CS352 Lecture - Concurrency

Last revised March 4, 2019

Objectives:

1. To introduce locking as a means of preserving the serializability of concurrent schedules.
2. To briefly introduce other approaches to this problem (timestamps, validation, multi-version schemes)
3. To introduce other important issues (phenomena related to insert/delete; weak levels of consistency; concurrency issues with index structures)

I. **Introduction**

A. We have previously considered the notion of a transaction, and noted that, if properly formed transactions are executed serially, with one completing before the next starts, we are guaranteed to preserve consistency in the database. In particular, a SERIAL SCHEDULE composed of consistent transactions always preserves database consistency.

B. There are many practical reasons for not wanting to be restricted to serial execution, thus allowing two or more transactions to be processed in parallel. (The individual steps are still done serially, but steps of one transaction can be interwoven with steps of another.)

1. Better use of system resources through overlapping IO and computational activities.

   a) Being able to perform useful computation on the CPU while waiting for a disk access to complete.

   b) If the system has multiple disks, then being able to do multiple disk accesses in parallel, given that better than 90% of the time consumed in a disk access is spent on head movement and rotational latency.

2. Support for access to the database by multiple users simultaneously.

3. If a transaction is interactive, and there is some delay while a human responds to some aspect of it, we do not want the system to be totally idle.

4. Not making the database totally inaccessible while processing a long-running transaction (e.g. posting interest to all savings accounts.)

C. We have seen that we can permit concurrency and still preserve consistency provided that we schedule overlapping transactions in a way that is SERIALIZABLE - i.e. equivalent to a serial schedule, and if we ensure that any schedule we produce is RECOVERABLE - i.e. does not allow a transaction to fully commit until any transaction whose data it has read has already fully committed.

D. In this lecture, we look at various strategies that might be used to ensure serializability. (Some also deal with recoverability).

II. Ensuring serializability by the use of locks

A. Thus far, we have discussed methods for testing a schedule to see if it is serializable. Since a concurrent system seldom "knows" ahead of time what sequence of transactions it will receive, a more useful goal is the development of a set of rules that ensures that any schedule that develops over time is serializable. One way to do this is by using a LOCKING PROTOCOL of some sort.

1. We extend the DBMS primitives beyond the basic read and write to include primitives lock and unlock. Basically, a lock operation excludes other transactions from certain kinds of access to a data item, and an unlock operation releases that hold.

2. These primitives do not appear as explicit DML statements, however. Instead, certain kinds of database operations result in appropriate locks being acquired as part of their implementation by the DBMS.

Example: a SQL UPDATE involving a computation on some column will lock the column value at least from before it reads it to after it writes the updated value (and perhaps longer).

Note: DBMS locking differs from the kind of locking we have met in Java, where the programmer is responsible to explicitly request it by using the keyword `synchronized`. DBMS locking is managed automatically as operations that require locks are executed.

3. In particular, locking is tied in with the notion of transactions, in the sense that sometimes locks acquired during a transaction will need to be held until the transaction is fully committed, as we shall see.

4. In the discussion below, we will show locking / unlocking operations explicitly. But this is only for pedagogical purposes - in reality, locking is done by the DBMS when an operation that requires a lock is started, and unlocking is done by the DBMS at the end of a transaction (or sometimes earlier) (There is no such thing as a LOCK or UNLOCK statement in SQL!)

B. Granularity of locks

1. One important issue in any locking system is the GRANULARITY of the locks - i.e. what size objects can be locked.

   a) The coarsest granularity is locking at the table level - any transaction wishing to access the table must lock the entire table. This is seldom an adequate solution since it severely limits potential concurrency.

   b) More desirable is locking at the row or even column level. Either an entire row can be locked, or possibly just some columns of a row. (Row locking is usually fine enough, though, and involves much less overhead.)

c) However, since data is usually read and written in physical blocks or pages, not by rows, locking is often implemented at the disk block or page level. Thus, a transaction wishing to lock a particular row will, in fact, end up locking the entire page on which it is stored. (This granularity works well with crash control schemes based on shadow paging which we will consider under the topic of crash recovery). A transaction that wishes to modify a page may work with its current copy, but must also lock the shadow copy until it commits to prevent lost updates from a second process also making a copy of the page.

