CHAPTER 16

Concurrency

16.1 Introduction
16.2 Three Concurrency Problems
16.3 Locking
16.4 The Three Concurrency Problems Revisited
16.5 Deadlock
16.6 Serializability
16.7 Recovery Revisited
16.8 Isolation Levels
16.9 Intent Locking
16.10 Dropping ACID
16.11 SQL Facilities
16.12 Summary
Exercises
References and Bibliography

16.1 INTRODUCTION

The term concurrency refers to the fact that DBMSs typically allow many transactions to access the same database at the same time. In such a system, some kind of control mechanism is clearly needed to ensure that concurrent transactions do not interfere with each other. (Examples of the kinds of interference that can occur in the absence of such controls are given in Section 16.2.) In this chapter, we examine this issue in depth. The structure of the chapter is as follows:

- As just indicated, Section 16.2 explains some of the problems that can arise if proper controls are not provided.

- Section 16.3 introduces the conventional mechanism for dealing with such problems, *locking*. (Locking is not the only possible mechanism, but it is far and away the one most commonly encountered in practice. Several others are described in the annotation to the references at the end of the chapter.)
- Section 16.4 then shows how locking can be used to solve the problems described in Section 16.2.
- Locking unfortunately introduces problems of its own, one of the best known of which is *deadlock*. Section 16.5 discusses this issue.
- Section 16.6 describes the concept of *serializability*, which is generally recognized as the formal criterion of correctness in this area.
- Section 16.7 discusses the effects of concurrency on the topic of the previous chapter, *recovery*.
- Sections 16.8 and 16.9 then go on to consider two significant refinements to the basic locking idea, *levels of isolation* and *intent locking*.
- Section 16.10 offers some fresh and slightly skeptical observations on the question of the so-called *ACID properties* of transactions.
- Section 16.11 describes the relevant facilities of SQL.
- Finally, Section 16.12 presents a summary.

We close this introductory section with a couple of general observations. First, the ideas of concurrency, like those of recovery, are largely independent of whether the underlying system is relational or otherwise (though it is significant that—as with recovery—most of the early theoretical work in the area was done in a relational context specifically, "for definiteness" [16.6]). Second, concurrency, like recovery, is a very large subject, and all we can hope to do in this chapter is introduce some of the most important and basic ideas. Certain more advanced aspects of the subject are discussed briefly in the annotation to some of the references at the end of the chapter.

16.2 THREE CONCURRENCY PROBLEMS

We begin by considering some of the problems that any concurrency control mechanism must address. There are essentially three ways in which things can go wrong: three ways, that is, in which a transaction, though correct in itself in the sense explained in Chapter 15, can nevertheless produce the wrong answer if some other transaction interferes with it in some way. Note carefully that—in accordance with our usual assumption—the interfering transaction will also be correct in itself; it is the uncontrolled interleaving of operations from the two individually correct transactions that produces the overall incorrect result. The three problems are:

- The *lost update* problem
- The *uncommitted dependency* problem

- The *inconsistent analysis* problem

We examine each in turn.

The Lost Update Problem

Consider the situation illustrated in Fig. 16.1. That figure is meant to be read as follows: Transaction *A* retrieves some tuple *t* at time *t1;* transaction *B* retrieves that same tuple *t* at time *t2;* transaction *A* updates the tuple (on the basis of the values seen at time *t1*) at time *t3;* and transaction *B* updates the same tuple (on the basis of the values seen at time *t2*, which are the same as those seen at time *t1*) at time *t4*. Transaction *A*'s update is lost at time *t4*, because transaction *B* overwrites it without even looking at it. *Note:* Once again we adopt the convenient fiction, here and throughout this chapter, that it makes sense to talk in terms of "updating a tuple."

Transaction A	*Time*	*Transaction B*
–	│	–
–	│	–
RETRIEVE t	t1	–
–	│	–
–	t2	RETRIEVE t
–	│	–
UPDATE t	t3	–
–	│	–
–	t4	UPDATE t
–	↓	–

Fig. 16.1 Transaction *A* loses an update at time *t4*

The Uncommitted Dependency Problem

The uncommitted dependency problem arises if one transaction is allowed to retrieve—or, worse, update—a tuple that has been updated by another transaction but not yet committed by that other transaction. For if it has not yet been committed, there is always a possibility that it never will be committed but will be rolled back instead, in which case the first transaction will have seen some data that now no longer exists (and in a sense never did exist). Consider Figs. 16.2 and 16.3 (overleaf).

In the first example (Fig. 16.2), transaction *A* sees an uncommitted update—also called an uncommitted change—at time *t2*. That update is then undone at time *t3*. Transaction *A* is therefore operating on a false assumption: namely, the assumption that tuple *t* has the value seen at time *t2*, whereas in fact it has whatever value it had prior to time *t1*. As a result, transaction *A* might well produce an incorrect result. Note, by the way, that the rollback of transaction *B* might be due to no fault of *B*'s; it might, for example, be the

Transaction A	Time	Transaction B
–		–
–		–
–	t1	UPDATE t
–		–
RETRIEVE t	t2	–
–		–
–	t3	ROLLBACK
–	↓	

Fig. 16.2 Transaction *A* becomes dependent on an uncommitted change at time *t2*

Transaction A	Time	Transaction B
–		–
–		–
–	t1	UPDATE t
–		–
UPDATE t	t2	–
–		–
–	t3	ROLLBACK
–	↓	

Fig. 16.3 Transaction *A* updates an uncommitted change at time *t2*, and loses that update at time *t3*

result of a system crash. (And transaction *A* might already have terminated by that time, in which case the crash would not cause a rollback to be issued for *A* also.)

The second example (Fig. 16.3) is even worse. Not only does transaction *A* become dependent on an uncommitted change at time *t2*, but it actually loses an update at time *t3*—because the rollback at time *t3* causes tuple *t* to be restored to its value prior to time *t1*. This is another version of the lost update problem.

The Inconsistent Analysis Problem

Consider Fig. 16.4, which shows two transactions *A* and *B* operating on account (ACC) tuples: Transaction *A* is summing account balances, transaction *B* is transferring an amount 10 from account 3 to account 1. The result produced by *A*, 110, is obviously incorrect; if *A* were to go on to write that result back into the database, it would actually leave the database in an inconsistent state.[1] In effect, *A* has seen an inconsistent state of the database and has therefore performed an inconsistent analysis. Note the difference between

[1] Regarding this possibility (i.e., writing the result back into the database), it is naturally necessary to assume there is no integrity constraint in place to prevent such a write.

```
   ACC 1                     ACC 2                     ACC 3
   [   40 ]                  [   50 ]                  [   30 ]
-----------------------------------------------------------------------
   Transaction A             Time              Transaction B
-----------------------------------------------------------------------
   -                          |                  -
   -                          |                  -
   RETRIEVE ACC 1 :           t1                 -
      sum = 40                |                  -
   -                          |                  -
   RETRIEVE ACC 2 :           t2                 -
      sum = 90                |                  -
   -                          |                  -
   -                          t3               RETRIEVE ACC 3
   -                          |                  -
   -                          t4               UPDATE ACC 3 :
   -                          |                   30 ──► 20
   -                          |                  -
   -                          t5               RETRIEVE ACC 1
   -                          |                  -
   -                          t6               UPDATE ACC 1 :
   -                          |                   40 ──► 50
   -                          |                  -
   -                          t7               COMMIT
   -                          |
   RETRIEVE ACC 3 :           t8
      sum = 110, not 120      |
   -                          ▼
```

Fig. 16.4 Transaction *A* performs an inconsistent analysis

this example and the previous one: There is no question here of *A* being dependent on an uncommitted change, since *B* commits all of its updates before *A* sees ACC 3. *Note:* We observe in passing that the term *inconsistent analysis* ought by rights to be "*incorrect* analysis." However, we stay with the former term for historical reasons.

A Closer Look

Note: You might want to skip this subsection on a first reading.

Let us take a slightly closer look at the foregoing problems. Clearly, the operations that are of primary interest from a concurrency point of view are database retrievals and database updates; in other words, we can regard a transaction as consisting of a sequence of such operations *only* (apart from the necessary BEGIN TRANSACTION and COMMIT or ROLLBACK operations, of course). Let us agree to refer to those operations as simply *reads* and *writes*, respectively. Then it is clear that if *A* and *B* are concurrent transactions, problems can occur if *A* and *B* want to read or write the same database object, say tuple *t*. There are four possibilities:

- RR: *A* and *B* both want to read *t*. Reads cannot interfere with each other, so there is no problem in this case.

- RW: *A* reads *t* and then *B* wants to write *t*. If *B* is allowed to perform its write, then (as we saw in Fig. 16.4) the inconsistent analysis problem can arise; thus, we can say that inconsistent analysis is caused by a RW conflict. *Note:* If *B* does perform its write and *A* then reads *t* again, it will see a value different from what it saw before, a state of affairs referred to (somewhat inaptly) as **nonrepeatable read**; thus, nonrepeatable reads too are caused by RW conflicts.
- WR: *A* writes *t* and then *B* wants to read *t*. If *B* is allowed to perform its read, then (as we saw in Fig. 16.2, except that here we are reversing the roles of *A* and *B*) the uncommitted dependency problem can arise; thus, we can say that uncommitted dependencies are caused by WR conflicts. *Note: B*'s read, if it is allowed, is said to be a **dirty read**.
- WW: *A* writes *t* and then *B* wants to write *t*. If *B* is allowed to perform its write, then (as we saw in Fig. 16.1 and, in effect, in Fig. 16.3 also) the lost update problem can arise; thus, we can say that lost updates are caused by WW conflicts. *Note: B*'s write, if it is allowed, is said to be a **dirty write**.

16.3 LOCKING

As mentioned in Section 16.1, the problems of Section 16.2 can all be solved by means of a concurrency control mechanism called locking. The basic idea is simple: When some transaction *A* needs an assurance that some object it is interested in—typically a database tuple—will not change in some manner while its back is turned (as it were), it **acquires a lock** on that object. The effect of acquiring the lock is to "lock other transactions out of" the object in question, loosely speaking, and thus in particular to prevent them from changing it. Transaction *A* is thus able to continue its processing in the certain knowledge that the object in question will remain in a stable state for as long as that transaction *A* wishes it to.

We now proceed to give a more detailed explanation of the way locking works.

1. First, we assume the system supports two kinds of locks, **exclusive locks** (X locks) and **shared locks** (S locks), defined as indicated in the next two paragraphs. *Note:* X and S locks are sometimes called **write locks** and **read locks**, respectively. We assume until further notice that X and S locks are the only kinds available; see Section 16.9 for a discussion of other possibilities. We also assume until further notice that tuples are the only kinds of things that can be locked; again, see Section 16.9 for a discussion of other possibilities.
2. If transaction *A* holds an exclusive (X) lock on tuple *t*, then a request from some distinct transaction *B* for a lock of either type on *t* cannot be immediately granted.
3. If transaction *A* holds a shared (S) lock on tuple *t*, then:
   - A request from some distinct transaction *B* for an X lock on *t* cannot be immediately granted.
   - A request from some distinct transaction *B* for an S lock on *t* can and will be immediately granted (that is, *B* will now also hold an S lock on *t*).

