

Chapter 13(2)

Concurrency

The topics of concurrency and recovery are pretty much intertwined, being part of the general topic of transaction management. The term concurrency refers to the fact the DBMS typically allows many transactions to access the same database at the same time, and thus, some kind of concurrency control mechanism is needed to ensure that concurrent transactions do not interfere with each other.

We will see why such control mechanism is needed and go through several options for such mechanisms.

Concurrency problems

When several transactions execute at the same time, they might interfere with each other. Particularly, a transaction might be correct in itself, but nevertheless produces the wrong answers if some other transaction interferes with it in some way.

In general, there are three problems any concurrency control mechanism has to deal with: the last update problem, the uncommitted dependency problem, and the inconsistent analysis problem.

The last update problem

Consider the following situation:

Trans. A	time	Trans. B
-	↓	-
Retrieve t	t_1	-
-	↓	-
-	t_2	Retrieve t
-	↓	-
Update t	t_3	-
-	↓	-
-	t_4	Update t
-	↓	-

The essential situation is that transaction A updates the tuple t at time t_3 ; and the transaction B updates the same tuple (on the basis of the values seen at time t_2 , which are the same as those seen at time t_1 .) at time t_4 . Thus, transaction A's update at time t_3 is lost at time t_4 , since transaction B overwrite it w/o checking it.

A possible scenario

John and Jane jointly own a bank account. They happen to withdraw money at roughly the same time during the holiday season.

John finds out there are %50 in the account, and decides to take out \$40. John's withdraw goes through. On the other hand, Jane also finds out the same figure and decides to take out \$30. Unfortunately, Jane's withdraw could not go through.

Question: Why?

Answer: Typical of the last undate problem.

Uncommitted dependency

This problem arises if transaction A is allowed to retrieve, or update, a tuple that has been updated, but not yet committed, by transaction B. Since, before commitment, there is always a possibility that it will be rolled back; thus, transaction A might see some data that no longer exists.

For example,

Trans. A	time	Trans. B
-	↓	-
-	t_1	Update t
-	↓	-
Retrieve t	t_2	-
-	↓	-
-	t_3	Rollback
-	↓	-

A possible scenario

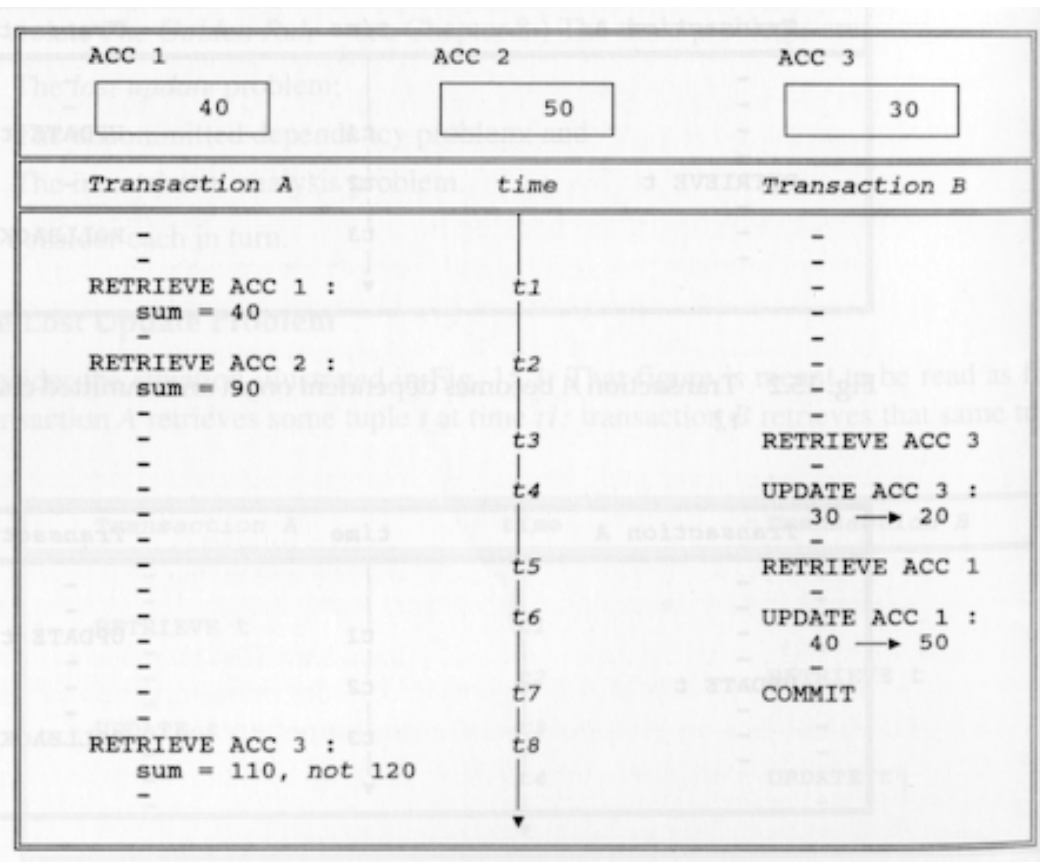
John got a check of \$1,000 from Mike and deposited into his bank account. John then took it out for a flat panel TV. Guess what? Mike's check is not valid.

