Ordering Resources*: Requiring transactions to acquire resources according to some lexicographic order of the resources will prevent a deadlock from arising.
- *Timestamp-Based Deadlock Avoidance*: Other schemes maintain a timestamp and base their abort/wait decision on whether the requesting transaction is newer or older than the one with the resource it wants. In the

wait-die scheme, an older requesting transaction waits, and a newer one is rolled back with the same timestamp. In the wound-wait scheme, a newer transaction waits and an older one forces the transaction with the resource to roll back and give up the resource.

- ✦ *Sagas*: When transactions involve long-duration steps that may take hours or days, conventional locking mechanisms may limit concurrency too much. A saga consists of a network of actions, each of which may lead to one or more other actions, to the completion of the entire saga, or to a requirement that the saga abort.

- ✦ *Compensating Transactions*: For a saga to make sense, each action must have a compensating action that will undo the effects of the first action on the database state, while leaving intact any other actions that have been made by other sagas that have completed or are currently in operation. If a saga aborts, the appropriate sequence of compensating actions is executed.

19.5 References for Chapter 19

Some useful general sources for topics covered here are [2], [1], and [7]. The material on logical logging follows [6].

Deadlock prevention was surveyed in [5]; the waits-for graph is from there. The wait-die and wound-wait schemes are from [8].

Long transactions were introduced by [4]. Sagas were described in [3].

1. N. S. Barghouti and G. E. Kaiser, "Concurrency control in advanced database applications," *Computing Surveys* **23**:3 (Sept., 1991), pp. 269–318.

2. S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, New York, 1984.

3. H. Garcia-Molina and K. Salem, "Sagas," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1987), pp. 249–259.

4. J. N. Gray, "The transaction concept: virtues and limitations," *Intl. Conf. on Very Large Databases* (1981), pp. 144–154.

5. R. C. Holt, "Some deadlock properties of computer systems," *Computing Surveys* **4**:3 (1972), pp. 179–196.

6. C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. on Database Systems* **17**:1 (1992), pp. 94–162.

7. M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, Englewood Cliffs NJ, 1999.

8. D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, "System-level concurrency control for distributed database systems," *ACM Trans. on Database Systems* **3**:2 (1978), pp. 178–198.

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.

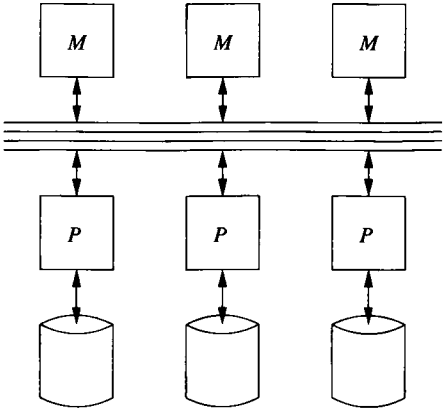

Figure 20.1: A shared-memory machine

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.

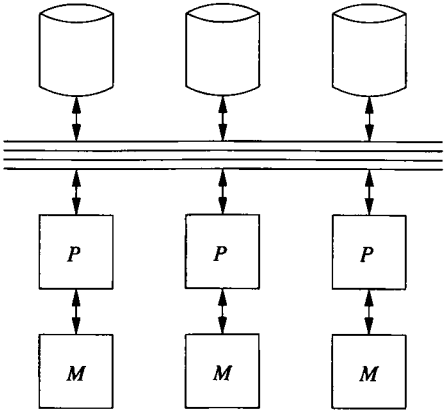

Figure 20.2: A shared-disk machine

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.

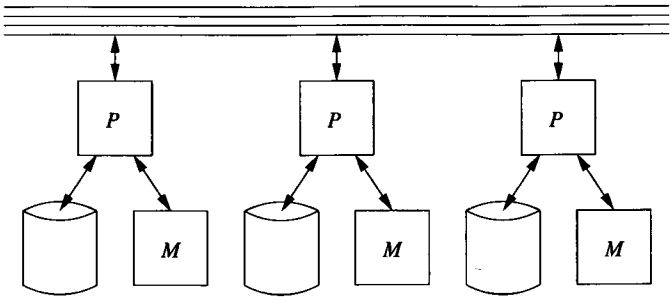

Figure 20.3: A shared-nothing machine

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.

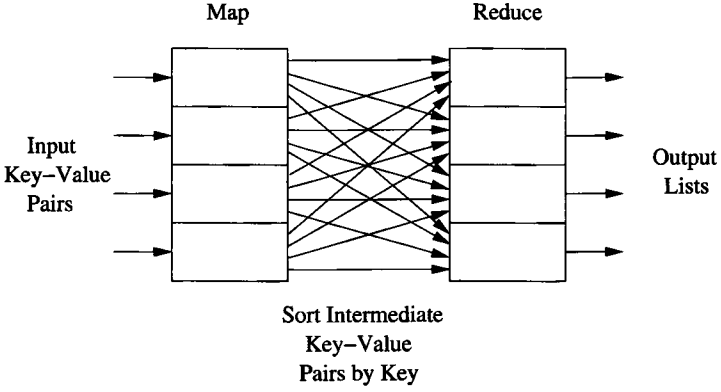

Figure 20.4: Execution of map and reduce functions

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.

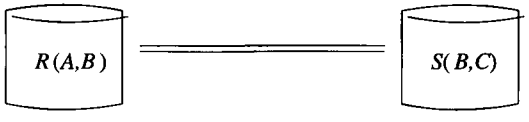

Figure 20.5: Joining relations at different nodes of a network

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 $R \ltimes S = R \bowtie (\pi_Y(S))$. 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.

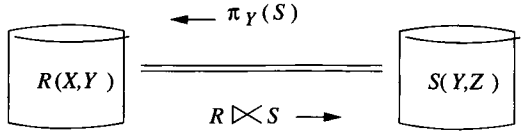

Figure 20.6: Exploiting the semijoin to minimize communication

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
1	1
2	2
.	.
.	.
.	.
n	n

R

B	C
1	1
2	2
.	.
.	.
.	.
n	n

S

C	A
1	2
2	3
.	.
.	.
.	.
$n-1$	n

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.

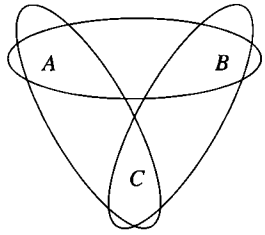

Figure 20.8: The hypergraph for Example 20.7

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. □

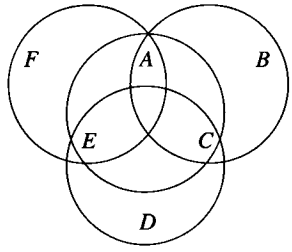

Figure 20.9: An acyclic hypergraph

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

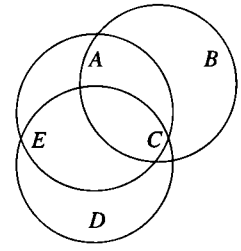

Figure 20.10: After one ear reduction

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.

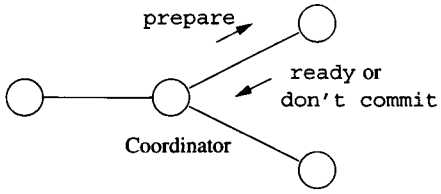

Figure 20.12: Messages in phase 1 of two-phase commit

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.

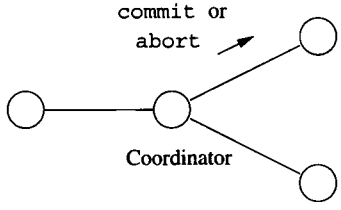

Figure 20.13: Messages in phase 2 of two-phase commit

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.

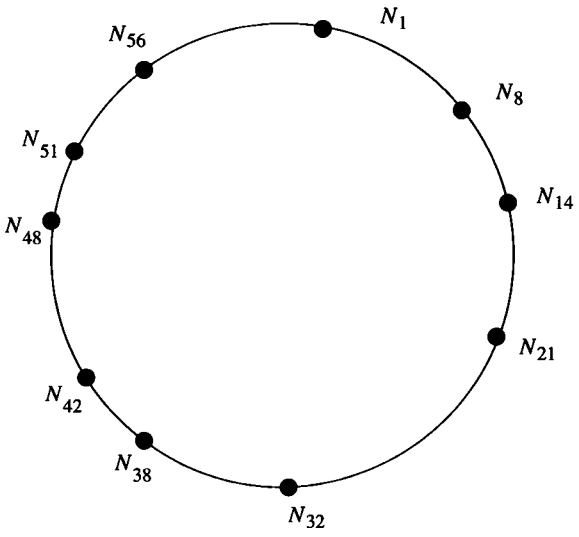

Figure 20.14: A chord circle

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.

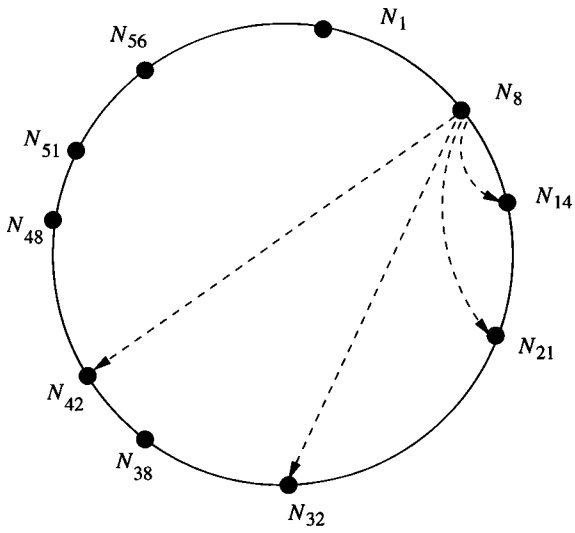

Figure 20.16: Links in the finger table for N_8

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. □

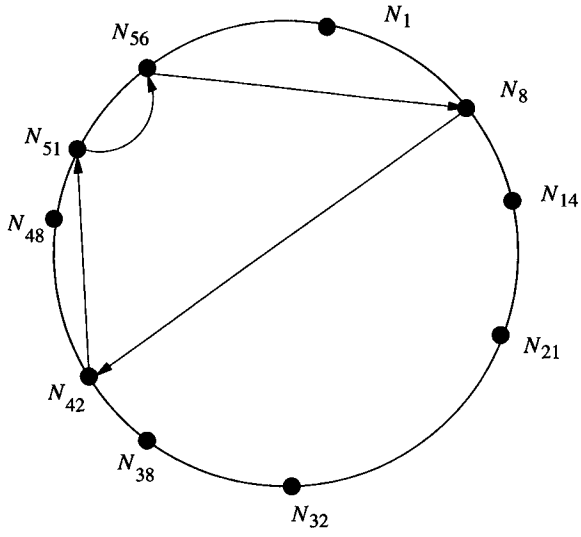

Figure 20.17: Message sequence in the search for (K, V)

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. □

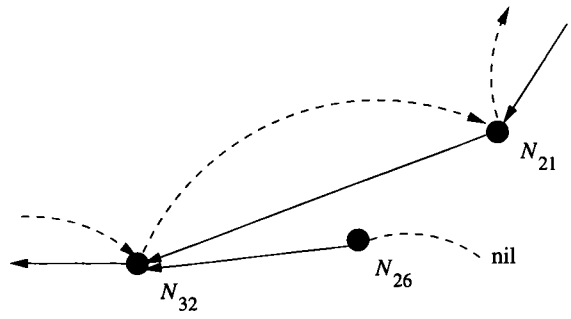

Figure 20.18: Adding node N_{26} to the network of peers

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}.

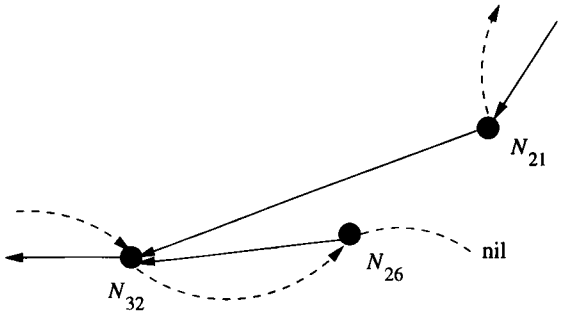

Figure 20.19: After making N_{26} the predecessor of N_{32}

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

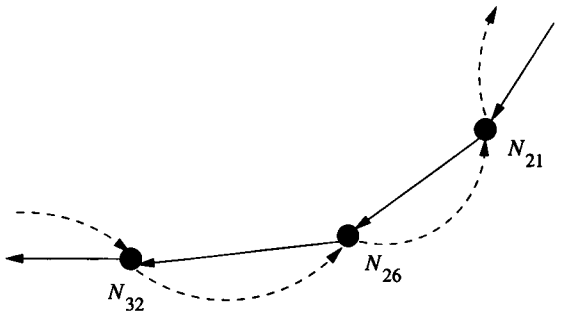

Figure 20.20: After N_{21} runs the stabilization algorithm

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].