2. Some systems allow multiple lock granularities - e.g. the possibility of locking an entire table, or just one part (row or database page). This can be useful if, for example, a transaction is performing some update operation on most or all of the rows in a table - in which case locking the entire table is more efficient than locking the rows one by one (especially if consistency requires that all the rows remain locked until the transaction completes.)

C. Locking protocols are generally based on two kinds of locks:

1. A SHARED lock is used when a transaction wants to read an item, but does not want to change it. It allows other transactions to also obtain shared locks, so they can read the item, provided they too do not change it. The transaction obtaining a shared lock on an item must hold it as long as it wants to be sure that the item is not altered.

   We will use the notation lock-s(item)

   Example: A transaction that simply prints a user's current balance on some one account need not hold a lock on that balance after it has read it.

   We will show this as:

   ```
   lock-s(balance)
   read(balance)
   unlock(balance)
   ```

Example: A transaction to print the total balance of all accounts held by a given user must hold a shared lock on each individual balance until it has read all of them. This will ensure that any simultaneous transfer of money between accounts does not result in an erroneous total amount being displayed.

We will show this as:

```
lock-s(balance1)
read(balance1)
lock-s(balance2)
read(balance2)
....
unlock(balance1)
unlock(balance2)
...
```

2. An EXCLUSIVE lock is used when a transaction wants to write an item (but also will allow it to read the item).

   a) A transaction wanting to obtain an exclusive lock on a given item will be forced to wait until any other transaction holding any kind of lock on the item releases it.

   b) While the exclusive lock is held, no other transaction can obtain any kind of lock on the item.

   c) If the desired operation is a read-modify-write on an item, then the transaction must obtain the exclusive lock before doing the read; or must obtain a shared lock before the read which is then upgraded to exclusive before the write. (That is, it must hold some kind of lock on the item at all times between the read and the write.)

   d) An exclusive lock must remain in force until the transaction either commits or is rolled back, to prevent lost or fallacious updates.

We will represent the obtaining of an exclusive lock by lock-x - e.g. if a transaction gives 5% interest to an account, the transaction might be represented as:

```
lock-x(balance)
read(balance)
[ add interest amount ]
write(balance)
unlock(balance)
```

3. To increase concurrency, some systems allow for the upgrading of locks. The book, for example, suggested the possibility of allowing a read-modify-write transaction to obtain a shared lock before its read and to upgrade it to an exclusive lock just before doing its write, instead of holding an exclusive lock the whole time.

   Example - the above case (though here this probably wouldn't be beneficial):

```
lock-s(balance)
read(balance)
[ add interest amount ]
upgrade(balance)
write(balance)
unlock(balance)
```

D. One problem that can arise with the use of locking protocols is DEADLOCK.

1. Two transactions T1 and T2 are deadlocked if T1 holds a lock on a resource R1 and needs to obtain a lock on another resource R2 before it can release the lock on R1, and if T2 similarly holds an incompatible lock on R2 and needs an incompatible lock on R1.

   Example: Consider two transactions: one to transfer $50.00 from a customer's checking account to his savings account, and another to print the total balance in his two accounts.

One possible schedule looks like this:

Transfer transaction T1	Inquiry transaction T2
lock-x(checking balance)	
read(checking balance)	
calculate new balance = old - 50	
write(checking balance)	
	lock-s(savings balance)
	read(savings balance)
	lock-s(checking balance) -- MUST WAIT
lock-x(savings balance) - MUST WAIT	

Under this schedule, the the inquiry transaction will only execute as far as the lock-s(checking balance) before being forced to wait and the transfer transaction will only execute as far as the lock-x(savings balance) before being forced to wait. Now neither transaction can proceed, and thus cannot unlock the resource the other needs.

In this case, the deadlock caused by locking actually prevents an unserializable schedule from being created. If T2 were able to proceed to read the checking balance and then print the sum, the result would be wrong since the checking balance has already been reduced but the savings balance has not yet been increased.

2. In the operating systems portion of the software systems course, we discuss several ways of dealing with deadlock:

   a) Deadlock prevention - design a scheme in such a way that deadlock can never occur. (This is not always possible)

   b) Deadlock avoidance - before any lock is granted, check to see if granting it might lead to a situation in which deadlock could occur. If so, delay it. (For example, in the above scenario, the system could force the inquiry transaction to wait when it requests the lock on savings balance. This would allow the transfer transaction to complete and release its locks; then the inquiry transaction could run.) Deadlock avoidance also generally requires some advance knowledge as to how a transaction will behave.

c) Deadlock detection - we allow deadlock to occur; but when it does, we deal with it by choosing one of the deadlocked transactions, rolling it back, and then restarting it after first allowing the other transaction to proceed past the point of the deadlock.

