Database Management Systems (DBMS)

Lec 27: Transaction Processing, Concurrency Control, and Recovery (Contd.)

Ramesh K. Jallu
IIIT Raichur

Date: 18/06/21

Recap

- Cascadeless schedules

- Concurrency control schemes
  - Lock based protocols
    - Binary locks
    - Shared/Exclusive locks (Read/ Write locks)
    - The two-phase locking protocol

Today's plan

- Variants of the two-phase locking protocol
  - Conservative 2PL scheme
  - Strict 2PL scheme
  - Rigorous 2PL scheme

- Timestamp based protocols

Conservative Two-Phase Locking

- A ***conservative 2PL*** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*, by predeclaring its *read-set* and *write-set*

- If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, *it waits until all the items are available for locking*

- Conservative 2PL is a *deadlock-free protocol*

- However, it is difficult to use in practice because of the need to predeclare the *read-set* and *write- set*, which may not be feasible

Strict Two-Phase Locking

- A ***strict schedule*** is a schedule in which transactions can *neither read nor write* an item *X* until the last transaction that wrote *X* has committed (or aborted)

- A strict schedule is a cascadeless schedule

- In strict 2PL scheme, a transaction *T* does not release any of its exclusive (write) locks until *after* it commits or aborts

- Thus, no other transaction can read or write an item that is written by *T* unless *T* has committed

- Strict 2PL ensures strict schedule, but is *not a deadlock-free* protocol

Rigorous Two-Phase Locking

- In rigorous 2PL scheme, a transaction *T* does not release any of its locks (exclusive or shared) until after it commits or aborts

- Like in strict 2PL, no other transaction can read or write an item that is written by *T* unless *T* has committed

- Rigorous 2PL is *not a deadlock-free* protocol

- It ensures a strict schedule

Disadvantage of 2PL scheme

- WKT, locking combined with the 2PL protocol guarantees serializability of schedules
- The equivalent serial schedule is based on the order in which executing transactions lock the items they acquire
- If a transaction needs an item that is already locked, it may be forced to wait until the item is released
- Some transactions may be aborted and restarted because of the deadlock problem

Concurrency Control Based on Timestamp Ordering

- Another approach to concurrency control involves using transaction timestamps to order transaction execution for an equivalent serial schedule

- Serializability is enforced by ordering conflicting operations in different transactions based on the transaction timestamps

- Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*

Timestamps

- A ***timestamp*** is a unique identifier for each transaction T_i in the system, denoted by $TS(T_i)$
- Typically, timestamp values are assigned in the order in which the transactions start execution
- If a transaction T_i has been assigned timestamp $TS(T_i)$, and a new transaction T_j enters the system, then $TS(T_i) < TS(T_j)$
- Two methods to generate timestamps
  1. System clock
  2. Logical counter

The Timestamp-Ordering Protocol (TOP)

- The TOP ensures that that any conflicting *read* and *write* operations are executed in timestamp order

- As a result, unlike 2PL, equivalent serial schedule obtained has the transactions in order of their timestamp values

- The protocol associates with each database item *X* two timestamp values
  - ***read_TS(X)***
  - ***write_TS(X)***

Contd.

- When a transaction T_i issues a *read_item*(X) or a *write_item*(X) operation, the **TOP** compares the timestamp of T_i with *read_TS*(X) and *write_TS*(X) to ensure that the timestamp order of transaction execution is not violated

- If this order is violated, then transaction T_i is aborted and resubmitted to the system as a new transaction with a *new timestamp*

- This protocol *suffers* from cascading rollback as the schedules produced are not guaranteed to be recoverable

- However, by enforcing an additional protocol we can ensure that the schedules are cascadeless and recoverable

Contd.

- When a transaction T_i issues the *write_item(X)*
  - If *read_TS(X)* > $TS(T_i)$ or if *write_TS(X)* > $TS(T_i)$, then abort and roll back T_i and reject the operation
  - Otherwise, the system executes the *write_item(X)* operation of T_i and sets *write_TS(X)* to $TS(T_i)$

- When a transaction T_i issues the *read_item(X)*
  - If *write_TS(X)* > $TS(T_i)$, then abort and roll back T_i and reject the operation
  - Otherwise, the system executes the *read_item(X)* operation of T_i and sets *read_TS(X)* to the *larger* of $TS(T_i)$ and the current *read_TS(X)*

Example

T_1: read(B);
read(A);
display($A + B$).

T_2: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

T_1	T_2
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Observations

- Whenever the TO algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it

- The schedules produced by TO are hence guaranteed to be *conflict serializable* because conflicting operations are processed in timestamp order

- As mentioned earlier, deadlock does not occur with timestamp ordering

- However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted

Strict Timestamp Ordering Protocol (STOP)

- This variation of TOP, called STOP, ensures that the schedules are both *recoverable* and *serializable*

- Uses a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits

- A transaction *T* issues a *read_item(X)* or *write_item(X)* such that *TS(T)* > *write_TS(X)* has its read or write operation *delayed* until the transaction *T'* that *wrote* the value of *X* has committed or aborted
  - The item *X* is locked by transaction *T'* until *T'* is either committed or aborted

- The STOP *does not cause deadlock*, since *T* waits for *T'* only if *TS(T)* > *TS(T')*

Thomas's Write Rule

- It is a modified TOP, which allows greater potential concurrency than TOP

- When T_{27} attempts its *write(Q)* operation, we find that TS(T_{27}) < *write_TS*(Q), since write_TS(Q) = TS(T_{28}). Thus, the write(Q) by T_{27} is rejected and transaction T_{27} must be rolled back

- Although the rollback of T_{27} is required by the timestamp-ordering protocol, it is unnecessary

- Since T_{28} has already written Q, the value that T_{27} is attempting to write is one that will never need to be read

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

Thomas's Write Rule (Contd.)

- Any transaction T_i with $TS(T_i) < TS(T_{28})$ that attempts a *read*(Q) will be rolled back, since $TS(T_i) < write\_TS(Q)$
- Any transaction T_j with $TS(T_j) > TS(T_{28})$ must read the value of Q written by T_{28}, rather than the value that T_{27} is attempting to write
- Thomas's write rule
  1. If $read\_TS(X) > TS(T)$, then abort and roll back T and reject the operation.
  2. If $write\_TS(X) > TS(T)$, then do not execute the write operation but continue processing
  3. If neither the condition in 1 nor the condition in 2 occurs, then execute the $write\_item(X)$ operation of T and set $write\_TS(X)$ to $TS(T)$

Thank you!