1. P. A. Bernstein and N. Goodman, "The power of natural semijoins," *SIAM J. Computing* **10**:4 (1981), pp. 751–771.

2. J. Dean and S. Ghemawat, "MapReduce: simplified processing on large clusters," *Sixth Symp. on Operating System Design and Implementation*, 2004.

3. H. Garcia-Molina, "Performance comparison of update algorithms for distributed databases," TR Nos. 143 and 146, Computer Systems Laboratory, Stanford Univ., 1979.

4. H. Garcia-Molina, "Elections in a distributed computer system," *IEEE Trans. on Computers* **C-31**:1 (1982), pp. 48–59.

5. M. H. Graham, "On the universal relation," Technical report, Dept. of CS, Univ. of Toronto, 1979.

6. Hadoop home page `lucene.apache.org/hadoop`.

7. B. Lampson and H. Sturgis, "Crash recovery in a distributed data storage system," Technical report, Xerox Palo Alto Research Center, 1976.

8. D. E. Shaw, "Knowledge-based retrieval on a relational database machine," Ph. D. thesis, Dept. of CS, Stanford Univ. (1980).

9. D. Skeen, "Nonblocking commit protocols," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1981), pp. 133–142.

10. I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalabale peer-to-peer lookup service for Internet applications," *Proc. ACM SIGCOMM* (2001) pp. 149–160.

11. M. Stonebraker, "Retrospection on a database system," *ACM Trans. on Database Systems* **5**:2 (1980), pp. 225–240.

12. R. H. Thomas, "A majority consensus approach to concurrency control," *ACM Trans. on Database Systems* 4:2 (1979), pp. 180–219.

13. C. T. Yu and M. Z. Ozsoyoglu, "An algorithm for tree-query membership of a distributed query," *Proc. IEEE COMPSAC* (1979), pp. 306–312.

Part V

Other Issues in Management of Massive Data

Chapter 21

Information Integration

Information integration is the process of taking several databases or other information sources and making the data in these sources work together as if they were a single database. The integrated database may be physical (a "warehouse") or virtual (a "mediator" or "middleware" that may be queried even though its does not exist physically). The sources may be conventional databases or other types of information, such as collections of Web pages.

We begin by exploring the ways in which seemingly similar databases can actually embody conflicts that are hard to resolve correctly. The solution lies in the design of "wrappers" — translators between the schema and data values at a source and the schema and data values at the integrated database.

Information-integration systems require special kinds of query-optimization techniques for their efficient operation. Mediator systems can be divided into two classes: "global-as-view" (the data at the integrated database is defined by how it is constructed from the sources) and "local-as-view" (the content of the sources is defined in terms of the schema that the integrated database supports). We examine capability-based optimization for global-as-view mediators. We also consider local-as-view mediation, which requires effort even to figure out how to compose the answer to a query from defined views, but which offers advantages in flexibility of operation.

In the last section, we examine another important issue in information integration, called "entity resolution." Different information sources may talk about the same entities (e.g., people) but contain discrepancies such as misspelled names or out-of-date addresses. We need to make a best estimate of which data elements at the different sources actually refer to the same entity.

21.1 Introduction to Information Integration

In this section, we discuss the ways in which information-integration is essential for many database applications. We then sample some of the problems that make information integration difficult.

21.1.1 Why Information Integration?

If we could start anew with an architecture and schema for all the data in the world, and we could put that data in a single database, there would be no need for information integration. However, in the real world, matters are rather different.

- Databases are created independently, even if they later need to work together.
- The use of databases evolves, so we cannot design a database to support every possible future use.

To see the need for information integration, we shall consider two typical scenarios: building applications for a university and integrating employee databases. In both scenarios, a key problem is that the overall data-management system must make use of *legacy* data sources — databases that were created independently of any other data source. Each legacy source is used by applications that expect the structure of "their" database not to change, so modification of the schema or data of legacy sources is not an option.

University Databases

As databases came into common use, each university started using them for several functions that were once done by hand. Here is a typical scenario. The Registrar builds a database of courses, and uses it to record the courses each student took and their grades. Applications are built using this database, such as a transcript generator.

The Bursar builds another database for recording tuition payments by students. The Human Resources Department builds a database for recording employees, including those students with teaching-assistant or research-assistant jobs. Applications include generation of payroll checks, calculation of taxes and social-security payments to the government, and many others. The Grants Office builds a database to keep track of expenditures on grants, which includes salaries to certain faculty, students, and staff. It may also include information about biohazards, use of human subjects, and many other matters related to research projects.

Pretty soon, the university realizes that all these databases are not helping nearly as much as they could, and are sometimes getting in the way. For example, suppose we want to make sure that the Registrar does not record grades for students that the Bursar says did not pay tuition. Someone has to get a list of students who paid tuition from the Bursar's database and compare that with a list of students from the Registrar's database. As another example, when Sally is appointed on grant 123 as a research assistant, someone needs to tell the Grants Office that her salary should be charged to grant 123. Someone also needs to tell Human Resources that they should pay her salary. And the salaries in the two databases had better be exactly the same.

So at some point, the university decides that it needs one database for all functions. The first thought might be: start over. Build one database that contains all the information of all the legacy databases and rewrite all the applications to use the new database. This approach has been tried, with great pain resulting. In addition to paying for a very expensive software-architecture task, the university has to run both the old and new systems in parallel for a long time to see that the new system actually works. And when they cut over to the new system, the users find that the applications do not work in the accustomed way, and turmoil results.

A better way is to build a layer of abstraction, called *middleware*, on top of all the legacy databases and allow the legacy databases to continue serving their current applications. The layer of abstraction could be relational views — either virtual or materialized. Then, SQL can be used to "query" the middleware layer. Often, this layer is defined by a collection of classes and queried in an object-oriented language. Or the middleware layer could use XML documents, which are queried using XQuery. We mentioned in Section 9.1 that this middleware may be an important component of the application tier in a 3-tier architecture, although we did not show it explicitly.

Once the middleware layer is built, new applications can be written to access this layer for data, while the legacy applications continue to run using the legacy databases. For example, we can write a new application that enters grades for students only if they have paid their tuition. Another new application could appoint a research assistant by getting their name, grant, and salary from the user. This application would then enter the name and salary into the Human-Resources database and the name, salary, and grant into the Grants-Office database.

Integrating Employee Databases

Compaq bought DEC and Tandem, and then Hewlett-Packard bought Compaq. Each company had a database of employees. Because the companies were previously independent, the schemas and architecture of their databases naturally differed. Moreover, each company actually had many databases about employees, and these databases probably differed on matters as basic as who is an employee. For example, the Payroll Department would not include retirees, but might include contractors. The Benefits Department would include retirees but not contractors. The Safety Office would include not only regular employees and contractors, but the employees of the company that runs the cafeteria.

For reasons we discussed in connection with the university database, it may not be practical to shut down these legacy databases and with them all the applications that run on them. However, it is possible to create a middleware layer that holds — virtually or physically — all information available for each employee.

21.1.2 The Heterogeneity Problem

When we try to connect information sources that were developed independently, we invariably find that the sources differ in many ways, even if they are intended to store the same kinds of data. Such sources are called *heterogeneous*, and the problem of integrating them is referred to as the *heterogeneity problem*. We shall introduce a running example of an automobile database and then discuss examples of the different levels at which heterogeneity can make integration difficult.

Example 21.1: The Aardvark Automobile Co. has 1000 dealers, each of which maintains a database of their cars in stock. Aardvark wants to create an integrated database containing the information of all 1000 sources.[1] The integrated database will help dealers locate a particular model at another dealer, if they don't have one in stock. It also can be used by corporate analysts to predict the market and adjust production to provide the models most likely to sell.

However, the dealers' databases may differ in a great number of ways. We shall enumerate below the most important ways and give some examples in terms of the Aardvark database. □

Communication Heterogeneity

Today, it is common to allow access to your information using the HTTP protocol that drives the Web. However, some dealers may not make their databases available on the Web, but instead accept remote accesses via remote procedure calls or anonymous FTP, for instance.

Query-Language Heterogeneity

The manner in which we query or modify a dealer's database may vary. It would be nice if the database accepted SQL queries and modifications, but not all do. Of those that do, each accepts a dialect of SQL — the version supported by the vendor of the dealer's DBMS. Another dealer may not have a relational database at all. They could use an Excel Spreadsheet, or an object-oriented database, or an XML database using XQuery as the language.

Schema Heterogeneity

Even assuming that all the dealers use a relational DBMS supporting SQL as the query language, we can find many sources of heterogeneity. At the highest level, the schemas can differ. For example, one dealer might store cars in a single relation that looks like:

[1] Most real automobile companies have similar facilities in place, and the history of their development may be different from our example; e.g., the centralized database may have come first, with dealers later able to download relevant portions to their own database. However, this scenario serves as an example of what companies in many industries are attempting today.

```
Cars(serialNo, model, color, autoTrans, navi,...)
```

with one boolean-valued attribute for every possible option. Another dealer might use a schema in which options are separated out into a second relation, such as:

```
Autos(serial, model, color)
Options(serial, option)
```

Notice that not only is the schema different, but apparently equivalent relation or attribute names have changed: `Cars` becomes `Autos`, and `serialNo` becomes `serial`.

Moreover, one dealer's schema might not record information that most of the other dealers provide. For instance, one dealer might not record colors at all. To deal with missing values, sometimes we can use `NULL`'s or default values. However, because missing schema elements are a common problem, there is a trend toward using semistructured data such as XML as the data model for integrating middleware.

Data type differences

Serial numbers might be represented by character strings of varying length at one source and fixed length at another. The fixed lengths could differ, and some sources might use integers rather than character strings.

Value Heterogeneity

The same concept might be represented by different constants at different sources. The color black might be represented by an integer code at one source, the string `BLACK` at another, and the code `BL` at a third. The code `BL` might stand for "blue" at yet another source.

Semantic Heterogeneity

Terms may be given different interpretations at different sources. One dealer might include trucks in the `Cars` relation, while another puts only automobile data in the `Cars` relation. One dealer might distinguish station wagons from minivans, while another doesn't.

21.2 Modes of Information Integration

There are several ways that databases or other distributed information sources can be made to work together. In this section, we consider the three most common approaches:

1. *Federated databases.* The sources are independent, but one source can call on others to supply information.

2. *Warehousing.* Copies of data from several sources are stored in a single database, called a (*data*) *warehouse.* Possibly, the data stored at the warehouse is first processed in some way before storage; e.g., data may be filtered, and relations may be joined or aggregated. The warehouse is updated periodically, perhaps overnight. As the data is copied from the sources, it may need to be transformed in certain ways to make all data conform to the schema at the warehouse.

3. *Mediation.* A mediator is a software component that supports a *virtual database*, which the user may query as if it were *materialized* (physically constructed, like a warehouse). The mediator stores no data of its own. Rather, it translates the user's query into one or more queries to its sources. The mediator then synthesizes the answer to the user's query from the responses of those sources, and returns the answer to the user.

We shall introduce each of these approaches in turn. One of the key issues for all approaches is the way that data is transformed when it is extracted from an information source. We discuss the architecture of such transformers — called *wrappers*, *adapters*, or *extractors* — in Section 21.3.

21.2.1 Federated Database Systems

Perhaps the simplest architecture for integrating several databases is to implement one-to-one connections between all pairs of databases that need to talk to one another. These connections allow one database system D_1 to query another D_2 in terms that D_2 can understand. The problem with this architecture is that if n databases each need to talk to the $n-1$ other databases, then we must write $n(n-1)$ pieces of code to support queries between systems. The situation is suggested in Fig. 21.1. There, we see four databases in a federation. Each of the four needs three components, one to access each of the other three databases.