3. DBMS's often use the third approach of deadlock detection and recovery by rollback, since deadlock prevention may limit concurrency too much, and deadlock avoidance requires advance knowledge of a transaction's behavior that may not be available. Deadlock detection and recovery is generally NOT regarded as a good solution for operating systems, because the rollback costs may be too large; but it is acceptable for DBMS's because:

   a) Most transactions are relatively small - thus, the cost of a rollback is relatively small.

   b) DBMS's deal with large numbers of resources (e.g. each row in the database); thus, the probability that two transactions will deadlock over a given resource is relatively small and deadlocks will be relatively rare.

   c) DBMS's have to be prepared to deal with transaction rollback in any case.

   Example: A DBMS might well allow the deadlock described above to occur - at which point the inquiry transaction could be rolled back and restarted after the transfer transaction completes

E. Simply insisting that a transaction obtain an appropriate kind of lock on an item before accessing it, however, does not guarantee serializability.

Example: Consider the parallel execution of a transaction to transfer $50 from a checking account to a savings account (owned by the same person) and a transaction to print the total balance in the person's accounts. Obviously, we don't care whether the total balance is computed on the the basis of the balances before the transfer or after the transfer, since the total balance is the same in either case.

1. However, the following is a non-serializable schedule that yields the wrong result, even though the locking rules we have discussed thus far are used:|

Transfer transaction T1	Total balance inquiry T2
	lock-s(savings balance)
	read savings balance (S)
lock-x(checking balance)	
read checking balance (C)	
write checking balance (C-50)	
unlock(checking balance)	
	lock-s(checking balance)
	read checking balance (C-50)
	unlock(savings balance)
	unlock(checking balance)
lock-x(savings balance)	
read savings balance (S)	
write savings balance (S+50)	
unlock(savings balance)	

Note that each transaction obtains an appropriate lock before it does a read or write operation, and the inquiry transaction retains its lock on the savings balance until after it has read the checking balance, too, in an attempt to ensure that an update does not come along and alter one of the values and so mess the result up. However, this does not protect it from error, since the transfer transaction has left the sum of the balances momentarily incorrect, but with no locks held.

2. To prevent such inconsistencies, some protocol must be used to govern the ORDER in which a transaction acquires and releases locks.

3. One strategy that is widely used is called the TWO-PHASE locking protocol. In this protocol, we require that a transaction cannot acquire any new locks after it has once released a lock that it has held. Thus, a transaction will execute in two phases:

   a) In the GROWING phase, it may acquire new locks, but may not release any. (Converting a lock from a lower level to a higher level - eg from shared to exclusive - is also allowed in this phase.)

   b) In the SHRINKING phase, it may release locks it holds, but not acquire new ones. The shrinking phase begins, then, with the first unlock operation done by the transaction.

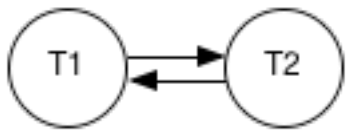
      Example: The inquiry transaction above is two-phase, but the transfer transaction is not. Notice how a non two-phase transaction is able to destroy the consistency of another transaction, even though it is two-phase. To ensure consistency, EVERYBODY must obey the protocol

   c) Why does the two phase protocol work?

      (1) Observe that, for any schedule S, we can construct a directed graph using the precedes relationship: there is an edge from Ti to Tj iff in S Ti acquires a lock on some resource A BEFORE Tj acquires an incompatible lock on the same resource. We say that Ti PRECEDES Tj in S (written Ti -> Tj).

         If this graph is acyclic, then the schedule is serializable.

Example: The precedes relationship for the above schedule:



There is an edge from T1 to T2 since T1 locks
the checking balance before T2 does.
There is an edge from T2 to T1 since T2

This graph is not acyclic, and the schedule is not serializable.

(2) The two-phase protocol guarantees serializability because the precedes graph for any set of two-phase transactions is acyclic. To see this, consider two transactions T1 and T2 that must both hold incompatible locks on the same item or items. The only way the precedes graph could contain a cycle involving these two transactions is if T1 locks some item T2 needs before T2 does, and T2 locks some item T1 needs before T1 does. But in this case, T1 and T2 will deadlock, since neither can release the lock it holds while it still needs to acquire another. One will then be rolled back, and the cycle in the precedes graph will be destroyed.

d) The two-phase protocol can be extended to also ensure recoverabilty, as follows:

(1) The STRICT two-phase protocol requires that all exclusive locks acquired by a transaction be held until the transaction commits.