Question: Why every bank has to wait for a few days before the check becomes available?

Answer: Because of this uncommitted dependency problem.

Inconsistent analysis

Below shows the actions of trans. A and B operating on account (ACC) tuples: Trans. A is adding up account balances, while trans. B is transferring an amount 10 from account 3 to account 1. Since trans. A is not aware of what trans. B is doing, it reaches a wrong result of 110, instead of 120 at the end.



Locking

The above problems can be solved by using the concurrency control technique of *locking*.

The basic idea is simple: when a transaction wants to make sure that an object will not change in any way while being worked on, it *asks a lock* to be placed on that object, to prevent other transactions from changing it.

The first transaction is thus able to carry out the processing, knowing that the object in question will remain in a stable state for as long as that transaction wishes it to.

More specifically...

1. We assume the system only supports two kinds of locks, *write locks* (X locks) and *read locks* (S locks), and tuples are the only lockable objects. Thus, transactions can work on a table at the same time, but not on a tuple at the same time.
2. If trans. A holds an X lock on a tuple r , then a request from another trans. B to put a lock of either type on t will be denied.
3. If trans. A holds an S lock on t , then a) A request from some another trans. B to put an X lock on t will be denied, but, b) a request from B to put an S lock on t will be granted.

Question: Why is the policy?

Answer: When you are writing something, you don't want to share with the others; but multiple people can read the same thing. (Think about how a web page is accessed.)

Thus, a tuple can have at most one X lock, but, it can have several S locks, as indicated by the following *compatibility matrix*.

	<i>X</i>	<i>S</i>	—
<i>X</i>	N	N	Y
<i>S</i>	N	Y	Y
—	Y	Y	Y

Where “—” means there is no lock placed.

Locking protocol

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 that tuple, e.g., when it is in a RETRIEVE-UPDATE sequence, then it must promote that S lock to X level, first.
3. If a lock requests from trans. B is denied because it conflicts with a lock already held by trans. A, B will wait, e.g., in a FIFO style, until the lock placed on A is released.
- 4 X locks are held until end-of-transaction, i.e., COMMIT or ROLLBACK. S locks are normally held until that time also.

Problems revisited

Below is the modified situation on the last update problem. Notice that we are now getting into a *deadlock*.

Trans. A	time	Trans. B
-	↓	-
Retrieve t (acquires S lock)	t_1	-
-	↓	-
-	↓	-
-	t_2	Retrieve t (acquire S lock)
-	↓	-
Update t (request X lock)	t_3	-
wait	↓	-
wait	↓	Update t (request X lock)
wait	t_4	wait
wait	↓	wait
wait	↓	wait
wait	↓	wait