	X	S	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

Fig. 16.5 Compatibility matrix for lock types X and S

These rules can be conveniently summarized by means of a **lock type compatibility matrix** (Fig. 16.5). That matrix is interpreted as follows: Consider some tuple *t;* suppose transaction *A* currently holds a lock on *t* as indicated by the entries in the column headings (dash = no lock); and suppose some distinct transaction *B* issues a request for a lock on *t* as indicated by the entries down the left side (for completeness we again include the "no lock" case). An "N" indicates a **conflict** (*B*'s request cannot be immediately granted), a "Y" indicates **compatibility** (*B*'s request can and will be immediately granted). The matrix is obviously symmetric.

Next, we introduce a **data access protocol** or **locking protocol** that makes use of X and S locks as just defined to guarantee that problems such as those described in Section 16.2 cannot occur:

1. A transaction that wishes to retrieve a tuple must first acquire an S lock on that tuple.
2. A transaction that wishes to update a tuple must first acquire an X lock on that tuple. Alternatively, if it already holds an S lock on the tuple, as it will in a RETRIEVE-UPDATE sequence, then it must *promote* or *upgrade* that S lock to X level.

   Note: We interrupt ourselves at this point to explain that requests for locks are usually *implicit*—a "tuple retrieve" operation implicitly requests an S lock on the relevant tuple, and a "tuple update" operation implicitly requests an X lock (or implicitly requests promotion of an existing S lock to X level) on the relevant tuple. Also, we take the term *update,* as always, to include INSERTs and DELETEs as well as UPDATEs *per se,* but the rules require some refinement to take care of INSERTs and DELETEs. We omit the details here.
3. If a lock request from transaction *B* cannot be immediately granted because it conflicts with a lock already held by transaction *A, B* goes into a **wait state.** *B* will wait until the lock request can be granted, which at the earliest will not be until *A*'s lock is released. *Note:* We say "at the earliest" because when *A*'s lock is released, another request for a lock on the pertinent tuple can be granted, but it might not be granted to *B*—there might be other transactions waiting by then. Of course, the system must guarantee that *B* does not wait forever (a possibility referred to as **livelock** or **starvation**). A simple way of providing such a guarantee is to service lock requests on a first-come/first-served basis.
4. X locks are released at end-of-transaction (COMMIT or ROLLBACK). S locks are normally released at that time also (at least, we will assume so until we get to Section 16.8).

The foregoing protocol is called **strict two-phase locking**. We will discuss it in more detail—in particular, we will explain why it has that name—in Section 16.6.

16.4 THE THREE CONCURRENCY PROBLEMS REVISITED

Now we are in a position to see how the strict two-phase locking protocol solves the three problems described in Section 16.2. Again we consider them one at a time.

The Lost Update Problem

Fig. 16.6 is a modified version of Fig. 16.1, showing what would happen to the interleaved execution of that figure under the strict two-phase locking protocol. Transaction *A*'s UPDATE at time *t3* is not accepted, because it is an implicit request for an X lock on *t*, and such a request conflicts with the S lock already held by transaction *B*; so *A* goes into a wait state. For analogous reasons, *B* goes into a wait state at time *t4*. Now both transactions are unable to proceed, so there is no question of any update being lost. We have thus solved the lost update problem by reducing it to another problem!—but at least we have solved the original problem. The new problem is called *deadlock*. It is discussed in Section 16.5.

Transaction A	Time	Transaction B
—		—
—		—
RETRIEVE t	t1	—
(acquire S lock on t)		—
—		—
—	t2	RETRIEVE t
—		(acquire S lock on t)
—		—
UPDATE t	t3	—
(request X lock on t)		—
wait		—
wait	t4	UPDATE t
wait		(request X lock on t)
wait		wait
wait		wait
wait		wait

Fig. 16.6 No update is lost, but deadlock occurs at time *t4*

The Uncommitted Dependency Problem

Figs. 16.7 and 16.8 are, respectively, modified versions of Figs. 16.2 and 16.3, showing what would happen to the interleaved executions of those figures under the strict two-phase locking protocol. Transaction *A*'s operation at time *t2* (RETRIEVE in Fig. 16.7, UPDATE in Fig. 16.8) is not accepted in either case, because it is an implicit request for a

Transaction A	*Time*	*Transaction B*
–		–
–		–
–	*t1*	UPDATE t
–		(acquire X lock on t)
–		–
RETRIEVE t	*t2*	–
(request S lock on t)		–
wait		–
wait	*t3*	COMMIT / ROLLBACK
wait		(release X lock on t)
resume : RETRIEVE t	*t4*	
(acquire S lock on t)		
–		

Fig. 16.7 Transaction *A* is prevented from seeing an uncommitted change at time *t2*

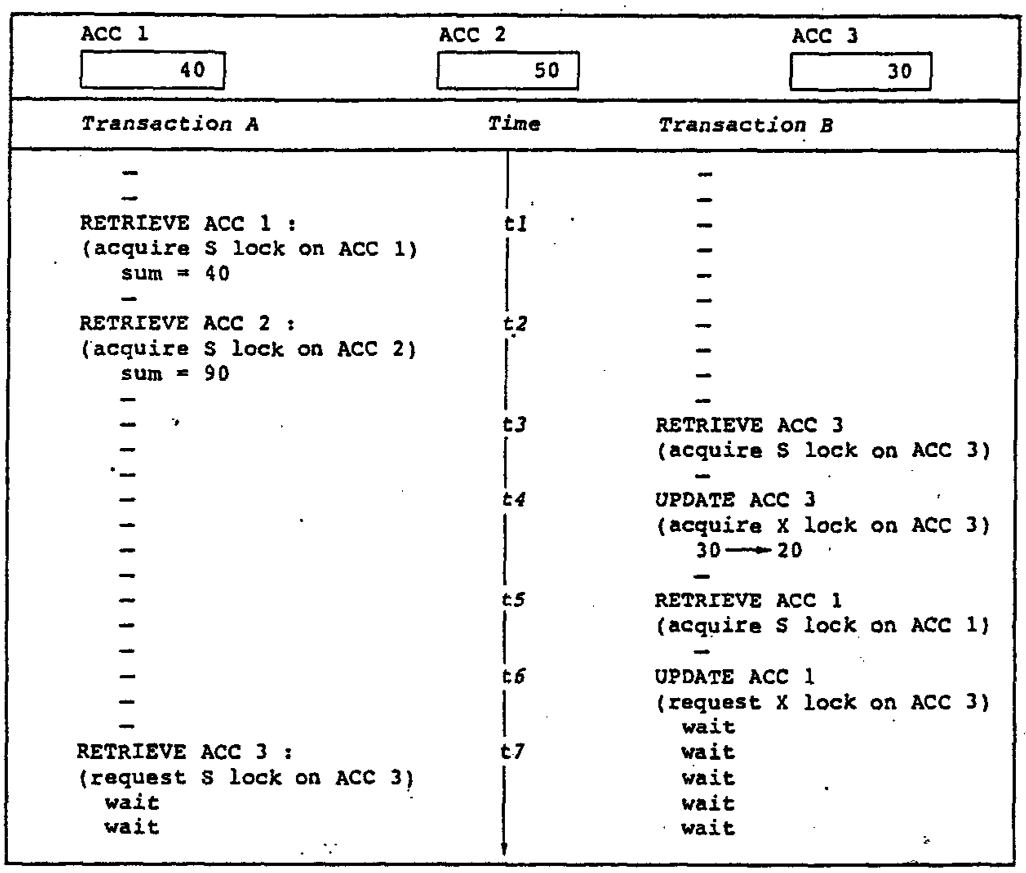
Transaction A	*Time*	*Transaction B*
–		–
–		–
–	*t1*	UPDATE t
–		(acquire X lock on t)
–		–
UPDATE t	*t2*	–
(request X lock on t)		–
wait		–
wait	*t3*	COMMIT / ROLLBACK
wait		(release X lock on t)
resume : UPDATE t	*t4*	
(acquire X lock on t)		
–		

Fig. 16.8 Transaction *A* is prevented from updating an uncommitted change at time *t2*

lock on *t*, and such a request conflicts with the X lock already held by *B:* so *A* goes into a wait state. It remains in that wait state until *B* reaches its termination (either COMMIT or ROLLBACK), when *B*'s lock is released and *A* is able to proceed; and at that point *A* sees a *committed* value (either the pre-*B* value, if *B* is rolled back, or the post-*B* value otherwise). Either way, *A* is no longer dependent on an uncommitted update, and so we have solved the original problem.

The Inconsistent Analysis Problem

Fig. 16.9 is a modified version of Fig. 16.4, showing what would happen to the interleaved execution of that figure under the strict two-phase locking protocol. Transaction *B*'s



```
ACC 1                          ACC 2                          ACC 3
[      40]                     [      50]                     [      30]

Transaction A                     Time          Transaction B

   -                               |               -
   -                               |               -
RETRIEVE ACC 1 :                   t1              -
(acquire S lock on ACC 1)          |               -
   sum = 40                        |               -
   -                               |               -
RETRIEVE ACC 2 :                   t2              -
(acquire S lock on ACC 2)          |               -
   sum = 90                        |               -
   -                               |               -
   -                               t3          RETRIEVE ACC 3
   -                               |           (acquire S lock on ACC 3)
   -                               |               -
   -                               t4          UPDATE ACC 3
   -                               |           (acquire X lock on ACC 3)
   -                               |              30 ──► 20
   -                               |               -
   -                               t5          RETRIEVE ACC 1
   -                               |           (acquire S lock on ACC 1)
   -                               |               -
   -                               t6          UPDATE ACC 1
   -                               |           (request X lock on ACC 3)
   -                               |              wait
RETRIEVE ACC 3 :                   t7             wait
(request S lock on ACC 3)          |              wait
  wait                             |              wait
  wait                             ▼              wait
```

Fig. 16.9 Inconsistent analysis is prevented, but deadlock occurs at time *t7*

UPDATE at time *t6* is not accepted, because it is an implicit request for an X lock on ACC 1, and such a request conflicts with the S lock already held by *A;* so *B* goes into a wait state. Likewise, transaction *A*'s RETRIEVE at time *t7* is also not accepted, because it is an implicit request for an S lock on ACC 3, and such a request conflicts with the X lock already held by *B;* so *A* goes into a wait state also. Again, therefore, we have solved the original problem (the inconsistent analysis problem, in this case) by forcing a deadlock. Again, deadlock is discussed in Section 16.5.