(2) The RIGOROUS two-phase protocol requires that all locks (both shared and exclusive) acquired by a transaction be held until the transaction commits.

Both guarantee cascadeless recoverability, because no transaction can ever read data written by an uncommitted transaction.

e) The latter two variants of the two-phase protocol are widely used, though they must be accompanied by some scheme for deadlock detection and recovery, since two-phase locking can lead to deadlock.

4. While the two-phase locking protocol is widely used, it is not the only possibility. The book discusses an alternative tree-based approach which we will not pursue here.. (This has the additional advantage of preventing deadlock, but it allows less concurrency.)

III. **Other Ways of Ensuring Serializability**

A. Locking is a widely-used way of ensuring serializability. But it is not the only possibility.

B. Another approach is based on TIMESTAMPs. As each transaction enters the system, it is given a unique timestamp which is some sort of clock reading or serial number that is different for each transaction.

1. We refer to the timestamp of some transaction T as TS(T).

2. We associate with each data item (e.g. each row in a table or each database page) two timestamps - R-timestamp, which is the timestamp of the last transaction that read it, and W-timestamp, which is the timestamp of the last transaction that wrote it.

3. We observe the following rules:

a) A transaction may not read a data item if the W-timestamp of that item is greater than the transaction's timestamp. If it attempts to do so, it must be rolled back.

b) A transaction may not write a data item if either the R-timestamp or the W-timestamp of that item is greater than the transaction's timestamp. If it attempts to do so, it must be rolled back.

There is a variant of this known as Thomas's write rule (discussed in the book) which instead ignores a useless write in the case where only the W-timestamp of the item is greater than the transaction's timestamp - in which case the write is just ignored instead of rolling the transaction back.

c) If a transaction is rolled back, it is restarted with a fresh timestamp (which is necessarily larger than the timestamps of the item is failed to access) and it is started over from the beginning.

4. The protocol operates to ensure that the actual schedule is equivalent to a serial schedule in which T_i completes before T_j starts iff $TS(T_i) < TS(T_j)$.

C. Both locking and timestamp protocols operate by STOPPING a transaction from doing a read or write that would result in a non-serializable schedule. Alternately, we might allow a transaction to read and write as it needs to, without interference; but then, before it commits, we check to see if the outcome is serializable. This can be very advantageous if the majority of transactions are read-only ones that do not interfere with one another. Such an approach is known as VALIDATION.

D. Yet another approaches ensures serializability by a Multi-Version scheme.

1. Consider the following excerpt from a non-serializable schedule:

T1	T2
	write(A)
read(A)	
write(B)	
	read(B)

As it stands, this is a non-serializable schedule. The write(A) in T2 followed by the read(A) in T1 requires that, in any equivalent serial schedule, T2 must precede T1; but the write(B) in T1 followed by read(B) in T2 requires the opposite.

2. Suppose, however, that when T1 writes B we retain the OLD value of B along with the new value. (We keep two versions of B in the database.) Then, when T2 reads B, we can give it the OLD value, producing a schedule equivalent to the serial schedule T2; T1. The effect is the same as if the read(B) in T2 preceded the write(B) in T1, though the opposite was actually the case.

3. This approach to concurrency control is called a MULTIVERSION SCHEME. It uses timestamping in two ways:

   a) Each transaction has a timestamp, as in previous schemes, recording its START time. The scheme will ensure that any schedule is equivalent to a serial schedule in which T_i precedes T_j if $TS(T_i) < TS(T_j)$.

   b) Each VERSION of each item has two timestamps, as in previous schemes:

      (1)W-Timestamp(Q) is the timestamp of the transaction that wrote it.

      (2)R-Timestamp(Q) is the highest timestamp of any transaction that has read this version of item Q.

   c) When a transaction T does a read for some item Q, the relevant version of the item is the one with the greatest W-Timestamp such that W-Timestamp(Q) <= TS(T).

   d) When a transaction T does a write for some item Q, but a "later" transaction has already read Q, it must be rolled back and restarted. (In fact, cascading rollback is now possible).