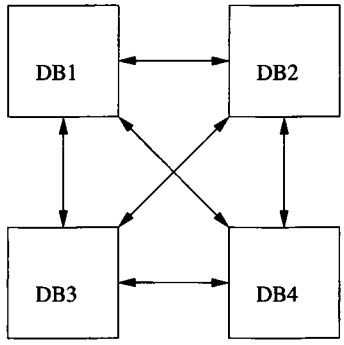

Figure 21.1: A federated collection of four databases needs 12 components to translate queries from one to another

Nevertheless, a federated system may be the easiest to build in some circumstances, especially when the communications between databases are limited in nature. An example will show how the translation components might work.

Example 21.2: Suppose the Aardvark Automobile dealers want to share inventory, but each dealer only needs to query the database of a few local dealers to see if they have a needed car. To be specific, consider Dealer 1, who has a relation

```
NeededCars(model, color, autoTrans)
```

whose tuples represent cars that customers have requested, by model, color, and whether or not they want an automatic transmission (`'yes'` or `'no'` are the possible values). Dealer 2 stores inventory in the two-relation schema discussed in Example 21.1:

```
Autos(serial, model, color)
Options(serial, option)
```

Dealer 1 writes an application program that queries Dealer 2 remotely for cars that match each of the cars described in `NeededCars`. Figure 21.2 is a sketch of a program with embedded SQL that would find the desired cars. The intent is that the embedded SQL represents remote queries to the Dealer 2 database, with results returned to Dealer 1. We use the convention from standard SQL of prefixing a colon to variables that represent constants retrieved from a database.

These queries address the schema of Dealer 2. If Dealer 1 also wants to ask the same question of Dealer 3, who uses the first schema discussed in Example 21.1, with a single relation

```
Cars(serialNo, model, color, autoTrans,...)
```

the query would look quite different. But each query works properly for the database to which it is addressed. □

21.2.2 Data Warehouses

In the *data warehouse* integration architecture, data from several sources is extracted and combined into a *global* schema. The data is then stored at the warehouse, which looks to the user like an ordinary database. The arrangement is suggested by Fig. 21.3, although there may be many more than the two sources shown.

Once the data is in the warehouse, queries may be issued by the user exactly as they would be issued to any database. There are at least three approaches to constructing the data in the warehouse:

1. The warehouse is periodically closed to queries and reconstructed from the current data in the sources. This approach is the most common, with reconstruction occurring once a night or at even longer intervals.

```
for(each tuple (:m, :c, :a) in NeededCars) {
    if(:a = TRUE) { /* automatic transmission wanted */
        SELECT serial FROM Autos, Options
        WHERE Autos.serial = Options.serial AND
              Options.option = 'autoTrans' AND
              Autos.model = :m AND Autos.color = :c;
    }
    else { /* automatic transmission not wanted */
        SELECT serial
        FROM Autos
        WHERE Autos.model = :m AND Autos.color = :c AND
              NOT EXISTS (
                  SELECT * FROM Options
                  WHERE serial = Autos.serial AND
                        option = 'autoTrans'
              );
    }
}
```

Figure 21.2: Dealer 1 queries Dealer 2 for needed cars

2. The warehouse is updated periodically (e.g., each night), based on the changes that have been made to the sources since the last time the warehouse was modified. This approach can involve smaller amounts of data, which is very important if the warehouse needs to be modified in a short period of time, and the warehouse is large (multiterabyte warehouses are in common use). The disadvantage is that calculating changes to the warehouse, a process called *incremental update*, is complex, compared with algorithms that simply construct the warehouse from scratch.

Note that either of these approaches allow the warehouse to get out of date. However, it is generally too expensive to reflect immediately, at the warehouse, every change to the underlying databases.

Example 21.3: Suppose for simplicity that there are only two dealers in the Aardvark system, and they respectively use the schemas

```
Cars(serialNo, model, color, autoTrans, navi,...)
```

and

```
Autos(serial, model, color)
Options(serial, option)
```

We wish to create a warehouse with the schema

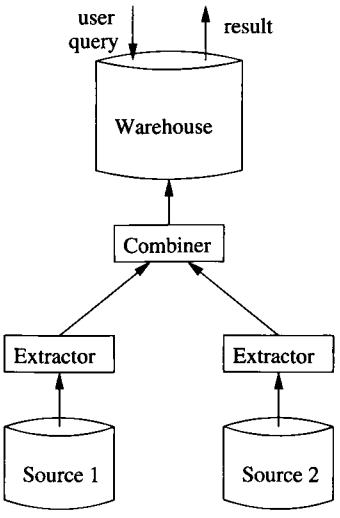

Figure 21.3: A data warehouse stores integrated information in a separate database

```
AutosWhse(serialNo, model, color, autoTrans, dealer)
```

That is, the global schema is like that of the first dealer, but we record only the option of having an automatic transmission, and we include an attribute that tells which dealer has the car.

The software that extracts data from the two dealers' databases and populates the global schema can be written as SQL queries. The query for the first dealer is simple:

```
INSERT INTO AutosWhse(serialNo, model, color,
        autoTrans, dealer)
    SELECT serialNo, model, color, autoTrans, 'dealer1'
    FROM Cars;
```

The extractor for the second dealer is more complex, since we have to decide whether or not a given car has an automatic transmission. We leave this SQL code as an exercise.

In this simple example, the combiner, shown in Fig. 21.3, for the data extracted from the sources is not needed. Since the warehouse is the union of the relations extracted from each source, the data may be loaded directly into the warehouse. However, many warehouses perform operations on the relations that they extract from each source. For instance relations extracted from two sources might be joined, and the result put at the warehouse. Or we might take the union of relations extracted from several sources and then aggregate

the data of this union. More generally, several relations may be extracted from each source, and different relations combined in different ways. □

21.2.3 Mediators

A mediator supports a virtual view, or collection of views, that integrates several sources in much the same way that the materialized relation(s) in a warehouse integrate sources. However, since the mediator doesn't store any data, the mechanics of mediators and warehouses are rather different. Figure 21.4 shows a mediator integrating two sources; as for warehouses, there would typically be more than two sources. To begin, the user or application program issues a query to the mediator. Since the mediator has no data of its own, it must get the relevant data from its sources and use that data to form the answer to the user's query.

Thus, we see in Fig. 21.4 the mediator sending a query to each of its wrappers, which in turn send queries to their corresponding sources. The mediator may send several queries to a wrapper, and may not query all wrappers. The results come back and are combined at the mediator; we do not show an explicit combiner component as we did in the warehouse diagram, Fig. 21.3, because in the case of the mediator, the combining of results from the sources is one of the tasks performed by the mediator.

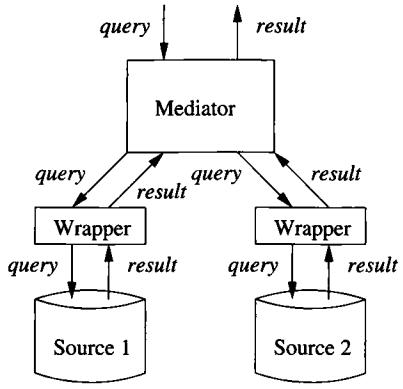

Figure 21.4: A mediator and wrappers translate queries into the terms of the sources and combine the answers

Example 21.4: Let us consider a scenario similar to that of Example 21.3, but use a mediator. That is, the mediator integrates the same two automobile sources into a view that is a single relation with schema:

```
AutosMed(serialNo, model, color, autoTrans, dealer)
```

Suppose the user asks the mediator about red cars, with the query:

```
SELECT serialNo, model
FROM AutosMed
WHERE color = 'red';
```

The mediator, in response to this user query, can forward the same query to each of the two wrappers. The way that wrappers can be designed and implemented to handle queries like this one is the subject of Section 21.3. In more complex scenarios, the mediator would first have to break the query into pieces, each of which is sent to a subset of the wrappers. However, in this case, the translation work can be done by the wrappers alone.

The wrapper for Dealer 1 translates the query into the terms of that dealer's schema, which we recall is

```
Cars(serialNo, model, color, autoTrans, navi,...)
```

A suitable translation is:

```
SELECT serialNo, model
FROM Cars
WHERE color = 'red';
```

An answer, which is a set of `serialNo-model` pairs, will be returned to the mediator by the first wrapper.

At the same time, the wrapper for Dealer 2 translates the same query into the schema of that dealer, which is:

```
Autos(serial, model, color)
Options(serial, option)
```

A suitable translated query for Dealer 2 is almost the same:

```
SELECT serial, model
FROM Autos
WHERE color = 'red';
```

It differs from the query at Dealer 1 only in the name of the relation queried, and in one attribute. The second wrapper returns to the mediator a set of `serial-model` pairs, which the mediator interprets as `serialNo-model` pairs. The mediator takes the union of these sets and returns the result to the user. □

There are several options, not illustrated by Example 21.4, that a mediator may use to answer queries. For instance, the mediator may issue one query to one source, look at the result, and based on what is returned, decide on the next query or queries to issue. This method would be appropriate, for instance, if the user query asked whether there were any Aardvark "Gobi" model sport-utility vehicles available in blue. The first query could ask Dealer 1, and only if the result was an empty set of tuples would a query be sent to Dealer 2.

21.2.4 Exercises for Section 21.2

! **Exercise 21.2.1:** Computer company A keeps data about the PC models it sells in the schema:

```
Computers(number, proc, speed, memory, hd)
Monitors(number, screen, maxResX, maxResY)
```

For instance, the tuple (123, `Athlon64`, 3.1, 512, 120) in `Computers` means that model 123 has an Athlon 64 processor running at 3.1 gigahertz, with 512M of memory and a 120G hard disk. The tuple (456, 19, 1600, 1050) in `Monitors` means that model 456 has a 19-inch screen with a maximum resolution of 1600×1050.

Computer company B only sells complete systems, consisting of a computer and monitor. Its schema is

```
Systems(id, processor, mem, disk, screenSize)
```

The attribute `processor` is the speed in gigahertz; the type of processor (e.g., Athlon 64) is not recorded. Neither is the maximum resolution of the monitor recorded. Attributes `id`, `mem`, and `disk` are analogous to `number`, `memory`, and `hd` from company A, but the disk size is measured in megabytes instead of gigabytes.

a) If company A wants to insert into its relations information about the corresponding items from B, what SQL insert statements should it use?

b) If Company B wants to insert into `Systems` as much information about the systems that can be built from computers and monitors made by A, what SQL statements best allow this information to be obtained?

! **Exercise 21.2.2:** Suggest a global schema that would allow us to maintain as much information as we could about the products sold by companies A and B of Exercise 21.2.1.

Exercise 21.2.3: Write SQL queries to gather the information from the data at companies A and B and put it in a warehouse with your global schema of Exercise 21.2.2.

Exercise 21.2.4: Suppose your global schema from Exercise 21.2.2 is used at a mediator. How would the mediator process the query that asks for the maximum amount of hard-disk available with any computer with a 3 gigahertz processor speed?

! **Exercise 21.2.5:** Suggest two other schemas that computer companies might use to hold data like that of Exercise 21.2.1. How would you integrate your schemas into your global schema from Exercise 21.2.2?

Exercise 21.2.6: In Example 21.3 we talked about a relation `Cars` at Dealer 1 that conveniently had an attribute `autoTrans` with only the values `'yes'` and `'no'`. Since these were the same values used for that attribute in the global schema, the construction of relation `AutosWhse` was especially easy. Suppose instead that the attribute `Cars.autoTrans` has values that are integers, with 0 meaning no automatic transmission, and $i > 0$ meaning that the car has an i-speed automatic transmission. Show how the translation from `Cars` to `AutosWhse` could be done by a SQL query.

Exercise 21.2.7: Write the insert-statements for the second dealer in Example 21.3. You may assume the values of `autoTrans` are `'yes'` and `'no'`.

Exercise 21.2.8: How would the mediator of Example 21.4 translate the following queries?

a) Find the serial numbers of cars with automatic transmission.

b) Find the serial numbers of cars without automatic transmission.

! c) Find the serial numbers of the blue cars from Dealer 1.

Exercise 21.2.9: Go to the Web pages of several on-line booksellers, and see what information about this book you can find. How would you combine this information into a global schema suitable for a warehouse or mediator?

21.3 Wrappers in Mediator-Based Systems

In a data warehouse system like Fig. 21.3, the source extractors consist of:

1. One or more predefined queries that are executed at the source to produce data for the warehouse.
2. Suitable communication mechanisms, so the wrapper (extractor) can:
   (a) Pass ad-hoc queries to the source,
   (b) Receive responses from the source, and
   (c) Pass information to the warehouse.