16.5 DEADLOCK

We have now seen how locking—more precisely, the strict two-phase locking protocol—can be used to solve the three basic concurrency problems. Unfortunately, however, we have also seen that locking can introduce problems of its own, principally the problem of *deadlock.* Two examples of deadlock were given in the previous section. Fig. 16.10 shows

Transaction A	Time	Transaction B
–		–
–		–
LOCK *r1* EXCLUSIVE	*t1*	–
–		–
–	*t2*	LOCK *r2* EXCLUSIVE
–		–
LOCK *r2* EXCLUSIVE	*t3*	–
wait		–
wait	*t4*	LOCK *r1* EXCLUSIVE
wait		wait
wait	↓	wait

Fig. 16.10 An example of deadlock

a slightly more general version of the problem: *r1* and *r2* in that figure are intended to represent any lockable resources, not necessarily just database tuples (see Section 16.9), and the "LOCK . . . EXCLUSIVE" statements are intended to represent any operations that request X locks, either explicitly or implicitly.

In general, then, deadlock is a situation in which two or more transactions are in a simultaneous wait state, each of them waiting for one of the others to release a lock before it can proceed.[2] Fig. 16.10 shows a deadlock involving two transactions, but deadlocks involving three, four, or more transactions are also possible, at least in principle. However, experiments with System R suggest that deadlocks almost never do involve more than two transactions in practice [16.9].

If a deadlock occurs, it is desirable that the system detect it and break it. Detecting the deadlock involves detecting a cycle in the **Wait-For Graph** (i.e., the graph of "who is waiting for whom"—see Exercise 16.4). Breaking the deadlock involves choosing one of the deadlocked transactions (i.e., one of the transactions in the cycle in the graph) as the victim and rolling it back, thereby releasing its locks and so allowing some other transaction to proceed. *Note:* In practice, not all systems do in fact detect deadlocks: some just use a timeout mechanism and simply assume that a transaction that has done no work for some prescribed period of time is deadlocked.

Observe, incidentally, that the victim has "failed" and been rolled back *through no fault of its own.* Some systems automatically restart such a transaction from the beginning, on the assumption that the conditions that caused the deadlock in the first place will probably not arise again. Other systems merely send a "deadlock victim" exception code back to the application; it is then up to the application to deal with the situation in some graceful manner. The first of these two approaches is clearly preferable from the application programmer's point of view. But even if the programmer does sometimes have to get involved, it is *always* desirable to conceal the problem from the end user, for obvious reasons.

[2] Deadlock is also referred to in the literature, somewhat colorfully, as *deadly embrace.*

Deadlock Avoidance

Instead of allowing deadlocks to occur and dealing with them when they do (which is what most systems do), it would be possible to avoid them entirely by modifying the locking protocol in various ways. We briefly consider one possible approach here. The approach in question (which was first proposed in the context of a distributed system [16.19], but could be used in a centralized system as well) comes in two versions, called *Wait-Die* and *Wound-Wait*. It works as follows:

- Every transaction is *timestamped* with its start time (which must be unique).
- When transaction *A* requests a lock on a tuple that is already locked by transaction *B*, then:
  - *Wait-Die: A* waits if it is older than *B;* otherwise, it "dies"—that is, *A* is rolled back and restarted.
  - *Wound-Wait: A* waits if it is younger than *B;* otherwise, it "wounds" *B*—that is, *B* is rolled back and restarted.
- If a transaction has to be restarted, it retains its original timestamp.

Note that the first component of the name (Wait or Wound) indicates in each case what happens if *A* is *older* than *B*. As you can see, Wait-Die means all waits consist of older transactions waiting for younger ones, Wound-Wait means all waits consist of younger transactions waiting for older ones. Whichever version is in effect, it is easy to see that deadlock cannot occur. It is also easy to see that every transaction is guaranteed to reach its proper conclusion eventually—that is, livelock cannot occur (no transaction will wait forever), and no transaction will be restarted over and over again forever either. The main drawback to the approach (either version) is that it does too many rollbacks.

16.6 SERIALIZABILITY

We have now laid the groundwork for explaining the crucial notion of *serializability*. Serializability is the generally accepted "criterion for correctness" for the interleaved execution of a set of transactions; that is, such an execution is considered to be correct if and only if it is serializable.[3] A given execution of a given set of transactions is serializable—and therefore correct—if and only if it is equivalent to (i.e., guaranteed to produce the same result as) some serial execution of the same transactions, where:

- A ***serial execution*** is one in which the transactions are run one at a time in some sequence.
- *Guaranteed* means that the given execution and the serial one always produce the same result as each other, no matter what the initial state of the database might be.

[3] Actually two kinds of serializability are defined in the literature, *conflict* serializability and *view* serializability. View serializability is of little practical interest, however, and it is usual to take the term *serializability* to mean conflict serializability specifically. See, for example, reference [16.21] for further discussion.

We justify this definition as follows:

1. Individual transactions are assumed to be correct; that is, they are assumed to transform a correct state of the database into another correct state, as discussed in Chapter 15.
2. Running the transactions one at a time in any serial order is therefore also correct ("any" serial order because individual transactions are assumed to be independent of one another).
3. It is thus reasonable to define an interleaved execution to be correct if and only if it is equivalent to some serial execution (i.e., *if and only if it is serializable*). Note that "only if"! The point is, a given interleaved execution might be nonserializable and yet still produce a result that happens to be correct, given some specific initial state of the database—see Exercise 16.3—but that would not be good enough; we want correctness to be *guaranteed* (i.e., independent of particular database states), not a matter of mere happenstance.

Referring back to the examples of Section 16.2 (Figs. 16.1–16.4), we can see that the problem in every case was precisely that the interleaved execution was not serializable—that is, it was never equivalent to running either *A*-then-*B* or *B*-then-*A*. And a study of Section 16.4 shows that the effect of the strict two-phase locking protocol is precisely to *force* serializability in every case. In Figs. 16.7 and 16.8, the interleaved execution is equivalent to *B*-then-*A*. In Figs. 16.6 and 16.9, a deadlock occurs, implying that one of the two transactions will be rolled back (and presumably run again later). If *A* is the one rolled back, then the interleaved execution again becomes equivalent to *B*-then-*A*.

Terminology: Given a set of transactions, any execution of those transactions, interleaved or otherwise, is called a **schedule**. Executing the transactions one at a time, with no interleaving, constitutes a **serial schedule**; a schedule that is not serial is an **interleaved schedule** (or simply a *nonserial* schedule). Two schedules are said to be **equivalent** if and only if, no matter what the initial state of the database, they are guaranteed to produce the same result as each other. Thus, a schedule is serializable, and correct, if and only if it is equivalent to some serial schedule.

Note that two different serial schedules involving the same transactions might well produce different results, and hence that two different interleaved schedules involving those same transactions might also produce different results, and yet both be correct. For example, suppose transaction *A* is of the form "Add 1 to x" and transaction *B* is of the form "Double x" (where x is some item in the database). Suppose also that the initial value of x is 10. Then the serial schedule *A*-then-*B* gives $x = 22$, whereas the serial schedule *B*-then-*A* gives $x = 21$. These two results are equally correct, and any schedule that is guaranteed to be equivalent to either *A*-then-*B* or *B*-then-*A* is likewise correct.

The concept of serializability was first introduced (although not by that name) by Eswaran *et al.* in reference [16.6]. That same paper also proved an important theorem, called the **two-phase locking theorem**, which we can state as follows:[4]

If all transactions obey the two-phase locking protocol, then all possible interleaved schedules are serializable.

[4] Two-phase locking has nothing to do with two-phase commit—they just have similar names.

The **two-phase locking protocol**, in turn, is as follows:

- Before operating on any object (e.g., a database tuple), a transaction must acquire a lock on that object.
- After releasing a lock, a transaction must never go on to acquire any more locks.

A transaction that obeys this protocol thus has two phases, a lock acquisition or "growing" phase and a lock releasing or "shrinking" phase. *Note:* In practice, the shrinking phase is often compressed into the single operation of COMMIT or ROLLBACK at end-of-transaction (a point we will come back to in Sections 16.7 and 16.8); if it is, then the protocol becomes the "strict" version described in Section 16.3.

The notion of serializability is a great aid to clear thinking in this potentially confusing area, and we therefore offer a few additional observations on it here. Let *I* be an interleaved schedule involving some set of transactions *T1, T2, ..., Tn*. If *I* is serializable, then there exists at least one serial schedule *S* involving *T1, T2, ..., Tn* such that *I* is equivalent to *S*. *S* is said to be a **serialization** of *I*.

Now let *Ti* and *Tj* be any two distinct transactions in the set *T1, T2, ..., Tn*. Let *Ti* precede *Tj* in the serialization *S*. In the interleaved schedule *I*, then, the effect must be as if *Ti* really did execute before *Tj*. In other words, an informal but very helpful characterization of serializability is that if *A* and *B* are any two transactions involved in some serializable schedule, then either *A* logically precedes *B* or *B* logically precedes *A* in that schedule; that is, **either B can see A's output or A can see B's**. (If *A* produces *x, y, ..., z* as output and *B* sees any of *x, y, ..., z* as input, then *B* sees them *either* all as they are after being output by *A or* all as they were before being output by *A*—not a mixture of the two.) Conversely, if the effect is not as if either *A* ran before *B* or *B* ran before *A*, then the schedule is not serializable and not correct.

Finally, we stress the point that if some transaction *A* is not two-phase (i.e., does not obey the two-phase locking protocol), then it is **always** possible to construct some other transaction *B* that can run interleaved with *A* in such a way as to produce an overall schedule that is not serializable and not correct. Now, in the interest of reducing resource contention and thereby improving performance and throughput, real-world systems typically do allow the construction of transactions that are not two-phase—that is, transactions that "release locks early" (prior to COMMIT) and then go on to acquire more locks. However, it should be clear that such transactions are a risky proposition; at best, allowing a given transaction *A* not to be two-phase amounts to a gamble that no interfering transaction *B* will ever coexist in the system with *A* (for if it does, then the system overall has the potential to produce wrong answers).

16.7 RECOVERY REVISITED

Given a serial schedule, individual transactions are obviously *recoverable*—such a transaction can always be undone and/or redone as necessary, using the techniques described in the previous chapter. However, it is not as obvious that transactions are still recoverable if

Transaction A	Time	Transaction B
–	|	–
–	|	–
–	t1	UPDATE t
–	|	–
RETRIEVE t	t2	–
–	|	–
COMMIT	t3	–
	|	–
	t4	ROLLBACK
	↓	

Fig. 16.11 An unrecoverable schedule

they are allowed to run interleaved. In fact, the uncommitted dependency problem discussed in Section 16.2 can cause recoverability problems, as we now show.

Assume for the moment that—as in Section 16.2—no locking protocol is in effect, and hence in particular that transactions never have to wait to acquire a lock. Now consider Fig. 16.11, which is a modified version of Fig. 16.2 (the difference is that transaction *A* now commits before transaction *B* rolls back). The problem here is that, in order to honor *B*'s ROLLBACK request and make it as if *B* never executed, we need to roll *A* back too, because *A* has seen one of *B*'s updates. But rolling back *A* is impossible, because *A* has already committed. Thus, the schedule shown in the figure is *unrecoverable*.

A sufficient condition for a schedule to be recoverable is as follows [15.2]:

If *A* sees any of *B*'s updates, then *A* must not commit before *B* terminates.

