Lecture 27 — Snapshot Isolation, Weak Consistency, Insert/Delete

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 19, 2017

Snapshot Isolation for Concurrency Control

We already got a look at the idea of transaction isolation but we are now going to examine more carefully how it works behind the scenes.

In general, the idea is that every transaction has its own “world” it can operate in and then we need to merge the result when the transaction is ready to commit.

And that needs to be done atomically.

Validation

We can use validation to decide whether a transaction is allowed to commit.

This applies really only for transactions that do at least one write.

If the two writes are on disjoint items, there are no problems there either.

It only gets interesting if there are two transactions that have items in common that run at the same time.

Remember that here, when we talk about concurrency, we mean two transactions that are “active” at the same time.

Lost Updates

Usually, we do not like to just allow both transactions to go forward because the first write is then overwritten by the second.

That is called a lost update.

This is sometimes acceptable as a choice, even if some textbooks say that this is really undesirable and horrible.

First Committer Wins

The first snapshot isolation strategy is called first committer wins.

Whichever transaction T is ready to commit first has to pass a simple test.

If any transaction concurrent with T has already written an update to any data item that T wants to write, T is rolled back.

Otherwise T commits and its updates are written to the database.

It's called first committer wins, because whatever transaction gets to the commit statement proceeds and any later transactions are rolled back.

First Updater Wins

The alternative is first updater wins.

It is the timing of the write rather than the timing of the commit that indicates which transaction is allowed to proceed and which one(s) must be rolled back.

Locking and Unlocking

Locks are always released when the transaction either commits or aborts as is necessary.

A transaction T_2 writes a value later than T_1 writes that same value might actually succeed if for some other reason the first transaction aborts.

But to keep things from getting rather confusing it is likely a simple implementation will just force the abort of T_2.

Serializability

This scheme as presented, however, does not ensure serializability.

Example 1: We have two concurrent transactions T_1 and T_2 that operate on A and B.

If T_1 reads A and B, then updates B and T_2 reads A and B and updates A, we have the potential for a conflict.

Neither transaction sees the updates of the other when they do their reads.

When it comes to the write... well...

Neither transaction is detected as conflicting and neither is rolled back.

Serializability

In spite of that, we have a problem!

We have an outcome (output data) that could not happen if we had serial execution of the transactions.

It looks like under this scheme, some things like foreign key checks, primary key checks, et cetera, cannot be checked in the transaction world itself.

They must be checked an extra time when the transaction commits and is finally ready to replace the actual stored version on the database.

Serializability

Imagine we still have T_1 and T_2 and A and B as before.

T_1 reads B and updates B, and T_2 reads A and B and updates A.

As before, there are no direct conflicts on the data items so neither first-committer-wins nor first-updater-wins would detect a problem.

Furthermore, there's a possible serial order here: there are no conflicts on A, and no cycle in the precedence graph:

T_2 reads the value of B that existed before the write by T_1.

A Wild Transaction Appears!

A third transaction, even one that is read-only, can cause the cycle to occur, however.

If T_1 commits and T_2 is still running, and T_3 is created and its view of the world has the update from T_1 but not T_2...

Suddenly we have the cycle in the precedence graph, rendering the schedule non-serializable.

Just Get It Done!

After all that, though, we can accept some degree of non-serializability as long as it does not produce inconsistent results.

For performance and other reasons it might be sensible to just let execution orders that are not serializable proceed.

Or we will allow such things if we are convinced that inconsistencies don't make a big difference.

Key Constraints

We might have to really enforce constraints: the primary key.

If you have a sequential counter of some sort, an ID number, then two newly created entries could produce the same new number.

Neither transaction can detect that another has tried to use the same ID number.

Only when we are actually ready to perform the second commit will the duplicate key value be detected.

Insertion and Deletion

All the discussion thus far on the subject of concurrency control has excluded the possibility of performing an insertion or deletion on the data items.

Only read and writes were allowed.

But those are pretty important operations, so we can’t ignore them forever.

Deletion

The impact of delete depends, obviously, on what the else is going on at the time.

We can take a view of what happens by considering the relation of the delete instruction relative to other instructions.

The delete operation requires an exclusive lock on that data item.

Deletion

- **Read instruction**: If the delete $I_1(X)$ happens before a read of X then the read will get a logical error; if the read happens before the delete, there is no problem.
- **Write Instruction**: If the delete happens before the write, the write will get a logical error; if the write happens before the delete, there is no problem.
- **Delete Instruction**: Whichever instruction goes first will succeed; the second one will have an error (can't delete a deleted item).
- **Insert Instruction**: If the element doesn't exist to begin with, the delete can't be before the insert, otherwise there's nothing to delete... If the item did exist then a delete should go first so the new item replaces it.