The predefined queries to the source could be SQL queries if the source is a SQL database as in our examples of Section 21.2. Queries could also be operations in whatever language was appropriate for a source that was not a database system; e.g., the wrapper could fill out an on-line form at a Web page, issue a query to an on-line bibliography service in that system's own, specialized language, or use myriad other notations to pose the queries.

However, mediator systems require more complex wrappers than do most warehouse systems. The wrapper must be able to accept a variety of queries from the mediator and translate any of them to the terms of the source. Of

course, the wrapper must then communicate the result to the mediator, just as a wrapper in a warehouse system communicates with the warehouse. In the balance of this section, we study the construction of flexible wrappers that are suitable for use with a mediator.

21.3.1 Templates for Query Patterns

A systematic way to design a wrapper that connects a mediator to a source is to classify the possible queries that the mediator can ask into *templates*, which are queries with parameters that represent constants. The mediator can provide the constants, and the wrapper executes the query with the given constants. An example should illustrate the idea; it uses the notation T => S to express the idea that the template T is turned by the wrapper into the source query S.

Example 21.5: Suppose we want to build a wrapper for the source of Dealer 1, which has the schema

```
Cars(serialNo, model, color, autoTrans, navi,...)
```

for use by a mediator with schema

```
AutosMed(serialNo, model, color, autoTrans, dealer)
```

Consider how the mediator could ask the wrapper for cars of a given color. If we denote the code representing that color by the parameter `$c`, then we can use the template shown in Fig. 21.5.

```
SELECT *
FROM AutosMed
WHERE color = '$c';
    =>
SELECT serialNo, model, color, autoTrans, 'dealer1'
FROM Cars
WHERE color = '$c';
```

Figure 21.5: A wrapper template describing queries for cars of a given color

Similarly, the wrapper could have another template that specified only the parameter `$m` representing a model, yet another template in which it was only specified whether an automatic transmission was wanted, and so on. In this case, there are eight choices, if queries are allowed to specify any of three attributes: `model`, `color`, and `autoTrans`. In general, there would be 2^n templates if we have the option of specifying n attributes.[2] Other templates would

[2]If the source is a database that can be queried in SQL, as in our example, you would rightly expect that one template could handle any number of attributes equated to constants,

be needed to deal with queries that asked for the total number of cars of certain types, or whether there exists a car of a certain type. The number of templates could grow unreasonably large, but some simplifications are possible by adding more sophistication to the wrapper, as we shall discuss starting in Section 21.3.3. □

21.3.2 Wrapper Generators

The templates defining a wrapper must be turned into code for the wrapper itself. The software that creates the wrapper is called a *wrapper generator*; it is similar in spirit to the parser generators (e.g., YACC) that produce components of a compiler from high-level specifications. The process, suggested in Fig. 21.6, begins when a specification, that is, a collection of templates, is given to the wrapper generator.

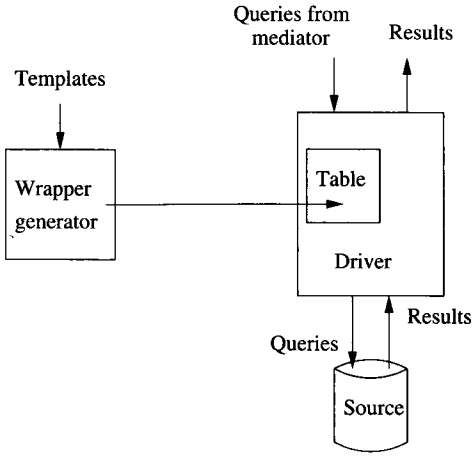

Figure 21.6: A wrapper generator produces tables for a driver; the driver and tables constitute the wrapper

The wrapper generator creates a table that holds the various query patterns contained in the templates, and the source queries that are associated with each. A *driver* is used in each wrapper; in general the driver can be the same for each generated wrapper. The task of the driver is to:

1. Accept a query from the mediator. The communication mechanism may be mediator-specific and is given to the driver as a "plug-in," so the same

simply by making the `WHERE` clause a parameter. While that approach will work for SQL sources and queries that only bind attributes to constants, we could not necessarily use the same idea with an arbitrary source, such as a Web site that allowed only certain forms as an interface. In the general case, we cannot assume that the way we translate one query resembles at all the way similar queries are translated.

driver can be used in systems that communicate differently.

2. Search the table for a template that matches the query. If one is found, then the parameter values from the query are used to instantiate a source query. If there is no matching template, the wrapper responds negatively to the mediator.

3. The source query is sent to the source, again using a "plug-in" communication mechanism. The response is collected by the wrapper.

4. The response is processed by the wrapper, if necessary, and then returned to the mediator. The next sections discuss how wrappers can support a larger class of queries by processing results.

21.3.3 Filters

Suppose that a wrapper on a car dealer's database has the template shown in Fig. 21.5 for finding cars by color. However, the mediator is asked to find cars of a particular model *and* color. Perhaps the wrapper has been designed with a more complex template such as that of Fig. 21.7, which handles queries that specify both model and color. Yet, as we discussed at the end of Example 21.5, it is not always realistic to write a template for every possible form of query.

```
SELECT *
FROM AutosMed
WHERE model = '$m' AND color = '$c';
    =>
SELECT serialNo, model, color, autoTrans, 'dealer1'
FROM Cars
WHERE model = '$m' AND color = '$c';
```

Figure 21.7: A wrapper template that gets cars of a given model and color

Another approach to supporting more queries is to have the wrapper *filter* the results of queries that it poses to the source. As long as the wrapper has a template that (after proper substitution for the parameters) returns a superset of what the query wants, then it is possible to filter the returned tuples at the wrapper and pass only the desired tuples to the mediator.

Example 21.6: Suppose the only template we have is the one in Fig. 21.5 that finds cars given a color. However, the wrapper is asked by the mediator to find blue Gobi model cars. A possible way to answer the query is to use the template of Fig. 21.5 with `$c = 'blue'` to find all the blue cars and store them in a temporary relation

```
TempAutos(serialNo, model, color, autoTrans, dealer)
```

Position of the Filter Component

We have, in our examples, supposed that the filtering operations take place at the wrapper. It is also possible that the wrapper passes raw data to the mediator, and the mediator filters the data. However, if most of the data returned by the template does not match the mediator's query, then it is best to filter at the wrapper and avoid the cost of shipping unneeded tuples.

The wrapper may then return to the mediator the desired set of automobiles by executing the local query:

```
SELECT *
FROM TempAutos
WHERE model = 'Gobi';
```

In practice, the tuples of `TempAutos` could be produced one-at-a-time and filtered one-at-a-time, in a pipelined fashion, rather than having the entire relation `TempAutos` materialized at the wrapper and then filtered. □

21.3.4 Other Operations at the Wrapper

It is possible to transform data in other ways at the wrapper, as long as we are sure that the source-query part of the template returns to the wrapper all the data needed in the transformation. For instance, columns may be projected out of the tuples before transmission to the mediator. It is even possible to take aggregations or joins at the wrapper and transmit the result to the mediator.

Example 21.7: Suppose the mediator wants to know about blue Gobis at the various dealers, but only asks for the serial number, dealer, and whether or not there is an automatic transmission, since the value of the `model` and `color` attributes are obvious from the query. The wrapper could proceed as in Example 21.6, but at the last step, when the result is to be returned to the mediator, the wrapper performs a projection in the `SELECT` clause as well as the filtering for the Gobi model in the `WHERE clause`. The query

```
SELECT serialNo, autoTrans, dealer
FROM TempAutos
WHERE model = 'Gobi';
```

does this additional filtering, although as in Example 21.6 relation `TempAutos` would probably be pipelined into the projection operator, rather than materialized at the wrapper. □

Example 21.8: For a more complex example, suppose the mediator is asked to find dealers and models such that the dealer has two red cars, of the same model, one with and one without an automatic transmission. Suppose also that the only useful template for Dealer 1 is the one about colors from Fig. 21.5. That is, the mediator asks the wrapper for the answer to the query of Fig. 21.8. Note that we do not have to specify a dealer for either `A1` or `A2`, because this wrapper can only access data belonging to Dealer 1. The wrappers for all the other dealers will be asked the same query by the mediator.

```
SELECT A1.model A1.dealer
FROM AutosMed A1, AutosMed A2
WHERE A1.model = A2.model AND
      A1.color = 'red' AND
      A2.color = 'red' AND
      A1.autoTrans = 'no' AND
      A2.autoTrans = 'yes';
```

Figure 21.8: Query from mediator to wrapper

A cleverly designed wrapper could discover that it is possible to answer the mediator's query by first obtaining from the Dealer-1 source a relation with all the red cars at that dealer:

```
RedAutos(serialNo, model, color, autoTrans, dealer)
```

To get this relation, the wrapper uses its template from Fig. 21.5, which handles queries that specify a color only. In effect, the wrapper acts as if it were given the query:

```
SELECT *
FROM AutosMed
WHERE color = 'red';
```

The wrapper can then create the relation `RedAutos` from Dealer 1's database by using the template of Fig. 21.5 with `$c = 'red'`. Next, the wrapper joins `RedAutos` with itself, and performs the necessary selection, to get the relation asked for by the query of Fig. 21.8. The work performed by the wrapper for this step is shown in Fig. 21.9. □

21.3.5 Exercises for Section 21.3

Exercise 21.3.1: In Fig. 21.5 we saw a simple wrapper template that translated queries from the mediator for cars of a given color into queries at the dealer with relation `Cars`. Suppose that the color codes used by the mediator in its schema were different from the color codes used at this dealer, and there was

```
SELECT DISTINCT A1.model, A1.dealer
FROM RedAutos A1, RedAutos A2
WHERE A1.model = A2.model AND
      A1.autoTrans = 'no' AND
      A2.autoTrans = 'yes';
```

Figure 21.9: Query performed at the wrapper (or mediator) to complete the answer to the query of Fig. 21.8

a relation `GtoL(globalColor, localColor)` that translated between the two sets of codes. Rewrite the template so the correct query would be generated.

Exercise 21.3.2: In Exercise 21.2.1 we spoke of two computer companies, A and B, that used different schemas for information about their products. Suppose we have a mediator with schema

```
PCMed(manf, speed, mem, disk, screen)
```

with the intuitive meaning that a tuple gives the manufacturer (A or B), processor speed, main-memory size, hard-disk size, and screen size for one of the systems you could buy from that company. Write wrapper templates for the following types of queries. Note that you need to write two templates for each query, one for each of the manufacturers.

a) Given a speed, find the tuples with that speed.

b) Given a screen size, find the tuples with that size.

c) Given memory and disk sizes, find the matching tuples.

Exercise 21.3.3: Suppose you had the wrapper templates described in Exercise 21.3.2 available in the wrappers at each of the two sources (computer manufacturers). How could the mediator use these capabilities of the wrappers to answer the following queries?

a) Find the manufacturer, memory size, and screen size of all systems with a 3.1 gigahertz speed and a 120 gigabyte disk.

! b) Find the maximum amount of hard disk available on a system with a 2.8 gigahertz processor.

c) Find all the systems with 512M memory and a screen size (in inches) that exceeds the disk size (in gigabytes).

21.4 Capability-Based Optimization

In Section 16.5 we introduced the idea of cost-based query optimization. A typical DBMS estimates the cost of each query plan and picks what it believes to be the best. When a mediator is given a query to answer, it often has little knowledge of how long its sources will take to answer the queries it sends them. Furthermore, many sources are not SQL databases, and often they will answer only a small subset of the kinds of queries that the mediator might like to pose. As a result, optimization of mediator queries cannot rely on cost measures alone to select a query plan.

Optimization by a mediator usually follows the simpler strategy known as *capability-based optimization*. The central issue is not what a query plan costs, but whether the plan can be executed at all. Only among plans found to be executable ("feasible") do we try to estimate costs.

21.4.1 The Problem of Limited Source Capabilities

Today, many useful sources have only Web-based interfaces, even if they are, behind the scenes, an ordinary database. Web sources usually permit querying only through a query form, which does not accept arbitrary SQL queries. Rather, we are invited to enter values for certain attributes and can receive a response that gives values for other attributes.