IV. **Further Issues**

A. Concurrency Issues Arising from Delete and Insert

1. Thus far, we have dealt with concurrency-related issues arising from the operations of read() and write() on some item in the database.

   a) The SQL UPDATE statement always does a write, and often does a read as well (e.g. SET SALARY = SALARY*1.05). The SELECT statement does a read, which may be preparatory to a write either to the same item or another item. The same is true for corresponding operations using other data models.

   b) What about the DELETE and INSERT operations?

2. The book shows that a delete operation is similar in principle to a write operation in terms of the table row being deleted.

3. The book shows that an insert operation is similar in principle to a write operation in terms of the table row being inserted

4. However, another issue that arises with delete or insert is the phenomenon of "phantom rows".

   a) Consider a query like the following, which - in general - looks at multiple rows:

   ```
   select count(*) from borrower
    where last_name = 'Aardvark'
   ```

   At the implementation level, the DBMS uses some sort of iterator to iterate over the relevant rows. (This is similar in principle to the cursor of embedded SQL, but internal to the DBMS)

b) Now suppose a concurrent transaction does an insert (or delete) of a row with name = 'Aardvark', or updates an existing row to change the name to Aardvark or to change 'Aardvark' to something else.

c) If the insert, delete or update happens ahead of the cursor (e.g. before it gets to the row in question), the operation impacts the count. But if the insert, delete or update happens behind the cursor (e.g. after it has passed the row in question), the operation has no impact on the count.

d) In terms of serialization order, if the insert, delete or update happens ahead of the cursor, then the insert/delete/update transaction in which it occurs must be BEFORE the count transaction; if the insert, delete, or update happens behind the cursor, the the insert/delete/update transaction must be AFTER the count transaction. Our concurrency control mechanism must take this into account in terms of locking or testing for serializability.

However, in the case where the insert/delete/update occurs behind the cursor, the changes to the row in question do not actually affect the way we count the row in the count transaction, so we have a case where the two transactions may conflict over a row that is not even part of one of them! (A "phantom row").

e) How can we expect a transaction to lock (or test the timestamps) of a row that is not even part of it?

(1)One solution involves in allowing an additional kind of lockable entity - the right to insert, delete, or update rows in a table.

(a)The insert, delete, or update transaction must obtain an exclusive lock on this right before doing its operation.

(b)The count transaction, or any transaction like it that scans rows in the table must obtain a shared lock on this right to ensure that no rows are inserted, deleted, or update while it is iterating over the table.

(c) Note that this lock is NOT the same as a multi-granularity lock on the table itself. We do not need to lock other transactions totally out of the table, if they do not depend on the table not changing under them (e.g. they are accessing a specific row based on primary key.)

(2) The book discusses an alternative based on locking B+ tree index leaves rather than the whole table, which allows more concurrency.

B. Weak Levels of Consistency

1. We have looked at a number of techniques for ensuring that any actual schedule produced by executing transactions concurrently is serializable. Basically, such techniques rely on one or the other of the following strategies:

   a) Making a transaction wait for some lock to be released before it can proceed.

   b) Rolling back a transaction that attempts an operation that would otherwise make the resulting schedule non-serializable.

   Of necessity, either sort of measure reduces the rate at which transactions can be processed - either by making a transaction wait, or by making it start over. Thus, concurrency control and maximizing throughput can conflict with one another.

2. Often times, there will be transactions for which an approximately correct answer is close enough. In such cases, we may choose to forgo strict enforcement of serializability to improve throughput. The book discusses this at some length, but we will not discuss this further, except to recall that SQL incorporates an ability to specify that a given query is to be run with weakened serializability requirements being enforced.

DB2 allows the specification of different serializability levels at the level of individual packages of compiled code in the database. The terminology and precise levels of isolation provided is a bit different from that discussed above - see the manual for details. (You will see output about this when binding packages to the database for your programming project, and you will see an example of this on a homrwork problem)

C. Locking and Index Structures

1. All of our discussion of locking has focussed on issues related to the actual locking of data in the tables themselves. In the case of a table that has one or more indexes, we also need to consider the impact of various operations on the index structure.

2. In the case of a dense index, every insert or delete operation necessarily impacts the index as well. Updates to a column that is the basis for an index also impact the index. (In effect, an update involves deleting the old entry and inserting a new one.)

3. In the case of a sparse index, insert, delete and update operations may or may not impact the index.

4. The book discusses locking issues involved with indexes. We will not pursue these here.