11.1 BASIC CONCEPTS

A *transaction* is a single execution of a program. This program may be a simple query expressed in one of the query languages of Chapter 6 or an elaborate host language program with embedded calls to a query language. Several independent executions of the same program may be in progress simultaneously; each is a transaction.

Items

We imagine that the database is partitioned into *items*, which are portions of the database that can be *locked*. That is, by locking an item, a transaction can prevent other transactions from accessing the item, until the transaction holding the lock unlocks the item. A part of a DBMS called the *lock manager* assigns and records locks, as well as arbitrating among two or more requests for a lock on the same item.

The nature and size of items are for the system designer to choose. In the relational model of data, for example, we could choose large items, like relations, or small items like individual tuples or even components of tuples. We could pick an intermediate size for items; for example, items could be collections of 100 tuples from some relation. In the network model, an item could be the collection of all records of a single type, or what the DBTG proposal terms a set occurrence, for example.

Choosing large items cuts down on the system overhead due to maintaining locks, since we need less space to store the locks, and we save time because fewer actions regarding locks need to be taken. However, choosing small items allows many transactions to operate in parallel, since transactions are then less likely to want locks on the same items.

At the risk of oversimplifying the conclusions of a number of analyses mentioned in the bibliographic notes, let us suggest that the proper choice for the size of an item is such that the average transaction accesses a few items. Thus if the typical transaction (in a relational system) reads or modifies one tuple, which it finds via an index, it would be appropriate to treat tuples as items. If the typical transaction takes a join of two or more relations, and thereby requires access to all the tuples of these relations, then we would be better off treating whole relations as items.

In what follows, we shall assume that when part of an item is modified, the whole item is modified and receives a value that is unique and unequal to the value that could be obtained by any other modification. We make this assumption not only to simplify the modeling of transactions. In practice, it requires too much work on the part of the system to deduce facts such as that the result of one modification of an item gives that item the same value as it had after some previous modification. Furthermore, if the system is to remember

A in database	5	5	5	5	6	6
T_1:	READ A		$A{:=}A+1$			WRITE A
T_2:		READ A		$A{:=}A+1$	WRITE A	
A in T_1's workspace	5	5	6	6	6	6
A in T_2's workspace		5	5	6	6	

Fig. 11.1. Transactions exhibiting a need to lock item A.

whether part of an item remains unchanged after the item is modified, it may as well divide the item into several smaller items. A consequence of our assumption of the indivisibility of items is that we shall not go wrong if we view items as simple variables as used in common programming languages.

Locks

Example 11.1: To see the need for locking items, let us consider two transactions T_1 and T_2. Each accesses an item A, which we assume has an integer value, and adds one to A. The two transactions are executions of the program P defined as

$$P : \text{READ } A;\ A := A + 1;\ \text{WRITE } A$$

The value of A exists in the database. P reads A into its workspace, adds one to the value in the workspace, and writes the result into the database. In Fig. 11.1 we see the two transactions executing in an interleaved fashion†, and we record the value of A as it appears in the database at each step.

We notice that although two transactions have each added 1 to A, the value of A has only increased by 1. This is a serious problem if A represents seats sold on an airplane flight, for example. □

One solution to the problem represented by Example 11.1 is to provide a lock on A. Before reading A, a transaction T must lock A, which prevents another transaction from accessing A until T is finished with A. Furthermore, the need for T to set a lock on A prevents T from accessing A if some other transaction is already using A. T must wait until the other transaction unlocks A, which it should do only after finishing with A.

Let us now consider programs that interact with the database not only by reading and writing items but by locking and unlocking them. We assume

† Note that we do not assume necessarily that two similar steps take the same time, so it is possible that T_2 finishes before T_1, even though both transactions execute the same steps. However, the point of the example is not lost if T_1 writes before T_2.

that a lock must be placed on an item before reading or writing it, and that the operation of locking acts as a synchronization primitive. That is, if a transaction tries to lock an already locked item, it waits until the lock is released by an unlock command, which is executed by the transaction holding the lock. We assume that each program is written to unlock any item it locks, eventually. A schedule of the elementary steps of two or more transactions, such that the above rules regarding locks are obeyed, is termed *legal*.

Example 11.2: The program P of Example 11.1 could be written with locks as

$$P: \text{LOCK } A\text{; READ } A\text{; } A{:=}A+1\text{; WRITE } A\text{; UNLOCK } A$$

Suppose again that T_1 and T_2 are two executions of P. If T_1 begins first, it requests a lock on A. Assuming no other transaction has locked A, the system grants this lock. Now T_1, and only T_1 can access A. If T_2 begins before T_1 finishes, then when T_2 tries to execute LOCK A, the system causes T_2 to wait. Only when T_1 executes UNLOCK A will the system allow T_2 to proceed. As a result, the anomaly indicated in Example 11.1 cannot occur; either T_1 or T_2 executes completely before the other starts, and their combined effect is to add 2 to A. □

Livelock and Deadlock

We have postulated a part of a DBMS that grants and enforces locks on items. Such a system cannot behave capriciously, or certain undesirable phenomena occur. As an instance, we assumed in Example 11.2 that when T_1 released its lock on A, the lock was granted to T_2. What if while T_2 was waiting, a transaction T_3 also requested a lock on A, and T_3 was granted the lock before T_2. Then while T_3 had the lock on A, T_4 requested a lock on A, which was granted after T_3 unlocked A, and so on. Evidently, it is possible that T_2 could wait forever, while some other transaction always had a lock on A, even though there are an unlimited number of times at which T_2 might have been given a chance to lock A.