Example 21.9: The Amazon.com interface allows us to query about books in many different ways. We can specify an author and get all their books, or we can specify a book title and receive information about that book. We can specify keywords and get books that match the keywords. However, there is also information we can receive in answers but cannot specify. For instance, Amazon ranks books by sales, but we cannot ask "give me the top 10 sellers." Moreover, we cannot ask questions that are too general. For instance, the query:

```
SELECT * FROM Books;
```

"tell me everything you know about books," cannot be asked or answered through the Amazon Web interface, although it could be answered behind the scenes if we were able to access the Amazon database directly. □

There are a number of other reasons why a source may limit the ways in which queries can be asked. Among them are:

1. Many of the earliest data sources did not use a DBMS, surely not a relational DBMS that supports SQL queries. These systems were designed to be queried in certain very specific ways only.

2. For reasons of security, a source may limit the kinds of queries that it will accept. Amazon's unwillingness to answer the query "tell me about

all your books" is a rudimentary example; it protects against a rival exploiting the Amazon database. As another instance, a medical database may answer queries about averages, but won't disclose the details of a particular patient's medical history.

3. Indexes on large databases may make certain kinds of queries feasible, while others are too expensive to execute. For instance, if a books database were relational, and one of the attributes were `author`, then without an index on that attribute, it would be infeasible to answer queries that specified only an author.[3]

21.4.2 A Notation for Describing Source Capabilities

If data is relational, or may be thought of as relational, then we can describe the legal forms of queries by *adornments*. These are sequences of codes that represent the requirements for the attributes of the relation, in their standard order. The codes we shall use for adornments reflect the most common capabilities of sources. They are:

1. f (free) means that the attribute can be specified or not, as we choose.
2. b (bound) means that we must specify a value for the attribute, but any value is allowed.
3. u (unspecified) means that we are not permitted to specify a value for the attribute.
4. $c[S]$ (choice from set S) means that a value must be specified, and that value must be one of the values in the finite set S. This option corresponds, for instance, to values that are specified from a pulldown menu in a Web interface.
5. $o[S]$ (optional, from set S) means that we either do not specify a value, or we specify one of the values in the finite set S.

In addition, we place a prime (e.g., f') on a code to indicate that the attribute is not part of the output of the query.

A *capabilities specification* for a source is a set of adornments. The intent is that in order to query the source successfully, the query must match one of the adornments in its capabilities specification. Note that, if an adornment has free or optional components, then queries with different sets of attributes specified may match that adornment.

[3]We should be aware, however, that information like Amazon's about products is not accessed as if it were a relational database. Rather, the information about books is stored as text, with an inverted index, as we discussed in Section 14.1.8. Thus, queries about any aspect of books — authors, titles, words in titles, and perhaps words in descriptions of the book — are supported by this index.

Example 21.10: Suppose we have two sources like those of the two dealers in Example 21.4. Dealer 1 is a source of data in the form:

```
Cars(serialNo, model, color, autoTrans, navi)
```

Note that in the original, we suggested relation `Cars` could have additional attributes representing options, but for simplicity in this example, let us limit our thinking to automatic transmissions and navigation systems only. Here are two possible ways that Dealer 1 might allow this data to be queried:

1. The user specifies a serial number. All the information about the car with that serial number (i.e., the other four attributes) is produced as output. The adornment for this query form is $b'uuuu$. That is, the first attribute, `serialNo` must be specified and is not part of the output. The other attributes must *not* be specified and *are* part of the output.

2. The user specifies a model and color, and perhaps whether or not automatic transmission and navigation system are wanted. All five attributes are printed for all matching cars. An appropriate adornment is

   $$ubbo[\text{yes, no}]o[\text{yes, no}]$$

   This adornment says we must not specify the serial number; we must specify a model and color, but are allowed to give any possible value in these fields. Also, we may, if we wish, specify whether we want automatic transmission and/or a navigation system, but must do so by using only the values "yes" and "no" in those fields.

□

21.4.3 Capability-Based Query-Plan Selection

Given a query at the mediator, a capability-based query optimizer first considers what queries it can ask at the sources to help answer the query. If we imagine those queries asked and answered, then we have bindings for some more attributes, and these bindings may make some more queries at the sources possible. We repeat this process until either:

1. We have asked enough queries at the sources to resolve all the conditions of the mediator query, and therefore we may answer that query. Such a plan is called *feasible.*

2. We can construct no more valid forms of source queries, yet we still cannot answer the mediator query, in which case the mediator must give up; it has been given an impossible query.

What Do Adornments Guarantee?

It would be wonderful if a source that supported queries matching a given adornment would return all possible answers to the query. However, sources normally have only a subset of the possible answers to a query. For instance, Amazon does not stock every book that has ever been written, and the two dealers of our running automobiles example each have distinct sets of cars in their database. Thus, a more proper interpretation of an adornment is: "I will answer a query in the form described by this adornment, and every answer I give will be a true answer, but I do not guarantee to provide all true answers." An important consequence of this state of affairs is that if we want all available tuples for a relation R, then we must query every source that might contribute such tuples.

The simplest form of mediator query for which we need to apply the above strategy is a join of relations, each of which is available, with certain adornments, at one or more sources. If so, then the search strategy is to try to get tuples for each relation in the join, by providing enough argument bindings that some source allows a query about that relation to be asked and answered. A simple example will illustrate the point.

Example 21.11: Let us suppose we have sources like the relations of Dealer 2 in Example 21.4:

```
Autos(serial, model, color)
Options(serial, option)
```

Suppose that *ubf* is the sole adornment for `Autos`, while `Options` has two adornments, *bu* and *uc*[autoTrans, navi], representing two different kinds of queries that we can ask at that source. Let the query be "find the serial numbers and colors of Gobi models with a navigation system."

Here are three different query plans that the mediator must consider:

1. Specifying that the model is Gobi, query `Autos` and get the serial numbers and colors of all Gobis. Then, using the *bu* adornment for `Options`, for each such serial number, find the options for that car and filter to make sure it has a navigation system.

2. Specifying the navigation-system option, query `Options` using the

   uc[autoTrans, navi]

   adornment and get all the serial numbers for cars with a navigation system. Then query `Autos` as in (1), to get all the serial numbers and colors of Gobis, and intersect the two sets of serial numbers.

3. Query `Options` as in (2) to get the serial numbers for cars with a navigation system. Then use these serial numbers to query `Autos` and see which of these cars are Gobis.

Either of the first two plans are acceptable. However, the third plan is one of several plans that will not work; the system does not have the capability to execute this plan because the second part — the query to `Autos` — does not have a matching adornment. □

21.4.4 Adding Cost-Based Optimization

The mediator's query optimizer is not done when the capabilities of the sources are examined. Having found the feasible plans, it must choose among them. Making an intelligent, cost-based optimization requires that the mediator know a great deal about the costs of the queries involved. Since the sources are usually independent of the mediator, it is difficult to estimate the cost. For instance, a source may take less time during periods when it is lightly loaded, but when are those periods? Long-term observation by the mediator is necessary for the mediator even to guess what the response time might be.

In Example 21.11, we might simply count the number of queries to sources that must be issued. Plan (2) uses only two source queries, while plan (1) uses one plus the number of Gobis found in the `Autos` relation. Thus, it appears that plan (2) has lower cost. On the other hand, if the queries of `Options`, one with each serial number, could be combined into one query, then plan (1) might turn out to be the superior choice.

21.4.5 Exercises for Section 21.4

Exercise 21.4.1: Suppose each relation from Exercise 21.2.1:

```
Computers(number, proc, speed, memory, hd)
Monitors(number, screen, maxResX, maxResY)
```

is an information source. Using the notation from Section 21.4.2, write one or more adornments that express the following capabilities:

a) We can query for computers having a given processor, which must be one of "P-IV," "G5," or "Athlon," a given speed, and (optionally) a given amount of memory.

b) We can query for computers having any specified hard-disk size and/or any given memory size.

c) We can query for monitors if we specify either the number of the monitor, the screen size, or the maximum resolution in both dimensions.

d) We can query for monitors if we specify the screen size, which must be either 19, 22, 24, or 30 inches. All attributes except the screen size are returned.

! e) We can query for computers if we specify any two of the processor type, processor speed, memory size, or disk size.

Exercise 21.4.2: Suppose we have the two sources of Exercise 21.4.1, but understand the attribute `number` of both relations to refer to the number of a complete system, some of whose attributes are found in one source and some in the other. Suppose also that the adornments describing access to the `Computers` relation are *buuuu*, *ubbff*, and *uuubb*, while the adornments for `Monitors` are *bfff* and *ubbb*. Tell what plans are feasible for the following queries (exclude any plans that are obviously more expensive than other plans on your list):

a) Find the systems with 512 megabytes of memory, an 80-gigabyte hard disk, and a 22-inch monitor.

b) Find the systems with a Pentium-IV processor running at 3.0 gigahertz with a 22-inch monitor and a maximum resolution of 1600-by-1050.

! c) Find all systems with a G5 processor running at 1.8 gigahertz, with 2 gigabytes of memory, a 300 gigabyte disk, and a 19-inch monitor.

21.5 Optimizing Mediator Queries

In this section, we shall give a greedy algorithm for answering queries at a mediator. This algorithm, called *chain*, always finds a way to answer the query by sending a sequence of requests to its sources, provided at least one solution exists. The class of queries that can be handled is those that involve joins of relations that come from the sources, followed by an optional selection and optional projection onto output attributes. This class of queries is exactly what can be expressed as Datalog rules (Section 5.3).

21.5.1 Simplified Adornment Notation

The Chain Algorithm concerns itself with Datalog rules and with whether prior source requests have provided bindings for any of the variables in the body of the rule. Since we care only about whether we have found all possible constants for a variable, we can limit ourselves, in the query at the mediator (although not at the sources), to the b (bound) and f (free) adornments. That is, a $c[S]$ adornment for an attribute of a source relation can be used as soon as we know all possible values of interest for that attribute (i.e., the corresponding position in the mediator query has a b adornment). Note that the source will not provide matches for the values outside S, so there is no point in asking questions about these values. The optional adornment $o[S]$ can be treated as free, since there is

no need to have a binding for the corresponding attribute in the query at the mediator (although we could). Likewise, adornment u can be treated as free, since although we cannot then specify a value for the attribute at the source, we can have, or not have, a binding for the corresponding variable at the mediator.

Example 21.12: Let us use the same query and source relations as in Example 21.11, but with different capabilities at the sources. In what follows we shall use superscripts on the predicate or relation names to show the adornment or permitted set of adornments. In this example, the permitted adornments for the two source relations are:

$$\texttt{Autos}^{buu}\texttt{(serial, model, color)}$$
$$\texttt{Options}^{uc[\texttt{autoTrans}, \texttt{navi}]}\texttt{(serial, option)}$$

That is, we can only access `Options` by providing a binding "autoTrans" or "navi" for the `option` attribute, and we can only access `Autos` by providing a binding for the `serial` attribute.

The query "find the serial numbers and colors of Gobi models with a navigation system" is expressed in Datalog by:

$$\texttt{Answer(s,c)} \leftarrow \texttt{Autos}^{fbf}\texttt{(s,"Gobi",c) AND Options}^{fb}\texttt{(s,"navi")}$$

Here, notice the adornments on the subgoals of the body. These, at the moment, are commentaries on what arguments of each subgoal are bound to a set of constants. Initially, only the middle argument of the `Autos` subgoal is bound (to the set containing only the constant "Gobi") and the second argument of the `Options` subgoal is bound to the set containing only "navi." We shall see shortly that as we use the sources to find tuples that match one or another subgoal, we get bindings for some of the variables in the Datalog rule, and thus change some of the f's to b's in the adornments. □

21.5.2 Obtaining Answers for Subgoals

We now need to formalize the comments made at the beginning of Section 21.5.1 about when a subgoal with some of its arguments bound can be answered by a source query. Suppose we have a subgoal $R^{x_1x_2\cdots x_n}(a_1, a_2, \ldots, a_n)$, where each x_i is either b or f. R is a relation that can be queried at some source, and which has some set of adornments.

Suppose $y_1y_2\cdots y_n$ is one of the adornments for R at its source. Each y_i can be any of b, f, u, $c[S]$ or $o[S]$ for any set S. Then it is possible to obtain a relation for the subgoal provided, for each $i = 1, 2, \ldots, n$, provided:

- If y_i is b or of the form $c[S]$, then $x_i = b$.
- If $x_i = f$, then y_i is not output restricted (i.e., not primed).

Note that if y_i is any of f, u, or $o[S]$, then x_i can be either b or f. We say that the adornment on the subgoal *matches* the adornment at the source.