Insertion

The insert operation conflicts were explained above when discussing deletion, so they don’t need to be repeated here.

Insert is also a lot like a write, so again, the insert operation is treated like a write.

The newly created item is treated as exclusively locked by its creating transaction until the transaction commits.

Perspective

At this level that the changes we want to make here are at the level of actual manipulation of the underlying data.

It is perfectly valid, from the perspective of the user writing the SQL, to do a read and get back “no results” because the item was very recently deleted.

The problem here is that we have identified what data elements we need to access and the ones we want are not there.

The Phantom of the Op... Database

The Phantom of the Op... Database

Suppose the following transactions are executed concurrently:

(1) a select statement that finds all books published by author “Jim Butcher”;

(2) an insert statement that adds a new book with author “Jim Butcher” amongst its attributes.

Sing Once Again With Me / Our Strange Duet

There are two possibilities for how this might go:

If T_2 happens first, nothing weird happens – in the serial schedule, first it's T_2 then T_1 and the new book appears in the results.

In the second case, if T_1 goes first and it does not contain the new book, then in the serial schedule, T_1 appears first...

Except... these transactions do not operate on any tuples in common and yet there is still some dependencies between them.

They conflict on a "phantom" tuple (one that doesn't exist).

This is called the Phantom Phenomenon

My Power Over You / Grows Stronger Yet

The phantom phenomenon can also happen if a read changes the value.

If an item was entered incorrectly as “Jim Butchr”, a corrective update statement run at the same time as T_1 above produces the same problem.

As a first solution, you might consider that an insertion or deletion should exclusively lock the relation entirely.

This can work but is very painful!

And Though You Turn From Me / To Glance Behind

Remember the index?

So far we have just assumed that data items being updated are tuples, but that's not strictly true, because index data elements also need to be lockable.

Idea: let's associate a data item (available to be locked) with each table, with the goal that it represents the way to find tuples in the relation.

The Phantom of the Database Is There

To update the data about what is in the relation, and transactions that want to read what tuples are in the relation need to also lock this item.

And anything that operates on an index must lock the index.

This moves the lock out of the realm of shadows and into the “real” world – the lock conflicts here are now on an actual lock item.

Inside Your Mind

Keep in mind this is very much distinct from locking the entire table.

Holding this sort of lock only restricts whether other transaction can update (or read) the information about what tuples are in the relation.

Even so, it's not all that dissimilar from the idea of having one lock for the whole table, in that it almost serializes all transactions.

An alternative is index locking, which allows us to lock parts of the index at finer granularity.

He's There, The Phantom of the Database

- Every relation must have at least one index.
- A transaction T_i can only access tuples after it has found them using an index (if we must do a full table scan, it's treated as if we just crawled through the whole of some index...)
- A transaction that performs a lookup (finding one tuple or multiple) must get a shared lock on all the index leaf nodes it accesses.
- A transaction may not insert, delete, or update any tuples in the relation without updating every index of that relation. That requires getting exclusive locks on all index leaf nodes affected by the operation.
- Locks are still obtained on tuples as usual.
- Two phase locking protocol still must be followed.

Predicate Locking

You may imagine that the idea of index locking does not actually match very well with the idea of making the locking match the predicates only.

Example: only checking on things that affect author.

That sort of locking technique sounds good but it is more difficult to implement.

Weak Consistency

The SQL standard provides several isolation levels, and it's not necessary that we stick with serializability as the level of consistency we are willing to accept.

We can weaken the rules a bit to get some more performance out of the database and that gives us what is called weak consistency.

There are rules, but they are more like... guidelines?

Degree-Two Consistency

The idea of degree-two consistency is to prevent cascading rollbacks (aborts) without guaranteeing serializability.

There are shared and exclusive locks, but two-phase behaviour is not required.

Shared locks can be released at any time and locks can be acquired at any time, but exclusive locks must be held until the transaction commits or aborts.

This means that we might read out of date data, but uncommitted values can never be read, so the level of transaction isolation here is “read-committed”.

Cursor Stability

Cursor stability is a form of degree-two consistency for programs that iterate over some set of tuples using an iterator or cursor.

This means that the tuples are examined or processed one at a time, in some order.

The tuple currently being examined needs to be locked: before processing it is locked in shared mode.