Such a condition is called *livelock*. It is a problem that occurs potentially in any environment where processes execute concurrently. A variety of solutions have been proposed by designers of operating systems, and we shall not discuss the subject here, as it does not pertain solely to database systems. A simple way to avoid livelock is for the system granting locks to record all requests that are not granted immediately, and when an item A is unlocked, grant a lock on A to the transaction that requested it first, among all those waiting to lock A. This first-come-first-served strategy eliminates livelocks,† and we shall assume from here on that livelock is not a problem.

There is a more serious problem of concurrent processing that can occur if

† Although it may cause "deadlock," to be discussed next.

we are not careful. This problem, called "deadlock," can best be illustrated by an example.

Example 11.3: Suppose we have two transactions T_1 and T_2 whose significant actions, as far as concurrent processing is concerned are:

T_1: LOCK A LOCK B UNLOCK A UNLOCK B
T_2: LOCK B LOCK A UNLOCK B UNLOCK A

Presumably T_1 and T_2 do something with A and B, but this is not important here. Suppose T_1 and T_2 begin execution at about the same time. T_1 requests and is granted a lock on A, and T_2 requests and is granted a lock on B. Then T_1 requests a lock on B, and is forced to wait because T_2 has a lock on that item. Similarly, T_2 requests a lock on A and must wait for T_1 to unlock A. Thus neither transaction can proceed; each is waiting for the other to unlock a needed item, so both T_1 and T_2 wait forever. □

A situation in which each member of a set S of two or more transactions is waiting to lock an item currently locked by some other transaction in the set S is called a *deadlock*. Since each transaction in S is waiting, it cannot unlock the item some other transaction in S needs to proceed, so all wait forever. Like livelock, the prevention of deadlock is a subject much studied in the literature of operating systems and concurrent processing in general. Among the approaches to a solution are the following.

1. Require each transaction to request all its locks at once, and let the system grant them all, if possible, or grant none and make the process wait, if one or more are held by another transaction. Notice how this rule would have prevented the deadlock in Example 11.3. The system would grant locks on both A and B to T_1 if it requested first; T_1 would complete, and then T_2 could have both locks.
2. Assign an arbitrary linear ordering to the items, and require all transactions to request locks in this order.

The second approach also prevents deadlock. In Example 11.3, suppose A precedes B in the ordering (there could be other items between A and B in the ordering). Then T_2 would request a lock for A before B and would find A already locked by T_1. T_2 would not yet get to lock B, so a lock on B would be available to T_1 when requested. T_1 would complete, whereupon the locks on A and B would be released. T_2 could then proceed. To see that no deadlocks can occur in general, suppose we have a set S of deadlocked transactions, and each transaction R_i in S is waiting for some other transaction in S to unlock an item A_i. We may assume that each R_j in S holds at least one of the A_i's, else we could remove R_j from S and still have a deadlocked set. Let A_k be the first item among the A_i's in the assumed linear order. Then R_k, waiting for A_k, cannot hold any of the A_i's, which is a contradiction.

Another approach to handling deadlocks is to do nothing to prevent them.

Rather, periodically examine the lock requests and see if there is a deadlock. The algorithm of drawing a *waits-for graph*, whose nodes are transactions and whose arcs $T_1 \rightarrow T_2$ signify that transaction T_1 is waiting to lock an item on which T_2 holds the lock, makes this test easy; every cycle indicates a deadlock, and if there are no cycles, neither are there any deadlocks. If a deadlock is discovered, at least one of the deadlocked transactions must be restarted, and its effects on the database must be cancelled. This process of restart can be complicated if we are not careful about the way transactions write into the database before they complete. The subject is taken up in Section 11.6.

In the future, we shall assume that neither livelocks nor deadlocks will occur when executing transactions.

Serializability

Now we come to a concurrency issue of concern primarily to database system designers, rather than designers of general concurrent systems. By way of introduction, let us review Example 11.1, where two transactions executing a program P each added 1 to A, yet A only increased by 1. Intuitively, we feel this situation is wrong, yet perhaps these transactions did exactly what the writer of P wanted. However, it is doubtful that the programmer had this behavior in mind, because if we run first T_1 and then T_2, we get a different result; 2 is added to A. Since it is always possible that transactions will execute one at a time (serially), it is reasonable to assume that the normal, or intended, result of a transaction is the result we obtain when we execute it with no other transactions executing concurrently. Thus, we shall assume from here on that the concurrent execution of several transactions is correct if and only if its effect is the same as that obtained by running the same transactions serially in some order.

Let us define a *schedule* for a set of transactions to be an order in which the elementary steps of the transactions (lock, read, and so on) are done. The steps of any given transaction must, naturally, appear in the schedule in the same order that they occur in the program of which the transaction is an execution. A schedule is *serial* if all the steps of each transaction occur consecutively. A schedule is *serializable* if its effect is equivalent to that of some serial schedule.

Example 11.4: Let us consider the following two transactions, which might be part of a bookkeeping operation that transfers funds from one account to another.

T_1: READ A; $A := A - 10$; WRITE A; READ B; $B := B + 10$; WRITE B
T_2: READ B; $B := B - 20$; WRITE B; READ C; $C := C + 20$; WRITE C

Clearly, any serial schedule has the property that the sum $A+B+C$ is preserved. In Fig. 11.2(a) we see a serial schedule, and in Fig. 11.2(b) is a serializable, but not serial, schedule. Figure 11.2(c) shows a nonserializable schedule. Note that