Clearly, we want our concurrency control mechanism—that is, our locking protocol, if locking is what we are using—to guarantee that all schedules are recoverable in this sense.

However, the foregoing is not the end of the story. Suppose now that we do have a locking protocol in place, and the protocol in question is *the nonstrict form* of two-phase locking, according to which a transaction can release locks before it terminates. Now consider Fig. 16.12, which is a modified version of Fig. 16.11 (the differences are that transaction *A* now does not commit before transaction *B*'s termination, but transaction *B* releases its lock on *t* "early"). As in Fig. 16.11, in order to honor *B*'s ROLLBACK request and make it as if *B* never executed, we need to roll *A* back too, because *A* has seen one of *B*'s updates. What is more, we *can* roll *A* back too, because *A* has not yet committed. But *cascading rollbacks* in this way is almost certainly undesirable; in particular, it is clear that if we permit the rollback of one transaction to cascade and cause the rollback of another, then we need to be prepared to deal with such "cascade chains" of arbitrary length. In other words, the trouble with the schedule shown in the figure is that it is not *cascade-free*.

A sufficient condition for a schedule to be cascade-free is as follows [15.2]:

If *A* sees any of *B*'s updates, then *A* must not do so before *B* terminates.

Transaction A	*Time*	*Transaction B*
–		–
–		–
–	*t1*	UPDATE t
–		(acquire X lock on t)
–		–
RETRIEVE t	*t2*	–
(request S lock on t)		–
wait		–
wait		–
wait	*t3*	(release lock on t)
wait		–
resume : RETRIEVE t	*t4*	–
(acquire S lock on t)		–
–		–
(forced rollback)	*t5*	ROLLBACK

Fig. 16.12 A schedule involving cascaded rollback

Strict two-phase locking will guarantee that all schedules are cascade-free, which is why that protocol is the one used in the vast majority of systems.[5] Also, it is easy to see that a cascade-free schedule must necessarily also be a recoverable schedule, as we defined that term previously.

16.8 ISOLATION LEVELS

Serializability guarantees *isolation* in the ACID sense. One direct and very desirable consequence is that if all schedules are serializable, then the application programmer writing the code for a given transaction *A* need pay absolutely no attention at all to the fact that some other transaction *B* might be executing in the system at the same time. However, it can be argued that the protocols used to guarantee serializability reduce the degree of concurrency or overall system throughput to unacceptable levels. In practice, therefore, systems usually support a variety of *levels* of "isolation" (in quotes because any level lower than the maximum means the transaction is not truly isolated from others after all, as we will soon see).

The **isolation level** that applies to a given transaction might be defined as the *degree of interference* the transaction in question is prepared to tolerate on the part of concurrent transactions. Now, if serializability is to be guaranteed, the only amount of interference that can possibly be tolerated is obviously none at all! In other words, the isolation level should be the maximum possible—for otherwise correctness, recoverability, and cascade-

[5] Actually, strict two-phase locking is slightly *too* strict; there is no need to keep S locks until end-of-transaction, just as long as transactions are still two-phase.

free schedules can no longer be guaranteed, in general. But the fact remains that, as already stated, systems typically do support levels that are less than the maximum, and in this section we briefly examine this issue.

At least five different isolation levels can be defined, though reference [16.10], the SQL standard, and DB2 each support just four. Generally speaking, the higher the isolation level, the less the interference (and the lower the concurrency); the lower the isolation level, the more the interference (and the higher the concurrency). By way of illustration, we consider two of the levels supported by DB2, *cursor stability* and *repeatable read*. Repeatable read (RR) is the maximum level: if all transactions operate at this level, all schedules are serializable. Under cursor stability (CS), by contrast, if a transaction *A*

- Obtains addressability to some tuple *t*,[6] and thus
- Acquires a lock on *t*, and then
- Relinquishes its addressability to *t* without updating it, and so
- Does not promote its lock to X level, then
- That lock can be released without having to wait for end-of-transaction.

But note that some other transaction *B* can now update *t* and commit the change. If transaction *A* subsequently comes back and looks at *t* again—note the violation of the two-phase locking protocol here!—it will see that change, and so might in effect see an inconsistent state of the database. Under repeatable read (RR), by contrast, *all* tuple locks (not just X locks) are held until end-of-transaction, and the problem just mentioned therefore cannot occur.

Points arising:

1. The foregoing problem is *not* the only problem that can occur under CS—it just happens to be the easiest to explain. But it unfortunately suggests that RR is needed only in the comparatively unlikely case that a given transaction needs to look at the same tuple twice. On the contrary, there are arguments to suggest that RR is *always* a better choice than CS; a transaction running under CS is not two-phase, and so (as explained in the previous section) serializability can no longer be guaranteed. The counterargument is that CS gives more concurrency than RR (probably but not necessarily).
2. The fact that serializability cannot be guaranteed under CS seems not to be very well understood in practice. The following is therefore worth repeating: *If transaction* T *operates at less than the maximum isolation level, then we can no longer guarantee that* T *if running concurrently with other transactions will transform a correct state of the database into another correct state.*
3. An implementation that supports any isolation level lower than the maximum will normally provide some explicit concurrency control facilities—typically explicit LOCK statements—in order to allow users to write their applications in such a way

[6] It does this by setting a *cursor* to point to the tuple, as explained in Chapter 4—hence the name "cursor stability." In the interest of accuracy, we should mention that the lock *T1* acquires on *t* in DB2 is actually an "update" (U) lock, not an S lock (see reference [4.21]).

as to guarantee safety in the absence of such a guarantee from the system itself. For example, DB2 provides an explicit LOCK TABLE statement, which allows users operating at a level less than the maximum to acquire explicit locks, over and above the ones that DB2 acquires automatically to enforce that level. (We remark in passing that the SQL standard includes no such explicit concurrency control mechanisms. See Section 16.11.)

By the way, please note that the foregoing characterization of RR as the maximum isolation level refers to repeatable read as implemented in DB2. Unfortunately, the SQL standard uses the same term *repeatable read* to mean an isolation level that is strictly lower than the maximum level (again, see Section 16.11).

Phantoms

One special problem that can occur if transactions operate at less than the maximum isolation level is the so-called *phantom* problem. Consider the following example (it is very contrived, but it suffices to illustrate the idea);

- First, suppose transaction *A* is computing the average account balance for all accounts held by customer Joe. Suppose there are currently three such accounts, each with a balance of $100. Transaction *A* thus scans the three accounts, acquiring shared locks on them as it proceeds, and obtains a result ($100).
- However, now suppose a concurrent transaction *B* is executing, the effect of which is to add another account to the database for customer Joe, with balance $200. Assume for definiteness that the new account is added after *A* has computed its $100 average. Assume too that after adding the new account, *B* immediately commits (releasing the exclusive lock it will have held on the new account).
- Now suppose *A* decides to scan the accounts for customer Joe again, counting them and summing their balances and then dividing the sum by the count (perhaps it wants to see if the average really is equal to the sum divided by the count). This time, it sees *four* accounts instead of three, and it obtains a result of $125 instead of $100!

Now, both transactions have followed strict two-phase locking here, and yet something has still gone wrong; to be specific, transaction *A* has seen something that did not exist the first time around—a phantom. As a consequence, serializability has been violated (the interleaved execution is clearly equivalent to neither *A*-then-*B* nor *B*-then-*A*).

Note carefully, however, that the problem here has nothing to do with two-phase locking *per se*. Rather, the problem is that transaction *A* did not lock what it logically needed to lock; instead of locking customer Joe's three accounts as such, it really needed to lock *the set of accounts held by Joe*, or in other words the *predicate* "account holder = Joe" (see references [16.6] and [16.13]).[7] If it could have done that, then transaction *B* would have had to wait when it tried to add its new account (because *B* would certainly request a lock on that new account, and that lock would conflict with the lock held by *A*).

[7] We might also say that *A* needed to lock the *nonexistence* of any other accounts belonging to Joe.

Although for reasons discussed in reference [15.12] most systems today do not support predicate locking as such, they do still generally manage to prevent "phantoms" from occurring by locking the *access path* used to get to the data under consideration. In the case of the accounts held by Joe, for example, if that access path happens to be an index on customer name, then the system can lock the entry in that index for customer Joe. Such a lock will prevent the creation of phantoms, because such creation would require the access path—the index entry, in this example—to be updated, and would thus require an X lock to be obtained on that access path. See reference [15.12] for further discussion.

16.9 INTENT LOCKING

Up to this point we have been assuming for the most part that the unit for locking purposes is the individual tuple. In principle, however, there is no reason why locks should not be applied to larger or smaller units of data—for example, an entire relvar, or even the entire database, or (going to the opposite extreme) a specific component within a specific tuple. We speak of **locking granularity** [16.10, 16.11]. As usual, there is a trade-off: The finer the granularity, the greater the concurrency; the coarser, the fewer the locks that need to be set and tested and the lower the overhead. For example, if a transaction has an X lock on an entire relvar, there is no need to set X locks on individual tuples within that relvar; on the other hand, no concurrent transaction will be able to obtain any locks on that relvar, or on tuples within that relvar, at all.

Suppose some transaction *T* does in fact request an X lock on some relvar *R*. On receipt of *T*'s request, the system must be able to tell whether any other transaction already has a lock on any tuple of *R*—for if it does, then *T*'s request cannot be granted at this time. How can the system detect such a conflict? It is obviously undesirable to have to examine every tuple in *R* to see whether any of them is currently locked by any other transaction, or to have to examine every existing lock to see whether any of them is for a tuple in *R*. Instead, therefore, we introduce another protocol, the **intent locking protocol**, according to which no transaction is allowed to acquire a lock on a tuple before first acquiring a lock—probably an *intent* lock (see the next paragraph)—on the relvar that contains it. Conflict detection in the example then becomes a comparatively simple matter of seeing whether any transaction has a conflicting lock *at the relvar level.*

Now, we have effectively suggested already that X and S locks make sense for whole relvars as well as for individual tuples. Following references [16.10, 16.11], we now introduce three additional kinds of locks, called **intent locks**, that also make sense for relvars, but not for individual tuples: **intent shared** (IS) locks, **intent exclusive** (IX) locks, and **shared intent exclusive** (SIX) locks. These new kinds of locks can be defined informally as follows. (We suppose that transaction *T* has requested a lock of the indicated type on relvar *R;* for completeness, we include definitions for types X and S as well.)

- *Intent shared (IS): T* intends to set S locks on individual tuples in *R*, in order to guarantee the stability of those tuples while they are being processed.
- *Intent exclusive (IX):* Same as IS, *plus T* might update individual tuples in *R* and will therefore set X locks on those tuples.

- *Shared (S): T* can tolerate concurrent readers, but not concurrent updaters, in *R* (*T* itself will not update any tuples in *R*).
- *Shared intent exclusive (SIX):* Combines S and IX; that is, *T* can tolerate concurrent readers, but not concurrent updaters, in *R*, *plus T* might update individual tuples in *R* and will therefore set X locks on those tuples.
- *Exclusive (X): T* cannot tolerate any concurrent access to *R* at all (*T* itself might or might not update individual tuples in *R*).

