Timestamp-based Concurrency Control and the Thomas Write Rule

Wei Lu

April 12, 2013

Based on Ch. 16 in *Database System Principles*, Silberschatz, Korth, and Sudarshan.

Timestamps

- Each transaction T_i, upon starting up, is assigned a timestamp $\mathsf{TS}(T_i)$.
- This can be implemented using either the system clock, or a logical counter that is incremented after a new timestamp is issued.
- Timestamps are used to determine the serializability order: if $\mathsf{TS}(T_i) < \mathsf{TS}(T_j)$, then for a schedule to be valid, it must be equivalent to some *serial* schedule in which T_i appears before T_j.
- Each data item Q is associated with two timestamp values.
  - $\mathsf{WTS}(Q)$: the timestamp of the most recent transaction that successfully executed **write**(Q).
  - $\mathsf{RTS}(Q)$: the timestamp of the most recent transaction that successfully executed **read**(Q).

Timestamp-Ordering Protocol and Its Rules

When a transaction T_i issues a **read**(Q) instruction:

- If $\mathsf{TS}(T_i) < \mathsf{WTS}(Q)$, T_i would read a value of Q that was already overwritten by a newer transaction $T_j \neq T_i$. Hence, this read is rejected and T_i will be rolled back.
- If $\mathsf{TS}(T_i) \geq \mathsf{WTS}(Q)$, the read is approved, and we set $\mathsf{RTS}(Q) := \max\{\mathsf{RTS}(Q), \mathsf{TS}(T_i)\}$.

Since multiple **read**(Q)'s are not conflict actions, there is no need to compare $\mathsf{TS}(T_i)$ and $\mathsf{RTS}(Q)$.

When a transaction T_i issues **write**(Q):

- If $\mathsf{TS}(T_i) < \mathsf{RTS}(Q)$, the write is rejected and T_i will be rolled back.
- (†) If $\mathsf{TS}(T_i) < \mathsf{WTS}(Q)$, then T_i is trying to write an *obsolete* value of Q, and hence it's not allowed and T_i is rolled back
- Otherwise, the write is approved, and we set $\mathsf{WTS}(Q) := \mathsf{TS}(T_i)$.

Once again, a schedule must be equivalent to some *serial* schedule in which T_i appears before T_j, and this is the very reason behind all rejection rules specified above.

The Thomas Write Rule

Can we relax the above rules to allow greater level of concurrency and avoid unnecessary rollbacks? It turns out we can. Rule (†) disables obsolete writes, but the roll-back is not really necessary. Hence, we replace (†) with the *Thomas Write Rule* (‡).

- (‡) If $\mathsf{TS}(T_i) < \mathsf{WTS}(Q)$, *ignore* this write.

Life becomes much simpler, right?

Why the Thomas Write Rule is correct?! Essentially, the question is why the Thomas Write Rule still guarantees the serializability order for the protocol. Below is a proof, which is essentially based on the notion of *view-equivalent* or *view-serializability*[1].

First, if $\mathsf{TS}(T_i) < \mathsf{WTS}(Q)$, then by definition of the protocol, there must exist some $T_j \neq T_i$, such that T_j is the most recent transaction executing **write**(Q) successfully and that $\mathsf{TS}(T_j) = \mathsf{WTS}(Q) > \mathsf{TS}(T_i)$.

Then, consider any other transaction T_k that is executed concurrently with T_i and T_j. Suppose T_k issues a **read**(Q). There are two possibilities:

1. If $\mathsf{TS}(T_k) < \mathsf{TS}(T_j)$, then $\mathsf{TS}(T_k) < \mathsf{WTS}(Q)$, and thus this read will not be allowed, with T_k being rolled back.

2. If $\mathsf{TS}(T_k) \geq \mathsf{TS}(T_j)$, then $\mathsf{TS}(T_k) \geq \mathsf{WTS}(Q)$, and thus T_k must read the value of Q written by T_j, rather than that by T_i.

Therefore, T_i is trying to write an out-dated value of Q that will never need to be read, under the timestamp-ordering protocol. Now that we have dealt with T_k's **read**(Q) request, what if T_k wants to **write**(Q)? In this case, apparently, T_k will be no different from T_i, and thus our argument still stands. This completes the proof.

[1] I hope you do understand these two concepts, though.