If any processing changes that tuple then it must first be locked in exclusive mode.

Cursor Stability

We don’t get serializability of the transactions.

It can be a way to improve performance in a database where there are a number of relations that are popular and frequently accessed.

But ultimately this offloads some of the work onto the applications using the database.

They must make sure they don’t have a problem with non-serializable schedules.

Users Ruin Everything...

Remember the example earlier about selecting seats on a flight (or at a concert or sporting event...).

When you go to select seats, you are presented with a map and you get to make your choice about which seat.

You have some period of time, e.g., 2 to 10 minutes to get this done.

That is a very long time as far as the database is concerned.

If two transactions select the same seats then we reject the second transaction.

Users Ruin Everything...

This does have a major drawback when we're dealing with user-level timeframes.

The database must remember information about updates performed by a transaction long after it has ended...

For as long as any other concurrent transaction is still alive.

In the case of transactions like a booking that give you 10 minutes of time, this means a transaction's information needs to be retained for up to 20 minutes.

Split 'em Up

A more likely solution is that we need to split it up into multiple transactions.

One transaction is completed to read the data and it is provided to a client application (e.g., web browser).

Then another transaction is needed when the user is ready to save changes: we create another transaction to save the data back to the database.

In The Upside Down

The data may have changed in the meantime, and if they did the attempt to merge the changes in that transaction will result in an error.

What the application needs to do is reload the data (repeat the read(s) that produced the data elements and see if they changed.

Detecting changes isn't as simple as just comparing fields, because it (1) would require saving the original read version, and (2) the ABA problem.

ABA Problems

The solution to the ABA problem is version numbers: every tuple is assigned a number that represents the version.

It is initialized to 0 and incremented (atomically) each time the tuple is updated.

Then a transaction needs only to look at the version number.

If the version number doesn’t match the original read, then it has changed.

Holy Long Names Batman!

This is called optimistic concurrency control without read validation.

We assume we are going to succeed without conflict (are optimistic) and rollback if we must.

It is without read validation, because we check version numbers only for writes that we are going to perform.

If we want optimistic concurrency control, then we must also check the version numbers of any reads that went into the transactions too.

Concurrency in Index Structures

We can use special handling for the index structures to improve performance.

More obviously, we can allow non-serializable access to the index as long as the results are consistent.

Crabbing Protocol

The first technique for locking on index structures is the crabbing protocol.

When searching for a key value, first lock the root note in shared mode.

When traversing down the tree, acquire a shared lock on the child node and release the lock on the parent node, repeating this until a leaf node is reached.

When performing an insertion or deletion of a leaf node:

- Follow the same protocol to reach the leaf node (acquire and release shared locks).
- Lock the leaf node in exclusive mode and perform the insertion or deletion.
- If a split or coalescence is needed, lock the parent in exclusive mode and do the changes; release the leaf nodes.
- If the parent itself needs splitting or coalescing, retain the lock on the parent and propagate the changes; otherwise release the parent.

B-Link Trees

The second technique we will talk about requires a modification of the B^+-trees called B-Link trees.

These have the added restriction that says every node (really, all of them) have a pointer to its sibling to the right.

This helps in case a search is taking place at a time when a node is being split and it can continue.

B-Link Lookup

Each node must be locked in shared mode before it is requested.

A lock on any non-leaf node is released before a lock on any other node is requested.

If a split takes place while a lookup is happening, then we might need to look in the sibling to find something.

Leaf nodes are locked using the two phase protocol.

B-Link Insertion/Deletion

The rules for lookup are followed to find the leaf node where the records will be inserted or deleted.

The lock for the leaf is upgraded to exclusive, and the insertion or deletion is performed.

Two phase locking is used to avoid the phantom phenomenon.

If as a result of insertion or deletion, a split or coalescence is needed...

B-Link Split

If a node is split, a new node is created as per normal for a B-Tree and it becomes the right sibling of the original node.

The previous right sibling of the original is now the right sibling of the new node.

Then the original node is released and an exclusive lock on the parent is requested to insert a pointer to that new node.