The formal definitions of these five lock types are given by an extended version of the lock type compatibility matrix first discussed in Section 16.3. See Fig. 16.13.

	X	SIX	IX	S	IS	–
X	N	N	N	N	N	Y
SIX	N	N	N	N	Y	Y
IX	N	N	Y	N	Y	Y
S	N	N	N	Y	Y	Y
IS	N	Y	Y	Y	Y	Y
–	Y	Y	Y	Y	Y	Y

Fig. 16.13 Compatibility matrix extended to include intent locks

Here now is a more precise statement of the intent locking protocol:

1. Before a given transaction can acquire an S lock on a given tuple, it must first acquire an IS or stronger lock (see later) on the relvar containing that tuple.
2. Before a given transaction can acquire an X lock on a given tuple, it must first acquire an IX or stronger lock (see later) on the relvar containing that tuple.

(Note, however, that this is still not a complete definition. See the annotation to reference [16.10].)

The notion touched on in the foregoing protocol of *relative lock strength* can be explained as follows. Refer to the **precedence graph** in Fig. 16.14. We say that lock type *L2* is stronger—that is, higher in the graph—than lock type *L1* if and only if, whenever there is an "N" (conflict) in *L1*'s column in the compatibility matrix for a given row, there is also an "N" in *L2*'s column for that same row (see Fig. 16.13). Note that a lock request that fails for a given lock type will certainly fail for a stronger lock type (and this fact implies that it is always safe to use a lock type that is stronger than strictly necessary). Note too that neither of S and IX is stronger than the other.

It is worth pointing out that, in practice, the relvar locks required by the intent locking protocol will usually be acquired implicitly. For a read-only transaction, for example, the system will probably acquire an IS lock implicitly on every relvar the transaction accesses. For an update transaction, it will probably acquire IX locks instead. But the system will probably also have to provide an explicit LOCK statement of some kind in

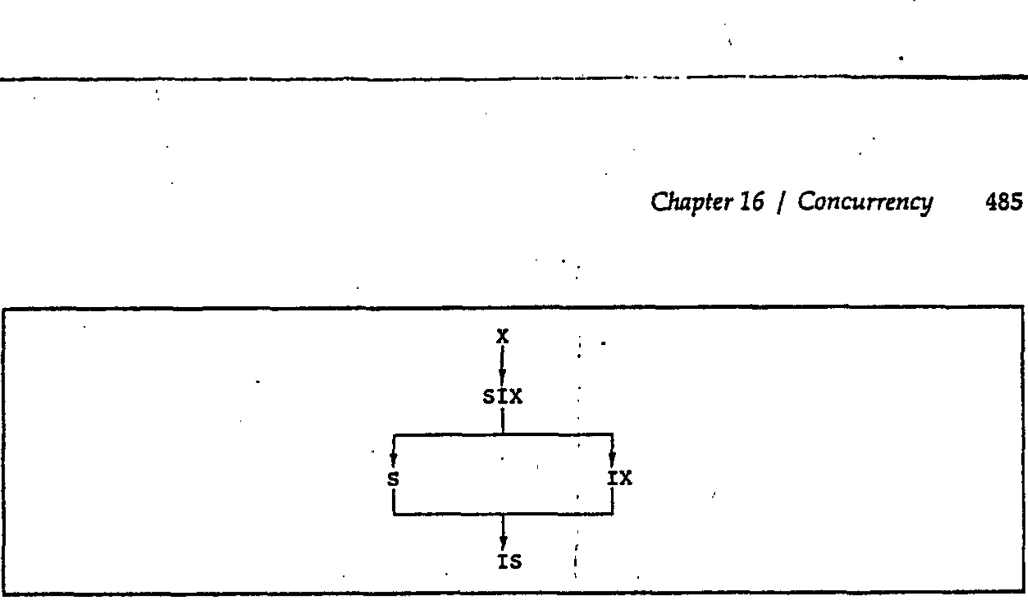


Fig. 16.14 Lock type precedence graph

order to allow transactions to acquire S, X, or SIX locks at the relvar level if they want them. Such a statement is supported by DB2, for example (for S and X locks only, not SIX locks).

We close this section with a remark on lock escalation, which is implemented in many systems and represents an attempt to balance the conflicting requirements of high concurrency and low lock management overhead. The basic idea is that when some prescribed threshold is reached, the system automatically replaces a collection of locks of fine granularity by a single lock of coarser granularity; for example, by trading in a set of individual tuple-level S locks and converting the IS lock on the containing relvar to an S lock. This technique seems to work well in practice [16.9].

16.10 DROPPING ACID

We promised in Chapter 15 that we would have more to say in this chapter regarding the ACID properties of transactions; in fact, we have some rather unorthodox opinions to offer on this topic, as will soon be clear.

Recall first that ACID is an acronym for *atomicity - correctness - isolation - durability*. Just to review briefly:

- *Atomicity:* Any given transaction is all or nothing.
- *Correctness* (usually called *consistency* in the literature): Any given transaction transforms a correct state of the database into another correct state, without necessarily preserving correctness at all intermediate points.
- *Isolation:* Any given transaction's updates are concealed from all other transactions, until the given transaction commits.
- *Durability:* Once a given transaction commits, its updates survive in the database, even if there is a subsequent system crash.

So ACID is a nice acronym—but do the concepts it represents really stand up to close examination? In this section, we present some evidence to suggest that the answer to this question is, in general, *no*.

Immediate Constraint Checking

We begin with what might look like a digression: a justification for our position, first articulated in Chapter 9, that all integrity constraints must be checked immediately (i.e., at end-of-statement), not deferred to end-of-transaction. We have at least four reasons for adopting this position, which we now proceed to explain.

1. As we know, a database can be regarded as a collection of propositions (assumed by convention to be true ones). And if that collection is ever allowed to include any inconsistencies, *then all bets are off.* We can never trust the answers we get from an inconsistent database; in fact, we can get *absolutely any answer whatsoever* from such a database (a proof of this fact appears in the annotation to reference [9.16] in Chapter 9). While the isolation or "I" property of transactions might mean that no more than one transaction will ever see any particular inconsistency, the fact remains that that particular transaction does see the inconsistency and can therefore produce wrong answers. Indeed, it is precisely because inconsistencies cannot be tolerated, not even if they are never visible to more than one transaction at a time, that the constraints need to be enforced in the first place.
2. In any case, it cannot be guaranteed that a given inconsistency (assuming such a thing is permitted) *will* be seen by just one transaction. Only if they follow certain protocols—certain *unenforced* (and in fact unenforceable) protocols—can transactions truly be guaranteed to be isolated from one another. For example, if transaction *A* sees an inconsistent state of the database and so writes inconsistent data to some file *F*, and transaction *B* then reads that same information from file *F*, then *A* and *B* are not really isolated from each other (regardless of whether they run concurrently or otherwise).[8] In other words, the "I" property of transactions is suspect, to say the least.
3. The previous edition of this book stated that relvar constraints were checked immediately but database constraints were checked at end-of-transaction (a position that many writers concur with, though they usually use different terminology). But *The Principle of Interchangeability* (of base and derived relvars—see Chapter 9) implies that the very same real-world constraint might be a relvar constraint with one design for the database and a database constraint with another! Since relvar constraints must *obviously* be checked immediately, it follows that database constraints must be checked immediately too.
4. The ability to perform "semantic optimization" requires the database to be consistent *at all times,* not just at transaction boundaries. *Note:* Semantic optimization is a technique for using integrity constraints to simplify queries in order to improve performance. Clearly, if the constraints are not satisfied, then the simplifications will be invalid. For further discussion, see Chapter 18.

Of course, the "conventional wisdom" is that database constraint checks, at least, surely *have* to be deferred. As a trivial example, suppose the suppliers-and-parts database is subject to the constraint "Supplier S1 and part P1 are in the same city." If supplier S1

[8] In fact the problem arises even if *A* does not see an inconsistent state of the database; it is still possible that *A* might write inconsistent data to some file that is subsequently read by *B*.

moves, say from London to Paris, then part P1 must move from London to Paris as well. The conventional solution to this problem is to wrap the two updates up into a single transaction, like this:

```
BEGIN TRANSACTION ;
UPDATE S WHERE S# = S# ('S1') { CITY := 'Paris' } ;
UPDATE P WHERE P# = P# ('P1') { CITY := 'Paris' } ;
COMMIT ;
```

In this conventional solution, the constraint is checked at COMMIT, and the database is inconsistent between the two UPDATE operations. Note in particular that if the transaction performing the UPDATEs were to ask the question "Are suppller S1 and part P1 in the same city?" between the two UPDATE operations, it would get the answer *no.*

Recall, however, that we require support for a *multiple assignment* operator, which lets us carry out several assignments as a single operation (i.e., within a single statement), without any integrity checking being done until all of the assignments in question have been executed. Recall too that INSERT, DELETE, and UPDATE are just shorthand for certain assignment operations. In the example, therefore, we should be able to perform the desired updating as a single operation, thus:

```
UPDATE S WHERE S# = S# ('S1') { CITY := 'Paris' } ,
UPDATE P WHERE P# = P# ('P1') { CITY := 'Paris' } ;
```

Now no integrity checking is done until both UPDATEs have been done (i.e., "until we reach the semicolon"). Note too that there is now no way for the transaction to see an inconsistent state of the database between the two UPDATEs, because the notion of "between the two UPDATEs" now has no meaning.

It follows from this example that if multiple assignment were supported, there would be no need for deferred checking in the traditional sense (i.e., checking that is deferred to end-of-transaction).

Now we turn to the ACID properties *per se*. However, it suits our purposes better to discuss them in the order C-I-D-A.

Correctness

We have already given our reasons (in Chapter 15) for preferring the term *correctness* here over the more usual *consistency.* In fact, however, the literature usually seems to equate the two concepts. Here, for example, is a quote from the glossary in the book by Gray and Reuter [15.12]:

Consistent. Correct.

And the same book defines the consistency property of transactions thus:

Consistency. A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction be a correct program [*sic*].

But if integrity constraints are always checked immediately, the database is *always* consistent—not necessarily correct!—and transactions always transform a consistent state of the database into another consistent state *a fortiori.*

So if the C in ACID stands for consistency, then in a sense the property is trivial;[9] and if it stands for correctness, then it is unenforceable. Either way, therefore, the property is essentially meaningless, at least from a formal standpoint. As already indicated, our own preference would be to say that the C stands for correctness; then we can go on to regard "the correctness property" not really as a property as such, but rather as a desideratum.

Isolation

We turn now to the isolation property. As already explained earlier in this section, in the subsection "Immediate Constraint Checking," this property too is somewhat suspect. It is at least true that if every transaction behaves as if it were the only transaction in the system, then a proper concurrency control mechanism will guarantee isolation (and serializability). However, "behaving as if it were the only transaction in the system" implies among other things that the transaction in question must:

- Make no attempt, intentional or otherwise, to communicate with other transactions, concurrent or otherwise
- Make no attempt even to recognize the possibility (by specifying an isolation level lower than the maximum) that other transactions might exist in the system

So the isolation property too is really more of a desideratum than an ironclad guarantee. What is more, real-world systems typically provide explicit mechanisms—namely, isolation levels lower than the maximum—whose effect is precisely to undermine isolation.