Example 21.13: Suppose the subgoal in question is $R^{bbff}(p, q, r, s)$, and the adornments for R at its source are $\alpha_1 = fc[S_1]uo[S_2]$ and $\alpha_2 = c[S_3]bfc[S_4]$. Then *bbff* matches adornment α_1, so we may use α_1 to get the relation for subgoal $R(p, q, r, s)$. That is, α_1 has no b's and only one c, in the second position. Since the adornment of the subgoal has b in the second position, we know that there is a set of constants to which the variable q (the variable in the second argument of the subgoal) has been bound. For each of those constants that are a member of the set S_1 we can issue a query to the source for R, using that constant as the binding for the second argument. We do not provide bindings for any other argument, even though α_1 allows us to provide a binding for the first and/or fourth argument as well.

However, *bbff* does not match α_2. The reason is that α_2 has $c[S_4]$ in the fourth position, while *bbff* has f in that position. If we were to try to obtain R using α_2, we would have to provide a binding for the fourth argument, which means that variable s in $R(p, q, r, s)$ would have to be bound to a set of constants. But we know that is not the case, or else the adornment on the subgoal would have had b in the fourth position. □

21.5.3 The Chain Algorithm

The *Chain Algorithm* is a greedy approach to selecting an order in which we obtain relations for each of the subgoals of a Datalog rule. It is not guaranteed to provide the most efficient solution, but it will provide a solution whenever one exists, and in practice, it is very likely to obtain the most efficient solution. The algorithm maintains two kinds of information:

- An adornment is maintained for each subgoal. Initially, the adornment for a subgoal has b if and only if the mediator query provides a constant binding for the corresponding argument of that subgoal, as for instance, the query in Example 21.12 provided bindings for the second arguments of both the `Autos` and `Options` subgoals. In all other places, the adornment has f's.
- A relation X that is (a projection of) the join of the relations for all the subgoals that have been *resolved*. We resolve a subgoal when the adornment for the subgoal matches one of the adornments at the source for this subgoal, and we have extracted from the source all possible tuples for that subgoal. Initially, since no subgoals have been resolved, X is a relation over no attributes, containing just the empty tuple (i.e., the tuple with zero components). Note that for empty X and any relation R, $X \bowtie R = R$; i.e., X is initially the identity relation for the natural-join operation. As the algorithm progresses, X will have attributes that are variables of the rule — those variables that correspond to b's in the adornments of the subgoals in which they appear.

The core of the Chain Algorithm is as follows. After initializing relation X and the adornments of the subgoals as above, we repeatedly select a subgoal

that can be resolved. Let $R^{\alpha}(a_1, a_2, \ldots, a_n)$ be the subgoal to be resolved. We do so by:

1. Wherever α has a b, we shall find that either the corresponding argument of R is a constant rather than a variable, or it is one of the variables in the schema of the relation X. Project X onto those of its variables that appear in subgoal R. Each tuple in the projection, together with constants in the subgoal R, if any, provide sufficient bindings to use one of the adornments for the source relation R — whichever adornment α matches.

2. Issue a query to the source for each tuple t in the projection of X. We construct the query as follows, depending on the source adornment β that α matches.

   (a) If a component of β is b, then the corresponding component of α is too, and we can use the corresponding component of t (or a constant in the subgoal) to provide the necessary binding for the source query.

   (b) If a component of β is $c[S]$, then again the corresponding component of α will be b, and we can obtain a constant from the subgoal or the tuple t. However, if that constant is not in S, then there is no chance the source can produce any tuples that match t, so we do not generate any source query for t.

   (c) If a component of β is f, then produce a constant value for this component in the source query if we can; otherwise do not provide a value for this component in the source query. Note that we can provide a constant exactly when the corresponding component of α is b.

   (d) If a component of β is u, provide no binding for this component, even if the corresponding component of α is b.

   (e) If a component of β is $o[S]$, treat this component as if it were f in the case that the corresponding component of α is f, and as $c[S]$ if the corresponding component of α is b.

   For each tuple returned, extend the tuple so it has one component for each argument of the subgoal (i.e., n components). Note that the source will return every component of R that is not output restricted, so the only components that are not present have b in the adornment α. Thus, the returned tuples can be padded by using either the constant from the subgoal, or the constant from the tuple in the projection of X. The union of all the responses is the relation R for the subgoal $R(a_1, a_2, \ldots, a_n)$.

3. Every variable among $a_1, a_2, \ldots, a_n$ is now bound. For each subgoal that has not yet been resolved, change its adornment so any position holding one of these variables is now bound (b).

4. Replace X by $X \bowtie \pi_S(R)$, where S is all the variables among

$$a_1, a_2, \ldots, a_n$$

5. Project out of X all components that correspond to variables that do not appear in the head or in any unresolved subgoal. These components can never be useful in what follows.

The complete Chain Algorithm, then, consists of the initialization described above, followed by as many subgoal-resolution steps as we can manage. If we succeed in resolving every subgoal, then relation X will be the answer to the query. If at some point, there are unresolved subgoals, yet none can be resolved, then the algorithm fails. In that case, there can be no other sequence of resolution steps that answers the query.

Example 21.14: Consider the mediator query

$$Q\text{:}\quad \texttt{Answer(c)} \leftarrow \texttt{R}^{bf}\texttt{(1,a) AND S}^{ff}\texttt{(a,b) AND T}^{ff}\texttt{(b,c)}$$

There are three sources that provide answers to queries about R, S, and T, respectively. The contents of these relations at the sources and the only adornments supported by these sources are shown in Fig. 21.10.

Relation	R	S	T
Data	w | x 1 | 2 1 | 3 1 | 4	x | y 2 | 4 3 | 5	y | z 4 | 6 5 | 7 5 | 8
Adornment	bf	$c'[2,3,5]f$	bu

Figure 21.10: Data for Example 21.14

Initially, the adornments on the subgoals are as shown in the query Q, and the relation X that we construct initially contains only the empty tuple. Since subgoals S and T have ff adornments, but the adornments at the corresponding sources each have a component with b or c, neither of these subgoals can be resolved. Fortunately, the first subgoal, $R(1, a)$, can be resolved, since the bf adornment at the corresponding source is matched by the adornment of the subgoal. Thus, we send the source for $R(w, x)$ a query with $w = 1$, and the response is the set of three tuples shown in the first column of Fig. 21.10.

We next project the subgoal's relation onto its second component, since only the second component of $R(1, a)$ is a variable. That gives us the relation

a
2
3
4

This relation is joined with X, which currently has no attributes and only the empty tuple. The result is that X becomes the relation above. Since a is now bound, we change the adornment on the S subgoal from ff to bf.

At this point, the second subgoal, $S^{bf}(a, b)$, can be resolved. We obtain bindings for the first component by projecting X onto a; the result is X itself. That is, we can go to the source for $S(x, y)$ with bindings 2, 3, and 4 for x. We do not need bindings for y, since the second component of the adornment for the source is f. The $c'[2, 3, 5]$ code for x says that we can give the source the value 2, 3, or 5 for the first argument. Since there is a prime on the c, we know that only the corresponding y value(s) will be returned, not the value of x that we supplied in the request. We care about values 2, 3, and 4, but 4 is not a possible value at the source for S, so we never ask about it.

When we ask about $x = 2$, we get one response: $y = 4$. We pad this response with the value 2 we supplied to conclude that $(2, 4)$ is a tuple in the relation for the S subgoal. Similarly, when we ask about $x = 3$, we get $y = 5$ as the only response and we add $(3, 5)$ to the set of tuples constructed for the S subgoal. There are no more requests to ask at the source for S, so we conclude that the relation for the S subgoal is

a	b
2	4
3	5

When we join this relation with the previous value of X, the result is just the relation above. However, variable a now appears neither in the head nor in any unresolved subgoal. Thus, we project it out, so X becomes

b
4
5

Since b is now bound, we change the adornment on the T subgoal, so it becomes $T^{bf}(b, c)$. Now this last subgoal can be resolved, which we do by sending requests to the source for $T(y, z)$ with $y = 4$ and $y = 5$. The responses we get back give us the following relation for the T subgoal:

b	c
4	6
5	7
5	8

We join it with the relation for X above, and then project onto the c attribute to get the relation for the head. That is, the answer to the query at the mediator is $\{(6), (7), (8)\}$. □

21.5.4 Incorporating Union Views at the Mediator

In our description of the Chain Algorithm, we assumed that each predicate in the Datalog query at the mediator was a "view" of data at one particular source. However, it is common for there to be several sources that can contribute tuples to the relation for the predicate. How we construct the relation for such a predicate depends on how we expect the sources for the predicate to interact.

The easy case is where we expect the sources for a predicate to contain replicated information. In that case, we can turn to any one of the sources to get the relation for a predicate. This case thus looks exactly like the case where there is a single source for a predicate, but there may be several adornments that allows us to query that source.

The more complex case is when the sources each contribute some tuples to the predicate that the other sources may not contribute. In that case, we should consult all the sources for the predicate. However, there is still a policy choice to be made. Either we can refuse to answer the query unless we can consult all the sources, or we can make best efforts to return all the answers to the query that we can obtain by combinations of sources.

Consult All Sources

If we must consult all sources to consider a subgoal resolved, then we can only resolve a subgoal when each source for its relation has an adornment matched by the current adornment of the subgoal. This rule is a small modification of the Chain Algorithm. However, not only does it make queries harder to answer, it makes queries impossible to answer when any source is "down," even if the Chain Algorithm provides a feasible ordering in which to resolve the subgoals. Thus, as the number of sources grows, this policy becomes progressively less practical.

Best Efforts

Under this assumption, we only need one source with a matching adornment to resolve a subgoal. However, we need to modify the chain algorithm to revisit each subgoal when that subgoal has new bound arguments. We may find that some source that could not be matched is now matched by the subgoal with its new adornment.

Example 21.15: Consider the mediator query

$$\texttt{answer(a,c)} \leftarrow \texttt{R}^{ff}\texttt{(a,b) AND S}^{ff}\texttt{(b,c)}$$

Suppose also that R has two sources, one described by adornment ff and the other by fb. Likewise, S has two sources, described by ff and bf. We could start by using either source with adornment ff; suppose we start with R's source. We query this source and get some tuples for R.

Now, we have some bindings, but perhaps not all, for the variable b. We can now use both sources for S to obtain tuples and the relation for S can be set to their union. At this point, we can project the relation for S onto variable b and get some b-values. These can be used to query the second source for R, the one with adornment fb. In this manner, we can get some additional R-tuples. It is only at this point that we can join the relations for R and S, and project onto a and c to get the best-effort answer to the query. □

21.5.5 Exercises for Section 21.5

Exercise 21.5.1: Apply the Chain Algorithm to the mediator query

```
Answer(a,e) ← R(a,b,c) AND S(c,d) AND T(b,d,e)
```

with the following adornments at the sources for R, S, and T. If there is more than one adornment for a predicate, either may be used.

a) R^{fff}, S^{bf}, T^{bff}, T^{fbf}.

b) R^{ffb}, S^{fb}, T^{fbf}, T^{bff}.

c) R^{fbf}, S^{fb}, S^{bf}, T^{fff}.

In each case:

i. Indicate all possible orders in which the subgoals can be resolved.

ii. Does the Chain Algorithm produce an answer to the query?

iii. Give the sequence of relational-algebra operations needed to compute the intermediate relation X at each step and the result of the query.

! **Exercise 21.5.2:** Suppose that for the mediator query of Exercise 21.5.1, each predicate is a view defined by the union of two sources. For each predicate, one of the sources has an all-f adornment. The other sources have the following adornments: R^{fbb}, S^{bf}, and T^{bff}. Find a best-effort sequence of source requests that will produce all the answers to the mediator query that can be obtained from these sources.

Exercise 21.5.3: Describe all the source adornments that are matched by a subgoal with adornment R^{bf}.

!! **Exercise 21.5.4:** Prove that if there is any sequence of subgoal resolutions that will resolve all subgoals, then the Chain Algorithm will find one. *Hint*: Notice that if a subgoal can be resolved at a certain step, then if it is not selected for resolution, it can still be resolved at the next step.

21.6 Local-as-View Mediators

The mediators discussed so far are called *global-as-view* (GAV) mediators. The global data (i.e., the data available for querying at the mediator) is like a view; it doesn't exist physically, but pieces of it are constructed by the mediator, as needed, by asking queries of the sources.

