CS322: Database Systems

Concurrency Control:

Timestamp-based Protocol, Validation-based Protocol, Snapshot Isolation

Dr. Manas Khatua
Assistant Professor
Dept. of CSE
IIT Jodhpur
E-mail: manaskhatua@iitj.ac.in

Timestamp-based Protocol

- Lock-based protocol determines the order between every pair of conflicting transactions at execution time
- *But, timestamp-ordering* protocol determines the serializability order in advance. It start working as soon as a transaction is created.
- With each transaction T_i in the system, we associate a unique fixed timestamp, denoted by TS(T_i).
- This timestamp is assigned by the database system before the transaction T_i starts execution.
- If a transaction T_i has been assigned timestamp TS(T_i), and a new transaction T_j enters the system, then TS(T_i) < TS(T_j).
- Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction.

Cont…

- Methods for assigning timestamp:
  - Use the value of the system clock as the timestamp
  - Use a logical counter that is incremented after a new timestamp has been assigned

- The timestamps of the transactions determine the serializability order.

- To implement this scheme, we apply two timestamps for an item Q
  - W-timestamp(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully.

  - R-timestamp(Q) denotes the largest timestamp of any transaction that executed read(Q) successfully.

Timestamp-Ordering Protocol

- The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

- Suppose that transaction T_i issues *read(Q)*.
  - If $TS(T_i) < W\text{-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and T_i is rolled back.
  - If $TS(T_i) \geq W\text{-timestamp}(Q)$, then the read operation is executed, and R-timestamp(Q) is set to the maximum of {R-timestamp(Q) and $TS(T_i)$}.

- Suppose that transaction T_i issues *write(Q)*.
  - If $TS(T_i) < R\text{-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously, and it is not needed now. Hence, the system rejects the write operation and rolls T_i back.
  - If $TS(T_i) < W\text{-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls T_i back.
  - Otherwise, the system executes the write operation and update W-timestamp(Q), sets W-timestamp(Q) to $TS(T_i)$.

Example

T_{25}: read(B);
read(A);
display($A + B$).

T_{26}: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Figure 15.17 Schedule 3.

- The rolled back transaction is assigned a new timestamp and restarts.

- The timestamp-ordering protocol ensures conflict serializability
- The protocol ensures freedom from deadlock, since no transaction ever waits.
- There is a possibility of starvation of long transactions
- The protocol can generate schedules that are not recoverable

Cont…

- To make the schedules recoverable
  - Recoverability and cascadelessness can be ensured by performing all writes together at the end of the transaction
  - Recoverability and cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits
  - Recoverability alone can be ensured by tracking uncommitted writes, and allowing a transaction T_i to commit only after the commit of any transaction that wrote a value that T_i read. Commit dependencies can be used for this purpose.

Thomas' Write Rule

- A modification to the timestamp-ordering protocol that allows greater potential concurrency than timestamp protocol

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

Figure 15.18 Schedule 4.

- Modification:
  - Obsolete write operations can be ignored under certain circumstances. It is called **Thomas' write rule**

- Suppose that transaction T_i issues write(Q).
  - If $TS(T_i) < R\text{-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously, and it is not needed now. Hence, the system rejects the write operation and rolls T_i back.
  - If $TS(T_i) < W\text{-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.
  - Otherwise, the system executes the write operation and update W-timestamp(Q), sets W-timestamp(Q) to $TS(T_i)$.

Validation-Based Protocols

- A concurrency-control scheme imposes overhead of code execution and possible delay of transactions.
- However, in cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low.

- How to reduce the overhead in such cases?
  - A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict.
  - We need a scheme for monitoring the system

- The validation protocol requires that each transaction T_i executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction.
  - Read phase
  - Validation phase
  - Write phase

Cont…

- Read phase. Transaction T_i reads the values of the various data items and stores them in variables local to T_i. It performs all write operations on temporary local variables, without updates of the actual database.

- Validation phase. This determines whether T_i is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.

- Write phase. If the validation test succeeds for transaction T_i, the temporary local variables that hold the results of any write operations performed by T_i are copied to the database.

- To perform the validation test, we need to know when the various phases of transactions took place. So, need of timestamps
  - Start(T_i), the time when T_i started its execution
  - Validation(T_i), the time when T_i finished its read phase and started its validation phase.
  - Finish(T_i), the time when T_i finished its write phase

Cont…

- The validation test for transaction T_i requires that, for all transactions T_k with $TS(T_k) < TS(T_i)$, one of the following two conditions must hold
  - (1) Finish(T_k) < Start(T_i).
  - This condition ensures that the writes of T_k finishes before T_i starts.

    OR
  - (2) The set of data items written by T_k does not intersect with the set of data items read by T_i, and
  - T_k completes its write phase before T_i starts its validation phase ($Start(T_i) < Finish(T_k) < Validation(T_i)$).
  - This condition ensures that the writes of T_k and T_i do not overlap.

- It generates serializable schedule
- It guards against cascading rollbacks
- However, there is a possibility of starvation of long transactions

Cont…

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
< validate>	
display($A + B$)	
	< validate>
	write(B)
	write(A)

Figure 15.19 Schedule 6, a schedule produced by using validation.