Durability

We turn now to the durability property. This property is reasonable, thanks to the system's recovery mechanism, *as long as there is no transaction nesting*—and indeed we have assumed that such is the case, prior to this point. Suppose, however, that transaction nesting is supported. To be specific, suppose transaction *B* is nested inside transaction *A*, and the following sequence of events occurs:

```
BEGIN TRANSACTION (transaction A) ;
   ...
   BEGIN TRANSACTION (transaction B) ;
   transaction B updates tuple t ;
   COMMIT (transaction B) ;
   ...
ROLLBACK (transaction A) ;
```

If *A*'s ROLLBACK is honored, then *B* is effectively rolled back too (because *B* is really part of *A*), and *B*'s effects on the database are thus not "durable"; in fact, *A*'s ROLLBACK causes tuple *t* to be restored to its pre-*A* value. In other words, the durability property can

[9] As a matter of fact, it would be trivial even if constraints are not checked immediately—the transaction would still be rolled back, and thus in effect have never executed, if it violated any constraint. In other words, it would still be the case that transactions have a lasting effect on the database only if they do not violate any constraints.

no longer be guaranteed, at least not for a transaction like *B* in the example that is nested inside some other transaction.

Now, many writers (beginning with Davies in reference [15.8]) have in fact proposed the ability to nest transactions in the manner suggested by the foregoing example. Reference [15.15] claims that such support is desirable for at least three reasons: intra-transaction parallelism, intra-transaction recovery control, and system modularity. As the example indicates, in a system with such support, COMMIT by an inner transaction commits that transaction's updates, *but only to the next outer level.* In effect, the outer transaction has veto power over the inner transaction's COMMIT—if the outer transaction does a rollback, the inner transaction is rolled back too. In the example, *B*'s COMMIT is a COMMIT to *A* only, not to the outside world, and indeed that COMMIT is subsequently revoked (rolled back).

Note: We remark that nested transactions can be thought of as a generalization of *savepoints*. Savepoints allow a transaction to be structured as a linear *sequence* of actions that are executed one at a time (and rollback can occur at any time to the start of any earlier action in the sequence). Nesting, by contrast, allows a transaction to be structured, recursively, as a *hierarchy* of actions that are executed concurrently. In other words:

- BEGIN TRANSACTION is extended to support "subtransactions" (i.e., if BEGIN TRANSACTION is issued when a transaction is already running, it starts a *child* transaction).
- COMMIT "commits" but only within the *parent scope* (if this transaction is a child).
- ROLLBACK undoes work, but only back to the start of this particular transaction (including child, grandchild, etc., transactions but *not* including the parent transaction, if any).

To revert to the main thread of our discussion, we now see that the durability property of transactions applies only at the outermost level[10] (in other words, to transactions not nested inside any other transaction). Thus we see that that property too is less than 100 percent guaranteed, in general.

Atomicity

Finally, we turn to the atomicity property. Like the durability property, this property is guaranteed by the system's recovery mechanism (even with nested transactions). Our objections here are a little different. To be specific, we simply observe that if the system supported multiple assignment, there would be no need for transactions as such to have the atomicity property; rather, it would be sufficient for *statements* to do so. What is more, it is *necessary* for statements to do so, too, for reasons already discussed in detail elsewhere.[11]

[10] Reference [15.15] says the same is true for the consistency property as well, because like most of the rest of the literature it assumes that the outermost transaction is not required to preserve consistency at intermediate points. However, we reject this position for reasons already explained.

[11] We remind you in passing that most statements in the SQL standard in particular do have the atomicity property.

Concluding Remarks

We can summarize this section by means of the following somewhat rhetorical questions:

- *Is the transaction a unit of work?* Yes, but only if multiple assignment is not supported.
- *Is it a unit of recovery?* Same answer.
- *Is it a unit of concurrency?* Same answer.
- *Is it a unit of integrity?* Yes, but only if "all constraint checking immediate" is not supported.

Note: We answer the questions the way we do despite various remarks made in earlier chapters (and in earlier editions of the book) that adhere more to "the conventional wisdom" in this area.

Overall, then, we conclude that the transaction concept is important more from a pragmatic point of view than it is from a theoretical one. Please nnderstand that this conclusion is not meant to be disparaging! We have nothing but respect for the many elegant and useful results obtained from over 25 years of transaction management research. We are merely observing that we now have a better understanding of some of the assumptions on which that research has been based—a better understanding of the crucial role of integrity constraints in particular, plus a recognition of the need to support multiple assignment as a primitive operator. Indeed, it would be surprising if a change in assumptions did not lead to a change in conclusions.

16.11 SQL FACILITIES

The SQL standard does not provide any explicit locking facilities; in fact, it does not mention locking, as such, at all.[12] However, it does require the implementation to provide the usual guarantees regarding interference, or rather lack thereof, among concurrently executing transactions. Most importantly, it requires that updates made by a given transaction *T1* not be visible to any distinct transacrion *T2* until and unless transaction *T1* commits. *Note:* The foregoing assumes that all transactions execute at isolation level READ COMMITTED, REPEATABLE READ, or SERIALIZABLE (see the next paragraph). Special considerations apply to transactions executing at the READ UNCOMMITTED level, which (a) are allowed to perform "dirty reads" but (b) are required to be READ ONLY (if READ WRITE were permitted, recoverability could no longer be guaranteed).

Recall now from Chapter 15 that SQL isolation levels are specified on START TRANSACTION. There are four possibilities—SERIALIZABLE, REPEATABLE READ, READ COMMITTED, and READ UNCOMMITTED.[13] The default is SERIALIZABLE;

[12] The omission is deliberate—the idea is that the system should be free to use any concurrency control mechanism it likes, just as long as it implements the desired functionality.

[13] SERIALIZABLE is not a good keyword here, since it is *schedules* that are supposed to be serializable, not *transactions*. A better term might be just TWO PHASE, meaning the transaction will obey (or will be forced to obey) the two-phase locking protocol.

Isolation level	*Dirty read*	*Nonrepeatable read*	*Phantom*
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

Fig. 16.15 SQL isolation levels

if any of the other three is specified, the implementation is free to assign some higher level, where "higher" is defined in terms of the ordering SERIALIZABLE > REPEATABLE READ > READ COMMITTED > READ UNCOMMITTED.

If all transactions execute at isolation level SERIALIZABLE (the default), then the interleaved execution of any set of concurrent transactions is guaranteed to be serializable. However, if any transaction executes at a lesser isolation level, then serializability can be violated in a variety of different ways. The standard defines three kinds of violations—*dirty read, nonrepeatable read,* and *phantoms* (the first two of these were explained in Section 16.2 and the third was explained in Section 16.8)—and the various isolation levels are defined in terms of the violations they permit.[14] They are summarized in Fig. 16.15 ("Y" means the violation can occur, "N" means it cannot).

We close this section by reminding you that the REPEATABLE READ of the SQL standard and the "repeatable read" (RR) of DB2 are not the same thing. In fact, DB2's RR is the same as the standard's SERIALIZABLE.

16.12 SUMMARY

We have examined the question of **concurrency control**. We began by looking at three problems that can arise in an interleaved execution of concurrent transactions if no such control is in place: the **lost update problem**, the **uncommitted dependency problem**, and the **inconsistent analysis problem**. All of these problems arise from schedules that are not **serializable**—that is, not equivalent to some serial schedule involving the same transactions.

The most widespread technique for dealing with such problems is **locking**. There are two basic types of locks, **shared** (S) and **exclusive** (X). If a transaction has an S lock on an object, other transactions can also acquire an S lock on that object, but not an X lock; if a transaction has an X lock on an object, no other transaction can acquire a lock on the object at all, of either type. Then we introduce a protocol for the use of these locks to ensure that the lost update and other problems cannot occur: Acquire an S lock on everything retrieved, acquire an X lock on everything updated, and keep all locks until end-of-transaction. This protocol guarantees serializability.

[14] But see references [16.2] and [16.14].

The protocol just described is a strict form of the **two-phase locking protocol**. It can be shown that if all transactions obey this protocol, then all schedules are serializable—the **two-phase locking theorem**. A serializable schedule implies that if *A* and *B* are any two transactions involved in that schedule, then either *A* can see *B*'s output or *B* can see *A*'s. The two-phase locking also guarantees **recoverability** and **cascade-free schedules**; unfortunately, it can also lead to **deadlocks**. Deadlocks can be resolved by choosing one of the deadlocked transactions as the victim and rolling it back (thereby releasing all of its locks).

Anything less than full serializability cannot be guaranteed to be safe (in general). However, systems typically allow transactions to operate at a **level of isolation** that is indeed unsafe, with the aim of reducing resource contention and increasing transaction throughput. We described one such "unsafe" level, **cursor stability** (this is the DB2 term; the SQL standard term is READ COMMITTED).

Next we briefly considered the question of **lock granularity** and the associated idea of **intent locking**. Basically, before a transaction can acquire a lock of any kind on some object, say a database tuple, it must first acquire an appropriate intent lock (at least) on the "parent" of that object (e.g., the containing relvar, in the case of a tuple). In practice, such intent locks will usually be acquired implicitly, just as S and X locks on tuples are usually acquired implicitly. However, **explicit LOCK** statements of some kind should be provided in order to allow a transaction to acquire stronger locks than the ones acquired implicitly (though the SQL standard provides no such mechanism).

Next, we took another look at the so-called **ACID** properties of transactions, concluding that matters are not nearly as clear-cut in this area as is commonly supposed. Finally, we outlined SQL's concurrency control support. Basically, SQL does not provide any explicit locking capabilities at all; however, it does support various isolation levels—**SERIALIZABLE**, **REPEATABLE READ**, **READ COMMITTED**, and **READ UNCOMMITTED**, which the DBMS will probably implement by means of locking behind the scenes.

EXERCISES

16.1 Explain *serializability* in your own words.

16.2 State (a) the two-phase locking protocol; (b) the two-phase locking theorem. Explain exactly how two-phase locking deals with RW, WR, and WW conflicts.

16.3 Let transactions *T1*, *T2*, and *T3* be defined to perform the following operations:

T1 : Add one to *A*
T2 : Double *A*
T3 : Display *A* on the screen and then set *A* to one

(where *A* is some numeric item in the database).

a. Suppose transactions *T1*, *T2*, *T3* are allowed to execute concurrently. If *A* has initial value zero, how many possible correct results are there? Enumerate them.

b. Suppose the internal structure of *T1, T2, T3* is as indicated in the following pseudocode. If the transactions execute *without* any locking, how many possible schedules are there?

