ECE356: Database Systems Winter 2018

Lecture 24 — Timestamp Protocols

Jeff Zarnett

Concurrency via Timestamps

Beyond just doing concurrency through some sort of locking protocol, we can also use timestamps for arranging concurrency. The timestamp, as you will recall, is a unique identifier, and they are typically assigned in the order the transactions are submitted to the system [EN11]. If timestamps are used appropriately, then consistent results are produced without the use of any locks or locking, meaning deadlock can never occur.

The notation for the timestamp of a transaction T is $TS(T)$. The transactions can be ordered based on their timestamps; if a transaction T_i arrives and then later another transaction T_j arrives, then $TS(T_i) < TS(T_j)$. If two transactions arrive at exactly the same time, then some sort of serialization procedure will be needed to make sure that they are not identical. It is commonly the case that the system clock is used to get the time stamp; just read the current value of the clock and use that.

The alternative is a logical counter: increment a simple integer counter for each transaction. After a timestamp is assigned, just increment the counter. Eventually there is a limit (even if you can have 2^{64} transactions, in theory, which is quite a lot, but not infinite) but at some point the transaction counter will need to roll over or reset.

Generation of the timestamps is fairly simple, it would seem, but what are they for? Timestamps are used for serializability order in the schedule; the system must ensure that the schedule executed is equivalent to a serial schedule in which transactions are ordered according to their timestamps (ascending) [SKS11]. Note that this is very different from two-phase locking. In two phase locking, a schedule is serializable by being equivalent to some schedule that is permitted under the locking rules; in timestamp ordering the schedule is equivalent to the a particular serial ordering matching ascending transaction timestamps [EN11].

Every data element in the database is associated with separate timestamps for reading and writing. The notation can vary: in [EN11] they are called **read_TS** and **write_TS**; in [SKS11] they are **r-timestamp** and **w-timestamp**. To save space I might personally like TS_r and TS_w. Each time the a data element X is read or written, its appropriate timestamp is updated to the timestamp of the transaction performing the read or write.

The formal definition of the read timestamp of an item X: the largest timestamp amongst all the timestamps of transactions that have successfully read X. So $TS(X) = TS(T_k)$ where T_k is the youngest transaction that has read X successfully [EN11].

The formal definition of the write timestamp of an item X: the largest timestamp amongst all the timestamps of transactions that have successfully written X. So $TS(X) = TS(T_k)$ where T_k is the youngest transaction that has written X successfully [EN11].

Unfortunately, simple timestamp ordering does not avoid the risk of cascading rollback. Schedules are not guaranteed to be recoverable. This leads us to the *timestamp ordering protocol* as described in [SKS11].

For a read operation T_i is performing on X:

1. If $TS(T_i) < TS_w(X)$ then T_i needs to read a value of X that was already overwritten – and therefore the read operation is not permitted and T_i is rolled back.
2. If $TS(T_i) \geq TS_w(X)$ the read operation is executed and TS_r is set to $\max(TS(T_i),\ TS_r(X))$.

For a write operation T_i is performing on X:

1. If $TS(T_i) < TS_r(X)$ then the value that T_i is producing was needed previously but the system assumed it wasn't going to happen and proceeded without it; thus the write is not permitted and T_i is rolled back.
2. If $TS(T_i) < TS_w(X)$ then T_i is attempting to write an obsolete value; the write is not permitted and T_i is rolled back.
3. Otherwise, the write proceeds and $TS_w(X)$ is updated to $TS(T_i)$.

When a transaction is rolled back and restarted, the transaction is assigned a new timestamp. The timestamp ordering protocol does provide us with conflict serializability, because conflicting operations are processed in timestamp order. Because there are no locks, there cannot be deadlocks – but can starvation still occur? The answer is yes, a very long transaction might be constantly frustrated by shorter transactions constantly swooping in and making changes that force a rollback of the long transaction. Unfortunately it might be necessary to block other transactions and let the long one finish [SKS11].

It is worth noting that neither two phase locking nor the simple timestamp ordering protocol covers all serializable schedules; that is, there are some that are valid under one but not valid under the other [EN11]. That's fine, as far as we are concerned – we don't need to consider all possibilities; we will get enough choices.

If we would like to recoverability we can have strict timestamp ordering. In strict timestamp ordering, a transaction T_i that issues a read or write of an item X where $TS(T_i) > TS_w(X)$, then the read or write operation is delated until the transaction that last wrote X has either committed or aborted. This, in some way, requires simulating locking item X until a previous write has been committed or aborted, but there cannot be a deadlock because a transaction can only wait on an older transaction (one with a lower timestamp) [EN11].

Thomas's Write Rule. A modification of the basic algorithm called Thomas's Write Rule (or the Thomas[1] Write Rule) allows greater concurrency than the basic algorithm and also, hopefully, rejects fewer writes. To explain it simply, it is "ignore outdated writes".

More formally, the write item checks are modified to the following [EN11]:

1. If $TS_r(X) > TS(T_i)$ then abort and roll back T_i (write rejected).
2. If $TS_w(X) > TS(T_i)$, then do not execute the write, but continue executing (do not roll back the transaction).
3. Otherwise, the write proceeds and $TS_w(X)$ is updated to $TS(T_i)$.

This requires some explanation. If a transaction T_i is going to write an old value, under the previous scheme that write would be rejected. Under Thomas's Write Rule, we will just skip doing the write (the value to be written is outdated) and we just carry on. Essentially we just pretend that the write happened but was immediately overwritten by the more up to date value.

References

[EN11] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 6th Edition*. Addison-Wesley, 2011.

[SKS11] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 6th Edition*. McGraw Hill, 2011.

[1] Named after the author of the paper describing this rule, Robert H. Thomas