Database Systems
Lecture20 – Chapter 18: Concurrency Control

SUNG KYUN KWAN UNIVERSITY

Beomseok Nam (남범석)

bnam@skku.edu

Multiversion Concurrency Control

Multiversion Schemes

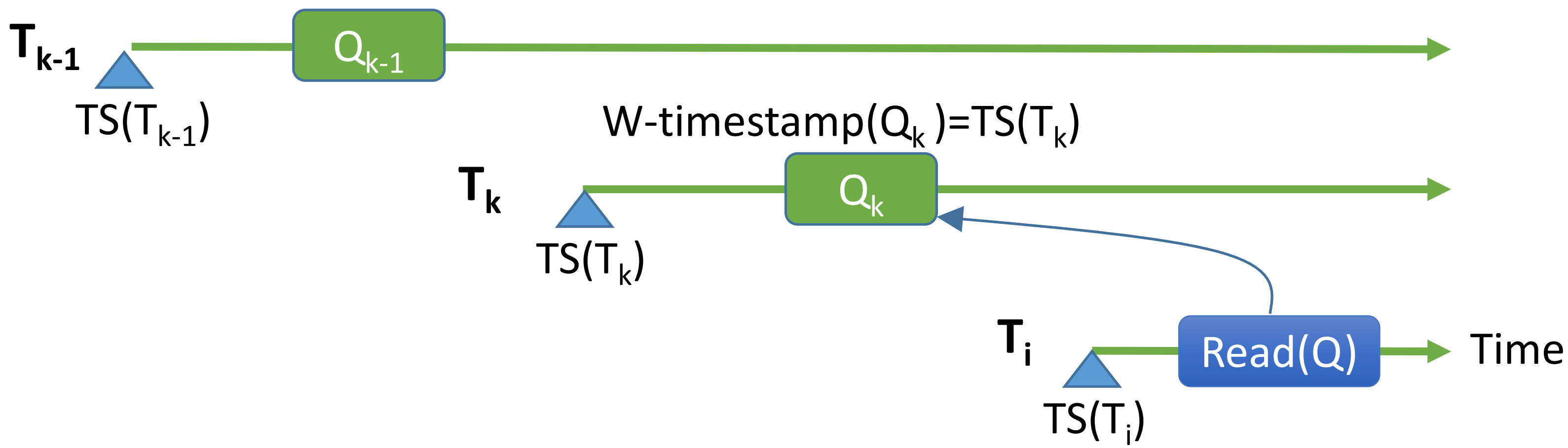
- Multiversion schemes keep old versions of data item to increase concurrency. Several variants:
  - **Multiversion Timestamp Ordering**
  - **Multiversion Two-Phase Locking**
  - **Snapshot isolation**
- Key ideas:
  - Each **write** results in the creation of a new version of the data.
  - Use timestamps to label versions.
  - When a **read**(*Q*) operation is issued, select/read an appropriate version of *Q* based on the timestamp of the transaction.
- **read**s never have to wait because an appropriate version is returned immediately.

Multiversion Timestamp Ordering

- Each data item *Q* has a sequence of versions $<Q_1, Q_2, ..., Q_m>$. Each version Q_k contains three data fields:
  - **Content** -- the value of version Q_k.
  - **W-timestamp**(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) -- largest timestamp of a transaction that successfully read version Q_k