In this section, we introduce another approach to connecting sources with a mediator. In a *local-as-view* (LAV) mediator, we define global predicates at the mediator, but we do not define these predicates as views of the source data. Rather, we define, for each source, one or more expressions involving the global predicates that describe the tuples that the source is able to produce. Queries are answered at the mediator by discovering all possible ways to construct the query using the views provided by the sources.

21.6.1 Motivation for LAV Mediators

In many applications, GAV mediators are easy to construct. You decide on the global predicates or relations that the mediator will support, and for each source, you consider which predicates it can support, and how it can be queried. That is, you determine the set of adornments for each predicate at each source. For instance, in our Aardvark Automobiles example, if we decide we want `Autos` and `Options` predicates at the mediator, we find a way to query each dealer's source for those concepts and let the `Autos` and `Options` predicates at the mediator represent the union of what the sources provide. Whenever we need one or both of those predicates to answer a mediator query, we make requests of each of the sources to obtain their data.

However, there are situations where the relationship between what we want to provide to users of the mediator and what the sources provide is more subtle. We shall look at an example where the mediator is intended to provide a single predicate $Par(c, p)$, meaning that p is a parent of c. As with all mediators, this predicate represents an abstract concept — in this case, the set of all child-parent facts that could ever exist — and the sources will provide information about whatever child-parent facts they know. Even put together, the sources probably do not know about everyone in the world, let along everyone who ever lived.

Life would be simple if each source held some child-parent information and nothing else that was relevant to the mediator. Then, all we would have to do is determine how to query each one for whatever facts they could provide. However, suppose we have a database maintained by the Association of Grandparents that doesn't provide any child-parent facts at all, but provides child-grandparent facts. We can never use this source to help answer a query about someone's parents or children, but we can use it to help answer a mediator query that uses the `Par` predicate several times to ask for the grandparents of an individual, or their great-grandparents, or another complex relationship among people.

GAV mediators do not allow us to use a grandparents source at all, if our goal is to produce a Par relation. Producing both a parent and a grandparent predicate at the mediator is possible, but it might be confusing to the user and would require us to figure out how to extract grandparents from all sources, including those that only allow queries for child-parent facts. However, LAV mediators allow us to say that a certain source provides grandparent facts. Moreover, the technology associated with LAV mediators lets us discover how and when to use that source in a given query.

21.6.2 Terminology for LAV Mediation

LAV mediators are always defined using a form of logic that serves as the language for defining views. In our presentation, we shall use Datalog. Both the queries at the mediator and the queries (view definitions) that describe the sources will be single Datalog rules. A query that is a single Datalog rule is often called a *conjunctive query*, and we shall use the term here.

A LAV mediator has a set of *global predicates*, which are used as the subgoals of mediator queries. There are other conjunctive queries that define *views*; i.e., their heads each have a unique view predicate that is the name of a view. Each view definition has a body consisting of global predicates and is associated with a particular source, from which that view can be constructed. We assume that each view can be constructed with an all-free adornment. If capabilities are limited, we can use the chain algorithm to decide whether solutions using the views are feasible.

Suppose we are given a conjunctive query Q whose subgoals are predicates defined at the mediator. We need to find all *solutions* — conjunctive queries whose bodies are composed of view predicates, but that can be "expanded" to produce a conjunctive query involving the global predicates. Moreover, this conjunctive query must produce only tuples that are also produced by Q. We say such expansions are *contained* in Q. An example may help with these tricky concepts, after which we shall define "expansion" formally.

Example 21.16: Suppose there is one global predicate $Par(c, p)$ meaning that p is a parent of c. There is one source that produces some of the possible parent facts; its view is defined by the conjunctive query

$$\mathtt{V_1(c,p)} \leftarrow \mathtt{Par(c,p)}$$

There is another source that produces some grandparent facts; its view is defined by the conjunctive query

$$\mathtt{V_2(c,g)} \leftarrow \mathtt{Par(c,p)\ AND\ Par(p,g)}$$

Our query at the mediator will ask for great-grandparent facts that can be obtained from the sources. That is, the mediator query is

$$\mathtt{Q(w,z)} \leftarrow \mathtt{Par(w,x)\ AND\ Par(x,y)\ AND\ Par(y,z)}$$

How might we answer this query? The source view V_1 contributes to the parent predicate directly, so we can use it three times in the obvious solution

```
Q(w,z) ← V1(w,x) AND V1(x,y) AND V1(y,z)
```

There are, however, other solutions that may produce additional answers, and thus must be part of the logical query plan for answering the query. In particular, we can use the view V_2 to get grandparent facts, some of which may not be inferrable by using two parent facts from V_1. We can use V_1 to make a step of one generation, and then use V_2 to make a step of two generations, as in the solution

```
Q(w,z) ← V1(w,x) AND V2(x,z)
```

Or, we can use V_2 first, followed by V_1, as

```
Q(w,z) ← V2(w,y) AND V1(y,z)
```

It turns out these are the only solutions we need; their union is all the great-grandparent facts that we can produce from the sources V_1 and V_2. There is still a great deal to explain. Why are these solutions guaranteed to produce only answers to the query? How do we tell whether a solution is part of the answer to a query? How do we find all the useful solutions to a query? We shall answer each of these questions in the next sections. □

21.6.3 Expanding Solutions

Given a query Q, a solution S has a body whose subgoals are views, and each view V is defined by a conjunctive query with that view as the head. We can substitute the body of V's conjunctive query for a subgoal in S that uses predicate V, as long as we are careful not to confuse variable names from one body with those of another. Once we substitute rule bodies for the views that are in S, we have a body that consists of global predicates only. The expanded solution can be compared with Q, to see if the results produced by the solution S are guaranteed to be answers to the query Q, in a manner we shall discuss later.

However, first we must be clear about the expansion algorithm. Suppose that there is a solution S that has a subgoal $V(a_1, a_2, \ldots, a_n)$. Here the a_i's can be any variables or constants, and it is possible that two or more of the a_i's are actually the same variable. Let the definition of view V be of the form

$$V(b_1, b_2, \ldots, b_n) \leftarrow B$$

where B represents the entire body. We may assume that the b_i's are distinct variables, since there is no need to have two identical components in a view, nor is there a need for components that are constant. We can replace $V(a_1, a_2, \ldots, a_n)$ in solution S by a version of body B that has all the subgoals of B, but with variables possibly altered. The rules for altering the variables of B are:

1. First, identify the *local variables* of B — those variables that appear in the body, but not in the head. Note that, within a conjunctive query, a local variable can be replaced by any other variable, as long as the replacing variable does not appear elsewhere in the conjunctive query. The idea is the same as substituting different names for local variables in a program.

2. If there are any local variables of B that appear in B or in S, replace each one by a distinct new variable that appears nowhere in the rule for V or in S.

3. In the body B, replace each b_i by a_i, for $i = 1, 2, \ldots, n$.

Example 21.17: Suppose we have the view definition

```
V(a,b,c,d) ← E(a,b,x,y) AND F(x,y,c,d)
```

Suppose further that some solution S has in its body a subgoal $V(x, y, 1, x)$.

The local variables in the definition of V are x and y, since these do not appear in the head. We need to change them both, because they appear in the subgoal for which we are substituting. Suppose e and f are variable names that appear nowhere in S. We can rewrite the body of the rule for V as

```
V(a,b,c,d) ← E(a,b,e,f) AND F(e,f,c,d)
```

Next, we must substitute the arguments of the V subgoal for a, b, c, and d. The correspondence is that a and d become x, b becomes y, and c becomes the constant 1. We therefore substitute for $V(x, y, 1, x)$ the two subgoals $E(x, y, e, f)$ and $F(e, f, 1, x)$. □

The expansion process is essentially the substitution described above for each subgoal of the solution S. There is one extra caution of which we must be aware, however. Since we may be substituting for the local variables of several view definitions, and may in fact need to create several versions of one view definition (if S has several subgoals with the same view predicate), we must make sure that in the substitution for each subgoal of S, we use unique local variables — ones that do not appear in any other substitution or in S itself. Only then can we be sure that when we do the expansion we do not use the same name for two variables that should be distinct.

Example 21.18: Let us resume the discussion we began in Example 21.16, where we had view definitions

```
V1(c,p) ← Par(c,p)
V2(c,g) ← Par(c,p) AND Par(p,g)
```

One of the proposed solutions S is

```
Q(w,z) ← V1(w,x) AND V2(x,z)
```

Let us expand this solution. The first subgoal, with predicate V_1 is easy to expand, because the rule for V_1 has no local variables. We substitute w and x for c and p respectively, so the body of the rule for V_1 becomes $Par(w, x)$. This subgoal will be substituted in S for $V_1(w, x)$.

We must also substitute for the V_2 subgoal. Its rule has local variable p. However, since p does not appear in S, nor has it been used as a local variable in another substitution, we are free to leave p as it is. We therefore have only to substitute x and z for the variables c and g, respectively. The two subgoals in the rule for V_2 become $Par(x, p)$ and $Par(p, z)$. When we substitute these two subgoals for $V_2(x, z)$ in S, we have constructed the complete expansion of S:

```
Q(w,z) ← Par(w,x) AND Par(x,p) AND Par(p,z)
```

Notice that this expansion is practically identical to the query in Example 21.16. The only difference is that the query uses local variable y where the expansion uses p. Since the names of local variables do not affect the result, it appears that the solution S is the answer to the query. However, that is not quite right. The query is looking for all great-grandparent facts, and all the expansion says is that the solution S provides only facts that answer the query. S might not produce all possible answers. For example, the source of V_2 might even be empty, in which case nothing is produced by solution S, even though another solution might produce some answers. □

21.6.4 Containment of Conjunctive Queries

In order for a conjunctive query S to be a solution to the given mediator query Q, the expansion of S, say E, must produce only answers that Q produces, regardless of what relations are represented by the predicates in the bodies of E and Q. If so, we say that $E \subseteq Q$.

There is an algorithm to tell whether $E \subseteq Q$; we shall see this test after introducing the following important concept. A *containment mapping* from Q to E is a function τ from the variables of Q to the variables and constants of E, such that:

1. If x is the ith argument of the head of Q, then $\tau(x)$ is the ith argument of the head of E.

2. Add to τ the rule that $\tau(c) = c$ for any constant c. If $P(x_1, x_2, \ldots, x_n)$ is a subgoal of Q, then $P\big(\tau(x_1), \tau(x_2), \ldots, \tau(x_n)\big)$ is a subgoal of E.

Example 21.19: Consider the following two conjunctive queries:

```
Q1:  H(x,y) ← A(x,z) AND B(z,y)
Q2:  H(a,b) ← A(a,c) AND B(d,b) AND A(a,d)
```

We claim that $Q_2 \subseteq Q_1$. In proof, we offer the following containment mapping: $\tau(x) = a$, $\tau(y) = b$, and $\tau(z) = d$. Notice that when we apply this substitution, the head of Q_1 becomes $H(a, b)$, which is the head of Q_2. The first subgoal of Q_1 becomes $A(a, d)$, which is the third subgoal of Q_2. Likewise, the second subgoal of Q_1 becomes the second subgoal of Q_2. That proves there is a containment mapping from Q_1 to Q_2, and therefore $Q_2 \subseteq Q_1$. Notice that no subgoal of Q_1 maps to the first subgoal of Q_2, but the containment-mapping definition does not require that there be one.

Surprisingly, there is also a containment mapping from Q_2 to Q_1, so the two conjunctive queries are in fact equivalent. That is, not only is one contained in the other, but on any relations A and B, they produce exactly the same set of tuples for the relation H. The containment mapping from Q_2 to Q_1 is $\rho(a) = x$, $\rho(b) = y$, and $\rho(c) = \rho(d) = z$. Under this mapping, the head of Q_2 becomes the head of Q_1, the first and third subgoals of Q_2 become the first subgoal of Q_1, and the second subgoal of Q_2 becomes the second subgoal of Q_1.

While it may appear strange that two such different looking conjunctive queries are equivalent, the following is the intuition. Think of A and B as two different colored edges on a graph. Then Q_1 asks for the pairs of nodes x and y such that there is an A-edge from x to some z and a B-edge from z to y. Q_2 asks for the same thing, using its second and third subgoals respectively, although it calls x, y, and z by the names a, b, and d respectively. In addition, Q_2 seems to have the added condition expressed by the first subgoal that there is an edge from node a to somewhere (node c). But we already know that there is an edge from a to somewhere, namely d. That is, we are always free to use the same node for c as we did for d, because there are no other constraints on c. □

Example 21.20: Here are two queries similar, but not identical, to those of Example 21.19:

P_1: `H(x,y) ← A(x,z) AND A(z,y)`
P_2: `H(a,b) ← A(a,c) AND A(c,d) AND A(d,b)`

Intuitively, if we think of A as representing edges in a graph, then P_1 asks for paths of length 2 and P_2 asks for paths of length 3. We do not expect either to be contained in the other, and indeed the containment-mapping test confirms that fact.