T1	*T2*	*T3*
R1: RETRIEVE *A*	R2: RETRIEVE *A*	R3: RETRIEVE *A*
INTO *a1* ;	INTO *a2* ;	INTO *a3* ;
a1 := *a1* + 1 ;	*a2* := *a2* * 2 ;	display *a3* ;
U1: UPDATE *A*	U2: UPDATE *A*	U3: UPDATE *A*
FROM *a1* ;	FROM *a2* ;	FROM 1 ;

c. If again *A* has initial value zero, are there any interleaved schedules that in fact produce a correct result and yet are not serializable?

d. Are there any schedules that are in fact serializable but could not be produced if all three transactions obeyed the two-phase locking protocol?

16.4 The following represents the sequence of events in a schedule involving transactions *T1, T2, ..., T12. A, B, ..., H* are items in the database.

```
time t0   ..........
time t1  (T1)   : RETRIEVE A ;
time t2  (T2)   : RETRIEVE B ;
  ...    (T1)   : RETRIEVE C ;
  ...    (T4)   : RETRIEVE D ;
  ...    (T5)   : RETRIEVE A ;
  ...    (T2)   : RETRIEVE E ;
  ...    (T2)   : UPDATE E ;
  ...    (T3)   : RETRIEVE F ;
  ...    (T2)   : RETRIEVE F ;
  ...    (T5)   : UPDATE A ;
  ...    (T1)   : COMMIT ;
  ...    (T6)   : RETRIEVE A ;
  ...    (T5)   : ROLLBACK ;
  ...    (T6)   : RETRIEVE C ;
  ...    (T6)   : UPDATE C ;
  ...    (T7)   : RETRIEVE G ;
  ...    (T8)   : RETRIEVE H ;
  ...    (T9)   : RETRIEVE G ;
  ...    (T9)   : UPDATE G ;
  ...    (T8)   : RETRIEVE E ;
  ...    (T7)   : COMMIT ;
  ...    (T9)   : RETRIEVE H ;
  ...    (T3)   : RETRIEVE G ;
  ...    (T10)  : RETRIEVE A ;
  ...    (T9)   : UPDATE H ;
  ...    (T6)   : COMMIT ;
  ...    (T11)  : RETRIEVE C ;
  ...    (T12)  : RETRIEVE D ;
  ...    (T12)  : RETRIEVE C ;
  ...    (T2)   : UPDATE F ;
  ...    (T11)  : UPDATE C ;
  ...    (T12)  : RETRIEVE A ;
  ...    (T10)  : UPDATE A ;
  ...    (T12)  : UPDATE D ;
  ...    (T4)   : RETRIEVE G ;
time t36  ..........
```

Assume that RETRIEVE *i* (if successful) acquires an S lock on *i*, and UPDATE *i* (if successful) promotes that lock to X level. Assume also that all locks are held until end-of-transaction. Draw a *Wait-For Graph* (showing who is waiting for whom) representing the state of affairs at time *t36*. Are there any deadlocks at that time?

16.5 Consider the concurrency problems illustrated in Figs. 16.1–16.4 once again. What would happen in each case if all transactions were executing under isolation level CS instead of RR? *Note:* CS and RR here refer to DB2's isolation levels as described in Section 16.8.

16.6 Give both informal and formal definitions of lock types X, S, IX, IS, and SIX.

16.7 Define the notion of relative lock strength and give the corresponding precedence graph.

16.8 Define the intent locking protocol in its most general form. What is the purpose of that protocol?

16.9 SQL defines three concurrency problems: *dirty read, nonrepeatable read,* and *phantoms.* How do these relate to the three concurrency problems identified in Section 16.2?

16.10 Sketch an implementation mechanism for the multi-version concurrency control protocols described briefly in the annotation to reference [16.1].

REFERENCES AND BIBLIOGRAPHY

In addition to the following, see also references [15.2], [15.10], and (especially) [15.12] in Chapter 15.

16.1 R. Bayer, M. Heller, and A. Reiser: "Parallelism and Recovery in Database Systems," *ACM TODS 5*, No. 2 (June 1980).

As noted in Chapter 15, newer application areas (e.g., hardware and software engineering) often involve complex processing requirements for which the classical transaction management controls as described in the body of this chapter and its predecessor are not well suited. The basic problem is that complex transactions might last for hours or days, instead of for just a few milliseconds at most as in traditional systems. As a consequence:

1. Rolling back a transaction all the way to the beginning might cause the loss of an unacceptably large amount of work.
2. The use of conventional locking might cause unacceptably long delays waiting for locks to be released.

The present paper is one of several to address such concerns (others include references [16.8], [16.12], [16.15], and [16.20]). It proposes a concurrency control technique called **multi-version locking** (also known as multi-version read, and now implemented in some commercial products). The biggest advantage of the technique is that read operations never have to wait—any number of readers *and one writer* can operate on the same logical object simultaneously. To be more specific:

- Reads are never delayed (as just stated).
- Reads never delay updates.
- It is never necessary to roll back a read-only transaction.
- Deadlock is possible only between update transactions.

These advantages are particularly significant in distributed systems—see Chapter 21—where updates can take a long time and read-only queries might thus be unduly delayed (and *vice versa*). The basic idea is as follows:

- If transaction *B* asks to read an object that transaction *A* currently has update access to, transaction *B* is given access to a *previously committed* version of that object (such a version must exist anyway in the system somewhere—probably in the log—for recovery purposes).
- If transaction *B* asks to update an object that transaction *A* currently has read access to, transaction *B* is given access to that object, while transaction *A* retains access to its own version of the object (which is now really the previous version).

- If transaction *B* asks to update an object that transaction *A* currently has update access to, transaction *B* goes into a wait state[15] (deadlock, and hence forced rollback, are thus still possible, as noted earlier).

Of course, the approach includes appropriate controls to ensure that each transaction always sees a consistent state of the database.

16.2 Hal Berenson *et al.*: "A Critique of ANSI SQL Isolation Levels," Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif. (May 1995).

This paper criticizes the attempt on the part of "ANSI SQL" [*sic*] to characterize isolation levels in terms of serializability violations (see Section 16.11): "[The] definitions fail to properly characterize several popular isolation levels, including the standard locking implementations of the levels covered." The paper goes on to point out in particular that the standard fails to prohibit "dirty writes" (see Section 16.2).

It does seem to be true that the standard does not prohibit dirty writes explicitly. What it actually says is the following (slightly reworded here):

- "The execution of concurrent transactions at isolation level SERIALIZABLE is guaranteed to be serializable." Thus, if all transactions operate at isolation level SERIALIZABLE, the implementation is *required* to prohibit dirty writes, since dirty writes would certainly violate serializability.
- "The four isolation levels guarantee that . . . no updates will be lost." This claim is just wishful thinking—the definitions of the four isolation levels by themselves do *not* provide any such guarantee—but it does indicate that the standard definers *intended* to prohibit dirty writes.
- "Changes made by one transaction cannot be perceived by other transactions [except those with isolation level READ UNCOMMITTED] until the original transaction [commits]." The question here is, what exactly does *perceived* mean? Would it be possible for a transaction to update a piece of "dirty data" without "perceiving" it?

See also reference [16.14].

16.3 Philip A. Bernstein and Nathan Goodman: "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems," Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canada (October 1980).

Discusses a collection of approaches to concurrency control based not on locking but on timestamping. The basic idea is that if transaction *A* starts execution before transaction *B*, then the system should behave as if *A* actually executed in its entirety before *B* started (as in a genuine serial schedule). It follows that:

1. *A* must never be allowed to see any of *B*'s updates.
2. *A* must never be allowed to update anything *B* has already seen.

These two requirements can be enforced as follows: For any given database request, the system compares the timestamp of the requesting transaction with the timestamp of the transaction that last retrieved or updated the requested tuple; if there is a conflict, then the requesting transaction can simply be restarted with a new timestamp (as in the so-called *optimistic* methods [16.16]).

[15] In other words, WW conflicts can still occur, and we assume here that locking is used to resolve them. Other techniques (e.g., timestamping [16.3]) might perhaps be used instead.

As the title of the paper suggests, timestamping was originally introduced in the context of a distributed system (where it was felt that locking imposed intolerable overheads, because of the messages needed to test and set locks, etc.). It is almost certainly not appropriate in a nondistributed system. Indeed, there is considerable skepticism as to its practicality in distributed systems also. One obvious problem is that each tuple has to carry the timestamp of the transaction that last *retrieved* it (as well as the timestamp of the transaction that last updated it), which implies that every read becomes a write! Another problem is that *B* must not see any of *A*'s updates until *A* commits, implying that (in effect) *A*'s updates are "locked exclusive" until commit anyway. In fact, reference [15.12] claims that timestamping schemes are really just a degenerate case of optimistic concurrency control schemes [16.16], which in turn suffer from problems of their own.

Note: A notion much discussed in the literature, "Thomas's write rule" [16.22], is effectively a refinement on the foregoing scheme; it is based on the idea that certain updates can be skipped because they are already obsolete (the user-level request is honored but no physical update is done).

16.4 M. W. Blasgen, J. N. Gray, M. Mitoma, and T. G. Price: "The Convoy Phenomenon," *ACM Operating Systems Review 13*, No. 2 (April 1979).

The convoy phenomenon is a problem encountered with high-traffic locks, such as the lock needed to write a record to the log, in systems with *preemptive scheduling*. ("Scheduling" here refers to the problem of allocating machine cycles to transactions, not to the interleaving of database operations from different transactions as discussed in the body of this chapter.) The problem is as follows. If a transaction *T* is holding a high-traffic lock and is preempted by the system scheduler—that is, forced into a wait state, perhaps because its timeslice has expired—then a *convoy* of transactions will form, all waiting for their turn at the high-traffic lock. When *T* comes out of its wait state, it will soon release the lock, but (precisely because the lock is high-traffic) it will probably rejoin the convoy before the next transaction has finished with the resource, will therefore not be able to continue processing, and so will go into a wait state again.

The root of the problem is that the scheduler is usually part of the underlying operating system, not the DBMS, and is therefore based on different design assumptions. As the authors observe, a convoy, once established, tends to be stable; the system is in a state of "lock thrashing," most of the machine cycles are devoted to process switching, and not much useful work is being done. A suggested solution—barring the possibility of replacing the scheduler—is to grant the lock not on a first-come/first-served basis but instead in random order.

16.5 Stephen Blott and Henry F. Korth: "An Almost-Serial Protocol for Transaction Execution in Main-Memory Database Systems," Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong (August 2002).

Proposes a serializability mechanism for main-memory systems that avoids the use of locks entirely.

16.6 K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger: "The Notions of Consistency and Predicate Locks in a Data Base System," *CACM 19*, No. 11 (November 1976).

The paper that first put the subject of concurrency control on a sound theoretical footing.

16.7 Peter Franaszek and John T. Robinson: "Limitations on Concurrency in Transaction Processing," *ACM TODS 10*, No. 1 (March 1985).

See the annotation to reference [16.16].

16.8 Peter A. Franaszek, John T. Robinson, and Alexander Thomasian: "Concurrency Control for High Contention Environments," *ACM TODS 17*, No. 2 (June 1992).