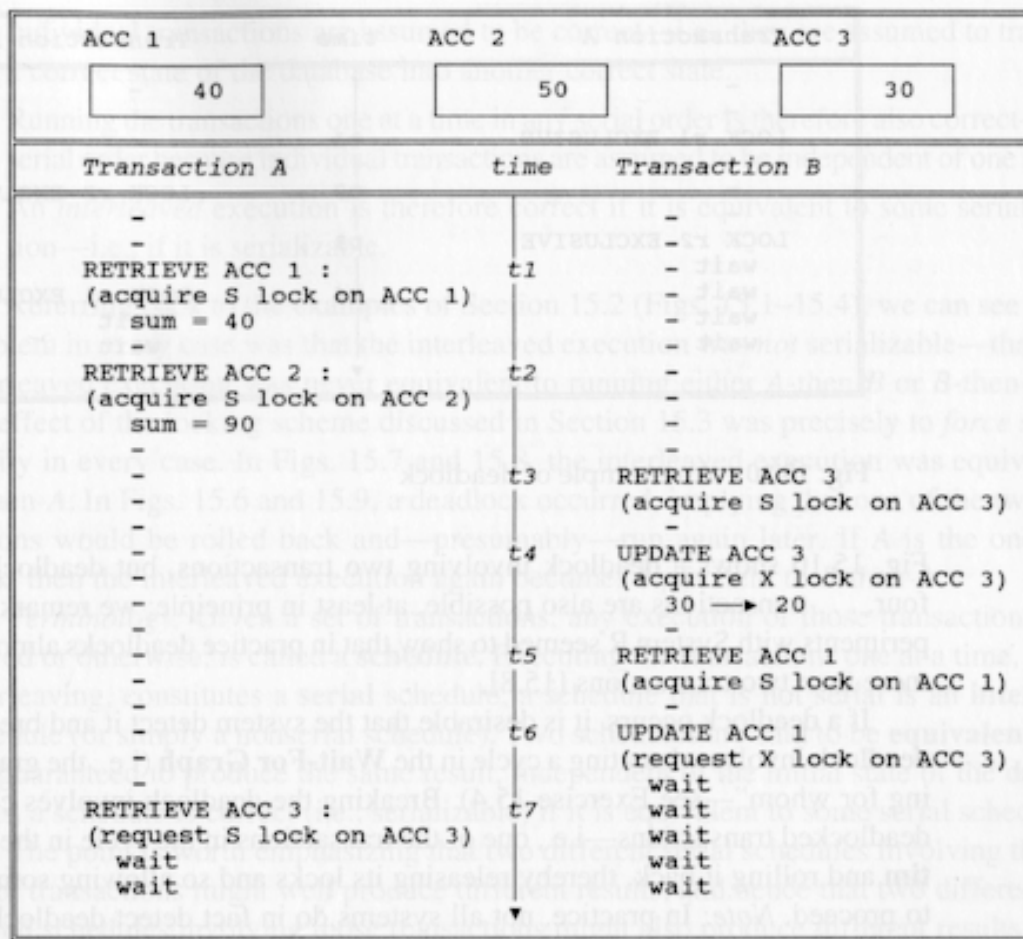
Solving the problem of uncommitted dependency

Below shows the revised situation of the uncommitted problem, when locking mechanism is in place.

Trans. A	time	Trans. B
-	↓	-
-	t_1	Update t (acquires X lock)
-	↓	-
Retrieve t (request S lock) wait	t_2	-
wait	↓	-
resume: Retrieve t (acquires S lock)	t_3	COMMIT/ROLLBACK (release X lock)
-	t_4	
	↓	

Solving the problem of inconsistency analysis

Below shows how to solve this problem by applying the locking protocol. But, again, the solution forces a deadlock.



Deadlock

The locking mechanism can introduce problems of its own, mainly the problem of deadlock. Below shows a more general version of the p problem: r_1 and r_2 are intended to represent any lockable objects, not just tuples, and the “LOCK...EXCLUSIVE” statement represents any operations that acquire X locks.

Trans. A	time	Trans. B
-	↓	-
LOCK r_1 EXCLUSIVE	t_1	-
-	↓	-
-	t_2	LOCK r_2 EXCLUSIVE
-	↓	-
LOCK r_2 EXCLUSIVE	t_3	-
wait	↓	
wait	t_4	LOCK r_1 EXCLUSIVE
wait	↓	wait

Deadlock is a situation when two or more transactions are in a simultaneous wait, each of them waiting for the other to release a lock before it can proceed.

When a deadlock occurs, the system should be able to detect it, and then break it. *Detecting the deadlock* involves detecting a cycle in the *Wait-For-Graph*,, i.e., a graph showing who is waiting for whom. *Breaking the deadlock* involves choosing one of the deadlocked transactions as the *victim*, and rolling it back, thus releasing its locks and allowing some other transaction to proceed. In practice, a system might use a timeout mechanism and simply assume that a transaction that has done no work for some period of time must be deadlocked.

Also, since a victim is rolled back, some systems will automatically restart it from the very beginning; while other systems simply send an exception message back to the application, and let the program decide what to do.

We will study deadlock in much more details in Operating System.