Consider a possible containment mapping τ from P_1 to P_2. Because of the conditions on heads, we know $\tau(x) = a$ and $\tau(y) = b$. To what does z map? Since we already know $\tau(x) = a$, the first subgoal $A(x, z)$ can only map to $A(a, c)$ of P_2. That means $\tau(z)$ must be c. However, since $\tau(y) = b$, the subgoal $A(z, y)$ of P_1 can only become $A(d, b)$ in P_2. That means $\tau(z)$ must be d. But z can only map to one value; it cannot map to both c and d. We conclude that no containment mapping from P_1 to P_2 exists.

A similar argument shows that there is no containment mapping from P_2 to P_1. We leave it as an exercise. □

Complexity of the Containment-Mapping Test

It is NP-complete to decide whether there is a containment mapping from one conjunctive query to another. However, in practice, it is usually quite easy to decide whether a containment mapping exists. Conjunctive queries in practice have few subgoals and few variables. Moreover, for the class of conjunctive queries that have no more than two subgoals with the same predicate — a very common condition — there is a linear-time test for the existence of a containment mapping.

The importance of containment mappings is expressed by the following theorem:

- If Q_1 and Q_2 are conjunctive queries, then $Q_2 \subseteq Q_1$ if and only if there is a containment mapping from Q_1 to Q_2.

Notice that the containment mapping goes in the opposite direction from the containment; that is, the containment mapping is from the conjunctive query that produces the larger set of answers to the one that produces the smaller, contained set.

21.6.5 Why the Containment-Mapping Test Works

We need to argue two points. First, if there is a containment mapping, why must there be a containment of conjunctive queries? Second, if there is containment, why must there be a containment mapping? We shall not give formal proofs, but will sketch the arguments.

First, suppose there is a containment mapping τ from Q_1 to Q_2. Recall from Section 5.3.4 that when we apply Q_2 to a database, we look for substitutions σ for all the variables of Q_2 that make all its relational subgoals be tuples of the corresponding relation of the database. The substitution for the head becomes a tuple t that is returned by Q_2. If we compose τ and then σ, we have a mapping from the variables of Q_1 to tuples of the database that produces the same tuple t for the head of Q_1. Thus, on any given database, everything that Q_2 produces is also produced by Q_1.

Conversely, suppose that $Q_2 \subseteq Q_1$. That is, on any database D, everything that Q_2 produces is also produced by Q_1. Construct a particular database D that has only the subgoals of Q_2. That is, pretend the variables of Q_2 are distinct constants, and for each subgoal $P(a_1, a_2, \ldots, a_n)$, put the tuple $(a_1, a_2, \ldots, a_n)$ in the relation for P. There are no other tuples in the relations of D.

When Q_2 is applied to database D, surely the tuple whose components are the arguments of the head of Q_2 is produced. Since $Q_2 \subseteq Q_1$, it must be that

Q_1 applied to D also produces the head of Q_2. Again, we use the definition in Section 5.3.4 of how a conjunctive query is applied to a database. That definition tells us that there is a substitution of constants of D for the variables of Q_1 that turns each subgoal of Q_1 into a tuple in D and turns the head of Q_1 into the tuple that is the head of Q_2. But remember that the constants of D are the variables of Q_2. Thus, this substitution is actually a containment mapping.

21.6.6 Finding Solutions to a Mediator Query

We have one more issue to resolve. We are given a mediator query Q, and we need to find all solutions S such that the expansion E of S is contained in Q. But there could be an infinite number of S built from the views using any number of subgoals and variables. The following theorem limits our search.

- If a query Q has n subgoals, then any answer produced by any solution is also produced by a solution that has at most n subgoals.

This theorem, often called the LMSS Theorem,[4] gives us a finite, although exponential task to find a sufficient set of solutions. There has been considerable work on making the test much more efficient in typical situations.

Example 21.21: Recall the query

$$Q_1: \quad \texttt{Q(w,z)} \leftarrow \texttt{Par(w,x) AND Par(x,y) AND Par(y,z)}$$

from Example 21.16. This query has three subgoals, so we don't have to look at solutions with more than three subgoals. One of the solutions we proposed was

$$S_1: \quad \texttt{Q(w,z)} \leftarrow \texttt{V}_1\texttt{(w,x) AND V}_2\texttt{(x,z)}$$

This solution has only two subgoals, and its expansion is contained in the query. Thus, it needs to be included among the set of solutions that we evaluate to answer the query.

However, consider the following solution:

$$S_2: \quad \texttt{Q(w,z)} \leftarrow \texttt{V}_1\texttt{(w,x) AND V}_2\texttt{(x,z) AND V}_1\texttt{(t,u) AND V}_2\texttt{(u,v)}$$

It has four subgoals, so we know by the LMSS Theorem that it does not need to be considered. However, it is truly a solution, since its expansion

$$E_2: \quad \texttt{Q(w,z)} \leftarrow \texttt{Par(w,x) AND Par(x,p) AND Par(p,z) AND Par(t,u) AND Par(u,q) AND Par(q,v)}$$

[4]For the authors, A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava.

is contained in the query Q_1. To see why, use the containment mapping that maps w, x, and z to themselves and y to p.

However, E_2 is also contained in the expansion E_1 of the smaller solution S_1. Recall from Example 21.18 that the expansion of S_1 is

$$E_1: \quad \texttt{Q(w,z)} \leftarrow \texttt{Par(w,x) AND Par(x,p) AND Par(p,z)}$$

We can see immediately that $E_2 \subseteq E_1$, using the containment mapping that sends each variable of E_1 to the same variable in E_2. Thus, every answer to Q_1 produced by S_2 is also produced by S_1. Notice, incidentally, that S_2 is really S_1 with the two subgoals of S_1 repeated with different variables. □

In principle, to apply the LMSS Theorem, we must consider a number of possible solutions that is exponential in the query size. We must consider not only the choices of predicates for the subgoals, but which arguments of which subgoals hold the same variable. Note that within a conjunctive query, the names of the variables do not matter, but it matters which sets of arguments have the same variable. Most query processing is worst-case exponential in the query size anyway, as we learned in Chapter 16. Moreover, there are some powerful techniques known for limiting the search for solutions by looking at the structure of the conjunctive queries that define the views. We shall not go into depth here, but one easy but powerful idea is the following.

- If the conjunctive query that defines a view V has in its body a predicate P that does not appear in the body of the mediator query, then we need not consider any solution that uses V.

21.6.7 Why the LMSS Theorem Holds

Suppose we have a query Q with n subgoals, and there is a solution S with more than n subgoals. The expansion E of S must be contained in query Q, which means that there is a containment mapping from Q to the expansion E, as suggested in Fig. 21.11. If there are n subgoals ($n = 2$ in Fig. 21.11) in Q, then the containment mapping turns Q's subgoals into at most n of the subgoals of the expansion E. Moreover, these subgoals of E come from at most n of the subgoals of the solution S.

Suppose we removed from S all subgoals whose expansion was not the target of one of Q's subgoals under the containment mapping. We would have a new conjunctive query S' with at most n subgoals. Now S' must also be a solution to Q, because the same containment mapping that showed $E \subseteq Q$ in Fig. 21.11 also shows that $E' \subseteq Q$, where E' is the expansion of S'.

We must show one more thing: that any answer provided by S is also provided by S'. That is, $S \subseteq S'$. But there is an obvious containment mapping from S' to S: the identity mapping. Thus, there is no need for solution S among the solutions to query Q.

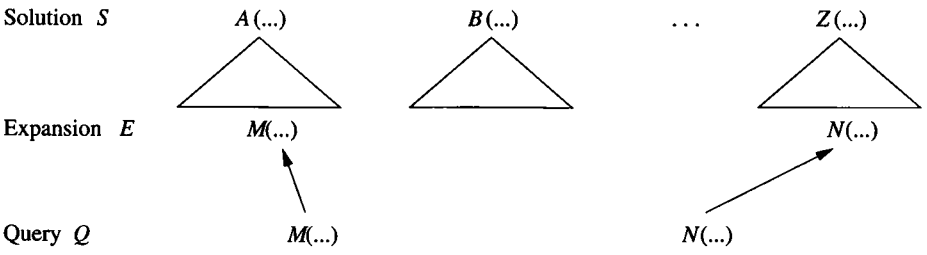

Figure 21.11: Why a query with n subgoals cannot need a solution with more than n subgoals

21.6.8 Exercises for Section 21.6

Exercise 21.6.1: Find all the containments among the following four conjunctive queries:

Q_1: `P(x,y) ← Q(x,a) AND Q(a,b) AND Q(b,y)`
Q_2: `P(x,y) ← Q(x,a) AND Q(a,b) AND Q(b,c) AND Q(c,y)`
Q_3: `P(x,y) ← Q(x,a) AND Q(b,c) AND Q(d,y) AND Q(x,b) AND Q(a,c) AND Q(c,y)`
Q_4: `P(x,y) ← Q(x,a) AND Q(a,1) AND Q(1,b) AND Q(b,y)`

! **Exercise 21.6.2:** For the mediator and views of Example 21.16, find all the needed solutions to the great-great-grandparent query:

```
Q(x,y) ← Par(x,a) AND Par(a,b) AND Par(b,c) AND Par(c,y)
```

! **Exercise 21.6.3:** Show that there is no containment mapping from P_2 to P_1 in Example 21.20.

! **Exercise 21.6.4:** Show that if conjunctive query Q_2 is constructed from conjunctive query Q_1 by removing one or more subgoals of Q_1, then $Q_1 \subseteq Q_2$.

21.7 Entity Resolution

We shall now take up a problem that must be solved in many information-integration scenarios. We have tacitly assumed that sources agree on the representation of entities or values, or at least that it is possible to perform a translation of data as we go through a wrapper. Thus, we are not afraid of two sources that report temperatures, one in Fahrenheit and one in Centigrade. Neither are we afraid of sources that support a concept like "employee" but have somewhat different sets of employees.

What happens, however, if two sources not only have different sets of employees, but it is unclear whether records at the two sources represent the same

individual or not? Discrepancies can occur for many reasons, such as misspellings. In this section, we shall begin by discussing some of the reasons why *entity resolution* — determining whether two records or tuples do or do not represent the same person, organization, place, or other entity — is a hard problem. We then look at the process of comparing records and merging those that we believe represent the same entity. Under some fairly reasonable conditions, there is an algorithm for finding a unique way to group all sets of records that represent a common entity and to perform this grouping efficiently.

21.7.1 Deciding Whether Records Represent a Common Entity

Imagine we have a collection of records that represent members of an entity set. These records may be tuples derived from several different sources, or even from one source. We only need to know that the records each have the same fields (although some records may have null in some fields). We hope to compare the values in corresponding fields to decide whether or not two records represent the same entity.

To be concrete, suppose that the entities are people, and the records have three fields: name, address, and phone. Intuitively, we want to say that two records represent the same individual if the two records have similar values for each of the three fields. It is not sufficient to insist that the values of corresponding fields be identical for a number of reasons. Among them:

1. *Misspellings.* Often, data is entered by a clerk who hears something over the phone, or who copies a written form carelessly. Thus, "Smythe" may appear as "Smith," or "Jones" may appear as "Jomes" ("m" and "n" are adjacent on the keyboard). Two phone numbers or street addresses may differ in a digit, yet really represent the same phone or house.

2. *Variant Names.* A person may supply their middle initial or not. They may use their complete first name or just their initial, or a nickname. Thus, "Susan Williams" may appear as "Susan B. Williams," "S. Williams," or "Sue Williams" in different records.

3. *Misunderstanding of Names.* There are many different systems of names used throughout the world. In the US, it is sometimes not understood that Asian names generally begin with the family name. Thus, "Chen Li" and "Li Chen" may or may not turn out to be the same person. The first author of this book has been referred to as "Hector Garcia-Molina," "Hector Garcia," and even "Hector G. Molina."

4. *Evolution of Values.* Sometimes, two different records that represent the same entity were created at different times. A person may have moved in the interrim, so the address fields in the two records are completely different. Or they may have started using a cell phone, so the phone