This paper claims that, for a variety of reasons, future transaction processing systems are likely to involve a significantly greater degree of concurrency than the systems of today, and that there is therefore likely to be substantially more data contention in such systems. The authors then present "a number of [nonlocking] concurrency control concepts and transaction scheduling techniques that are applicable to high-contention environments" that—it is claimed, on the basis of experiments with simulation models—"can offer substantial benefits" in such environments.

16.9 J. N. Gray: "Experience with the System R Lock Manager," IBM San Jose Research Laboratory internal memo (Spring 1980).

This reference is really just a set of notes, not a finished paper, and its findings might be a little out of date by now. Nevertheless, it does contain some interesting claims, the following among them:

- Locking imposes about ten percent overhead on online transactions, about one percent on batch transactions.
- It is desirable to support a variety of lock granularities.
- Automatic lock escalation works well.
- Repeatable read (RR) is more efficient, as well as safer, than cursor stability (CS).
- Deadlocks are rare in practice and never involve more than two transactions.
- Almost all deadlocks (97 percent) could be avoided by supporting *U locks*, as DB2 does but System R did not.

Note: U locks are defined to be compatible with S locks but not with other U locks, and certainly not with X locks. For further details, see reference [4.21].

16.10 J. N. Gray, R. A. Lorie, and G. R. Putzolu: "Granularity of Locks in a Large Shared Data Base," Proc. 1st Int. Conf. on Very Large Data Bases, Framingham, Mass. (September 1975).

The paper that introduced the concept of intent locking. As explained in Section 16.9, the term *granularity* refers to the size of the objects that can be locked. Since different transactions obviously have different characteristics and different requirements, it is desirable that the system provide a range of different locking granularities (as indeed many systems do). This paper presents an implementation mechanism for such a multi-granularity system, based on intent locking.

We elaborate here on the intent locking protocol, since the explanations given in the body of the chapter were deliberately somewhat simplified. First of all, the lockable object types need not be limited to just relvars and tuples, as we were assuming previously. Second, those lockable object types need not even form a strict hierarchy; the presence of indexes and other access structures means they should be regarded rather as a *directed acyclic graph.* For example, the suppliers-and-parts database might contain both (a stored form of) the parts relvar P and an index, XP say, on the P# attribute of that stored relvar. To get to the tuples of relvar P, we must start with the overall database, and then *either* go straight to the stored relvar and do a sequential scan *or* go to index XP and thence to the required P tuples. So the tuples of P have two "parents" in the graph, P and XP, both of which have the database as a "parent" in turn.

We can now state the protocol in its most general form:

- Acquiring an X lock on a given object implicitly acquires an X lock on all children of that object.
- Acquiring an S or SIX lock on a given object implicitly acquires an S lock on all children of that object.

- Before a transaction can acquire an S or IS lock on a given object, it must first acquire an IS (or stronger) lock on at least one parent of that object.
- Before a transaction can acquire an X, IX, or SIX lock on a given object, it must first acquire an IX (or stronger) lock on all parents of that object.
- Before a transaction can release a lock on a given object, it must first release all locks it holds on all children of that object.

In practice, the protocol does not impose as much run-time overhead as might be thought, because at any given moment the transaction will probably already have most of the locks it needs. For example, an IX lock (say) will probably be acquired on the entire database just once, when the program starts execution. That lock will then be held throughout all transactions executed during the lifetime of the program.

16.11 J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger: "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in G. M. Nijssen (ed.), *Proc. IFIP TC-2 Working Conf. on Modelling in Data Base Management Systems*. Amsterdam, Netherlands: North-Holland/New York, N.Y.: Elsevier Science (1976).

The paper that introduced the concept of isolation levels (under the name *degrees of consistency*[16]).

16.12 Theo Härder and Kurt Rothermel: "Concurrency Control Issues in Nested Transactions," *The VLDB Journal* 2, No. 1 (January 1993).

As noted in the "References and Bibliography" section in Chapter 15, several writers have suggested the idea of *nested transactions*. This paper proposes a set of locking protocols for such transactions.

16.13 J. R. Jordan, J. Banerjee, and R. B. Batman: "Precision Locks," Proc. 1981 ACM SIGMOD Int. Conf. on Management of Data, Ann Arbor, Mich. (April/May 1981).

Precision locking is a tuple-level locking scheme that guarantees that only those tuples that need to be locked (in order to achieve serializability) are actually locked, phantoms included. It is in fact a form of *predicate* locking (see Section 16.8, also reference [16.6]). It works by (a) checking update requests to see whether a tuple to be inserted or deleted satisfies an earlier retrieval request by some concurrent transaction and (b) checking retrieval requests to see whether a tuple that has already been inserted or deleted by some concurrent transaction satisfies the retrieval request in question. Not only is the scheme quite elegant, the authors claim that it actually performs better than conventional techniques (which typically lock too much anyway).

16.14 Tim Kempster, Colin Stirling, and Peter Thanisch: "Diluting ACID," *ACM SIGMOD Record* 28, No. 4 (December 1999).

This paper might better be titled "*Concentrating* ACID"! Among other things, it claims that conventional concurrency control mechanisms preclude certain serializable schedules ("isolation is really a sufficient but not necessary condition for serializability"). Like the SQL standard and reference [16.2], it defines isolation levels in terms of serializability violations; however, its definitions are more refined than, and admit more serializable schedules than, those previous attempts. The paper also demonstrates a flaw (having to do with phantoms) in reference [16.2].

[16] Not the happiest of names! Data is either consistent or it is not. The notion that there might be "degrees" of consistency thus sounds like it might be subject to some dispute. In fact, it seems likely that the theory behind "degrees of consistency" was developed before we had a clear notion of the fundamental importance of data integrity (or "consistency").

16.15 Henry F. Korth and Greg Speegle: "Formal Aspects of Concurrency Control in Long-Duration Transaction Systems Using the NT/PV Model," *ACM TODS 19*, No. 3 (September 1994).

As noted elsewhere (see, e.g., references [15.3], [15.9], [15.16], and [15.17]), serializability is often considered too demanding a condition to impose on certain kinds of transaction processing systems, especially in newer application areas that involve human interaction and hence transactions of long duration. This paper presents a new transaction model called NT/PV ("nested transactions with predicates and views") that addresses such concerns. Among other things, (a) it shows that the standard model of transactions with serializability is a special case, (b) it defines "new and more useful correctness classes," and (c) it claims that the new model provides "an appropriate framework for solving long-duration transaction problems."

16.16 H. T. Kung and John T. Robinson: "On Optimistic Methods for Concurrency Control," *ACM TODS 6*, No. 2 (June 1981).

Locking schemes can be described as *pessimistic*, inasmuch as they make the worst-case assumption that every piece of data accessed by a given transaction might be needed by some concurrent transaction and had therefore better be locked. By contrast, optimistic schemes—also known as *certification* or *validation* schemes—make the opposite assumption that conflicts are likely to be quite rare in practice. Thus, they operate by allowing transactions to run to completion completely unhindered, and then checking at commit time to see whether a conflict did in fact occur. If it did, the offending transaction is simply started again from the beginning. No updates are ever written to the database prior to successful completion of commit processing, so such restarts do not require any updates to be undone.

A subsequent paper [16.7] showed that, under certain assumptions, optimistic methods enjoy certain inherent advantages over traditional locking methods in terms of the expected level of concurrency (i.e., number of simultaneous transactions) they can support, suggesting that optimistic methods might become the technique of choice in systems with large numbers of parallel processors. (By contrast, reference [15.12] claims that optimistic methods in general are actually worse than locking in "hotspot" situations—where a *hotspot* is a data item that is updated very frequently, by many distinct transactions. See the annotation to reference [16.17] for a discussion of a technique that works well on hotspots.)

16.17 Patrick E. O'Neil: "The Escrow Transactional Method," *ACM TODS 11*, No. 4 (December 1986).

Consider the following simple example. Suppose the database contains a data item *TC* representing "total cash on hand," and suppose almost every transaction in the system updates *TC*, decrementing it by some amount (corresponding to some cash withdrawal, say). Then *TC* is an example of a "hotspot," that is, an item in the database that is accessed by a significant percentage of the transactions running in the system. Under traditional locking, a hotspot can very quickly become a bottleneck (to mix metaphors horribly). But using traditional locking on a data item like *TC* is really overkill. If *TC* initially has a value of 10 million dollars, and each individual transaction decrements it (on average) by only 10 dollars, then we could run 1,000,000 such transactions, *and furthermore apply the 1,000,000 corresponding decrements in any order*, before running into trouble. There is thus no need to apply a traditional lock to *TC* at all; instead, all that is necessary is to make sure that the current value is large enough to permit the required decrement, and then do the update. (If the transaction subsequently fails, the amount of the decrement must be added back in again, of course.)

The escrow method applies to situations such as the one just described—that is, situations in which the updates are of a certain special form, instead of being completely arbitrary. The system must provide a special new kind of update statement (e.g., "decrement by *x*, *if and only*

if the current value is greater than *y*"). It can then perform the update by placing the decrement amount *x* "in escrow," taking it out of escrow at end-of-transaction (and committing the change if end-of-transaction is COMMIT, or adding the amount back into the original total if end-of-transaction is ROLLBACK).

The paper describes a number of cases in which the escrow method can be used. One example of a commercial product that supports the technique is the Fast Path version of IMS, from IBM. We remark that the technique might be regarded as a special case of optimistic concurrency control [16.16]. Note, however, that the "special-case" aspect—the provision of the special update statements—is critical.

16.18 Christos Papadimitriou: *The Theory of Database Concurrency Control.* Rockville, Md.: Computer Science Press (1986).

A textbook, with emphasis on formal theory.

16.19 Daniel J. Rosencrantz, Richard E. Stearns, and Philip M. Lewis II: "System Level Concurrency Control for Distributed Database Systems," *ACM TODS 3,* No. 2 (June 1978).

16.20 Kenneth Salem, Hector Garcia-Molina, and Jeannie Shands: "Altruistic Locking," *ACM TODS 19,* No. 1 (March 1994).

Proposes an extension to two-phase locking according to which a transaction *A* that has finished with some locked piece of data but cannot unlock it (because of the two-phase locking protocol) can nevertheless "donate" the data back to the system, thereby allowing some other transaction *B* to acquire a lock on it. *B* is then said to be "in the wake of" *A*. Protocols are defined to prevent, for example, a transaction from seeing any updates by transactions in its wake. Altruistic locking (the term derives from the fact that "donating" data benefits other transactions, not the donor transaction) is shown to provide more concurrency than conventional two-phase locking, especially when some of the transactions are of long duration.

16.21 Abraham Silberschatz, Henry F. Korth, and S. Sudarshan: *Database System Concepts* (4th ed.). New York, N.Y.: McGraw-Hill (2002).

This general textbook on database management includes a strong treatment of transaction management issues (recovery as well as concurrency).

16.22 Robert H. Thomas: "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM TODS 4,* No. 2 (June 1979).

See the annotation to reference [16.3].

16.23 Alexander Thomasian: "Concurrency Control: Methods, Performance, and Analysis," *ACM Comp. Surv. 30,* No. 1 (March 1998).

A detailed investigation into the performance of a wide variety of concurrency control algorithms.