Snapshot Isolation

- Snapshot Isolation has gained wide acceptance in commercial and open-source systems, including Oracle, PostgreSQL, and SQL Server because of less complexity

- Basic Idea:
  - Each transaction is given a "snapshot" of the database at the time when it begins its execution
  - It then operates on that snapshot in complete isolation from concurrent transactions.
  - The data values in the snapshot consist only of values written by committed transactions.
  - Transactions that wants to update the database must interact with potentially conflicting concurrent update transactions before updates are actually placed in the database.
  - Updates are kept in the transaction's private workspace until the transaction successfully commits
  - When a transaction *T* is allowed to commit, all the updates made by T to the database must be done as an atomic action

Validation Steps for Update Transactions

- Lost Update:
  - Let two transactions operate in isolation using their own private snapshots,
  - neither transaction sees the update made by the other.
  - If both transactions are allowed to write to the database, the first update written will be overwritten by the second.
  - The result is a lost update.

- Concurrent Transaction:
  - A transaction is said to be concurrent with *T* if it was active or partially committed at any point from the start of *T* up to and including the time when this test is being performed.

- There are two variants of snapshot isolation, both of which prevent lost updates.
  - *first committer wins*
  - *first updater wins*

First Committer Wins

- when a transaction *T* enters the partially committed state, the following actions are taken in an atomic action:
  - A test is made to see if any transaction that was concurrent with T has already written an update to the database for some data item that T intends to write.
  - If some such transaction is found, then *T* aborts.
  - If no such transaction is found, then T commits and its updates are written to the database.
- This approach is called “first committer wins” because
  - if transactions conflict, the first one to be tested using the above rule succeeds in writing its updates, while the subsequent ones are forced to abort.

First Updater Wins

- The system uses a locking mechanism that applies only to updates
- When a transaction T_i attempts to update a data item, it requests a *write lock* on that data item.

- If the lock is not held by a concurrent transaction, the following steps are taken after the lock is acquired:
  - If the item has been updated by any concurrent transaction, then T_i aborts.
  - Otherwise, T_i may proceed with its execution including possibly committing.

- If, however, some other concurrent transaction T_j already holds a write lock on that data item, then T_i cannot proceed and the following rules are followed:
  - T_i waits until T_j aborts or commits.

- This approach is called "first updater wins" because
  - if transactions conflict, the first one to obtain the lock is the one that is permitted to commit and perform its update.
  - Those that attempt the update later abort unless the first updater subsequently aborts for some other reason.

Serializability Issues: Case 1

- One serious problem: *snapshot isolation does* not *ensure serializability*
- Few non-serializable executions under snapshot isolation

T1	T2
Read(A)	Read(A)
Read(B)	Read(B)
Write(B)	Write(A)

- Since *T1* and *T2* are concurrent, neither transaction sees the update by the other in its snapshot.
- Since they update different data items, both are allowed to commit regardless of whether the system uses the first-update-wins policy or the first-committer-wins policy.
- However, Precedence Graph has cycle; so the schedule is not serializable

Cont…

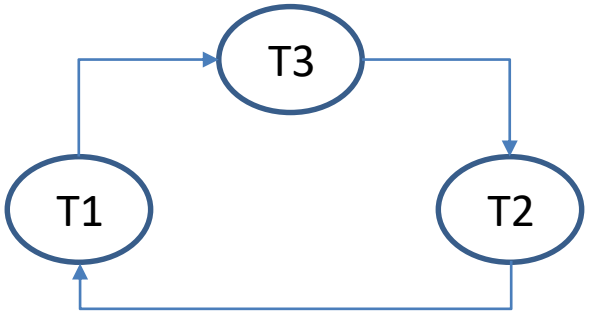
- Write skew: This situation, where each of a pair of transactions has read data that is written by the other, but there is no data written by both transactions

- Let, a customer has Current Account (CA) and Savings Account (SA).
- Present cash in CA = Rs. 100, and in SA= Rs. 200
- Suppose that the bank enforces the integrity constraint that the sum of the balances in the checking and the savings account of a customer must not be negative.
- Suppose that transaction *T*1 withdraws Rs.200 from the CA, after verifying the integrity constraint by reading both balances.
- Suppose that concurrent transaction *T*2 withdraws Rs.200 from the SA, again after verifying the integrity constraint.
- Since each of the transactions checks the integrity constraint on its own snapshot, there withdrawal do not violate the constraint.
- Under snapshot isolation both of them can commit.

- Does it create any problem?
  - Yes, the sum of the balances is Rs.-100, violating the integrity constraint.
- Possible Solution:
  - the database system must check these constraints on the current state of the database at the time of commit.

Serializability Issues: Case 2

- Running these two transactions concurrently causes no problem.
- There is no cycle in the precedence graph
- Suppose that, after *T1* commits but before *T2* commits, a new read-only transaction *T3* enters the system and *T3* reads both *A* and *B*.
- Its snapshot includes the update by *T1* because *T1* has already committed. However, since *T2* has not committed, its update has not yet been written to the database and is not included in the snapshot seen by *T3* .
- The updated precedence graph has cycle.



T1	T2
	Read(A)
Read(B)	Read(B)
Write(B)	Write(A)

T1	T2	T3
	Read(A)	
Read(B)	Read(B)	
Write(B)	Write(A)	
Commit		
		Read(A)
		Read(B)

Cont...

- Since consistency is the goal, we can accept the potential for non-serializable executions if we are sure that those non-serializable executions that might occur will not lead to inconsistency.

- Let we are dealing with a financial database
- Let the financial applications create consecutive sequence numbers, for example to number bills, by taking the maximum current bill number and adding 1 to the value to get a new bill number.
- If two such transactions run concurrently, each would see the same set of bills in its snapshot, and each would create a new bill with the same number!

- Creating two bills with the same number could have serious legal implications.

- The above problem is an example of the phantom phenomenon, since the insert performed by each transaction conflicts with the read performed by the other transaction to find the maximum bill number, but the conflict is not detected by snapshot isolation.

- An application developer can guard against certain snapshot anomalies by appending a for update clause to the SQL select query.

Thanks!