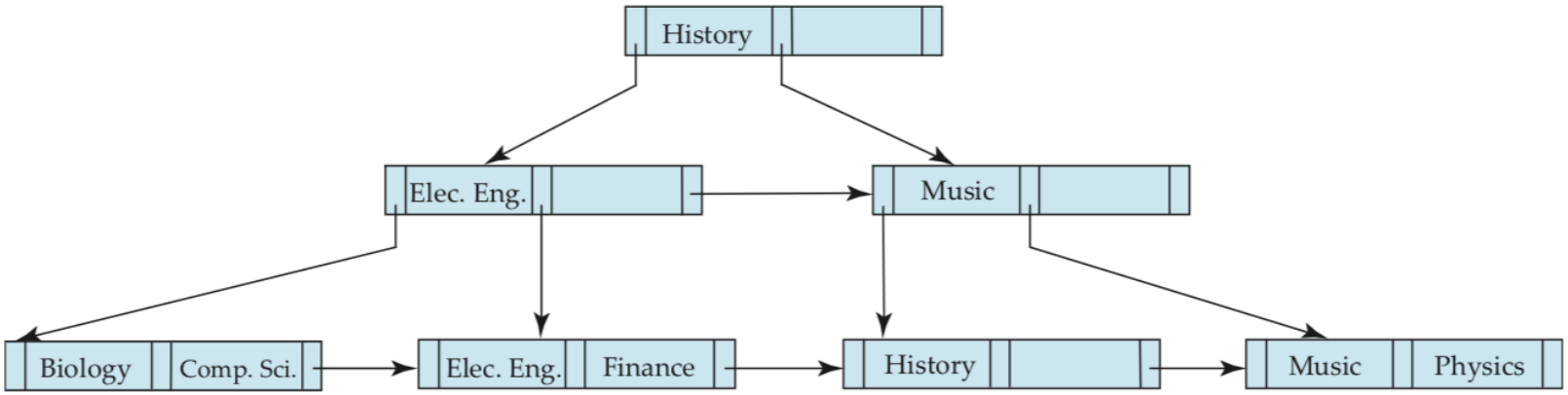
Coalescence

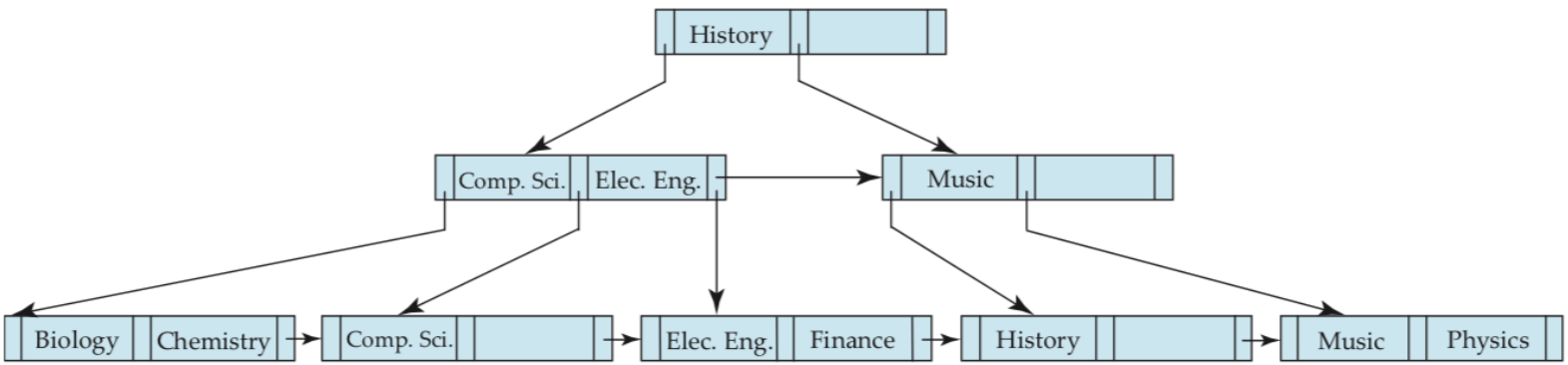
If a node need to be coalesced, then the node it will be merged with must be locked in exclusive mode.

Then the parent node of the newly-created node must be locked exclusively so that the deleted node can be removed.

The parent may be coalesced as well if necessary, otherwise it is released.

B-Link Chemistry





B-Link Chemistry

Now let's see what happens if there is a lookup while this is going on at this point.

The lookup gets blocked at the bottom-left leaf node because it is exclusively locked by the insertion.

The lookup holds no locks at this point.

What if instead of a split there was a coalescence?

Key-Value Locking

Key-value locking allows locking individual key values, allowing other operations to take place on values within the same leaf node.

The problem is that we could still have the phantom phenomenon, so the strategy of next key locking is employed.

Not only lock the keys that are in the range/search result but also whichever one is next.

This prevent something from being inserted, altered, or deleted inside the search range.