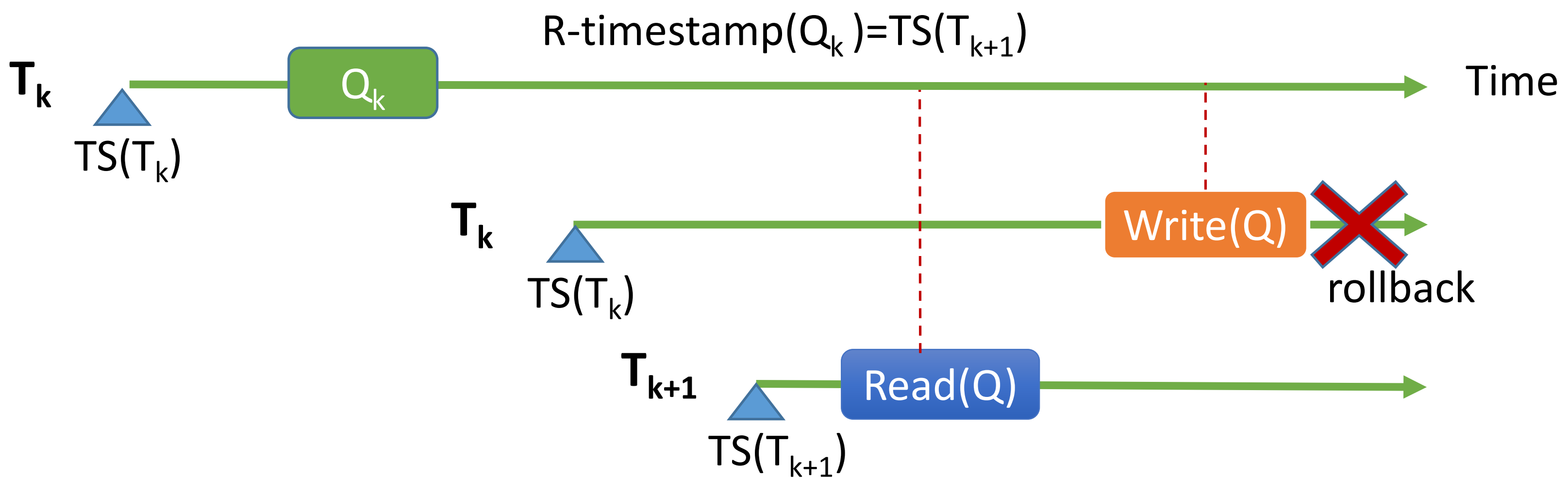
Multiversion Timestamp Ordering (Cont)

- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation.
- Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).



1. If transaction T_i issues a **read**(Q), then
   - return the content of version Q_k
   - If R-timestamp(Q_k) < TS(T_i), set R-timestamp(Q_k) = TS(T_i),

Multiversion Timestamp Ordering (Cont)



2. If transaction T_i issues a **write**(Q)
   1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
   2. if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten
   3. Otherwise, a new version Q_i of Q is created
      - W-timestamp(Q_i) and R-timestamp(Q_i) are initialized to TS(T_i).

Multiversion Timestamp Ordering (Cont)

- Observations
  - Reads always succeed
  - A write by T_i is rejected if some other transaction T_j that should read T_i's write, has already read a version created by a transaction older than T_i.
    - i.e., in the previous example, T_{k+1} should have read T_k's write.
- Protocol guarantees serializability

Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions
- **Update transactions** acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Read(Q) returns the latest version of the item
  - The first **write** of Q by T_i results in the creation of a new version Q_i
    - W-timestamp(Q_i) set to ∞ initially
  - When update transaction T_i completes, commit occurs:
    - Set TS(T_i) = **ts-counter** + 1
    - Set W-timestamp(Q_i) = TS(T_i) for all versions Q_i that it creates
    - **ts-counter** = **ts-counter +** 1

Multiversion Two-Phase Locking (Cont.)

- **Read-only transactions**
  - are assigned a timestamp = **ts-counter** when they start execution
  - follow the multiversion timestamp-ordering protocol for performing reads
    - Do not obtain any locks
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.
- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i.
- Only serializable schedules are produced.

MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g., if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again
- Issues with
  - primary key and foreign key constraint checking
  - Indexing of records with multiple versions

  See textbook for details

Snapshot Isolation

- Motivation: Decision support queries (OLAP transactions) that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results

- Snapshot Isolation: Give snapshot of database to every transaction
  - Takes snapshot of committed data at start
  - Always reads/modifies data in its own snapshot
  - Updates of concurrent transactions are not visible
  - Writes of a concurrent transaction complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that the transaction intends to write.

Snapshot Isolation

- Transactions executing with Snapshot Isolation

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) → 0 R(Y) → 1 W(X:=3) Commit-Req Abort	

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

Snapshot Read

- Concurrent updates invisible to snapshot read

$X_0 = 100,\ Y_0 = 0$

T_1 deposits 50 in Y	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
$r_1(Y_0, 0)$	
	$r_2(Y_0, 0)$
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(Y_1, 50)$	
$r_1(X_0, 100)$ (update by T_2 not seen)	
$r_1(Y_1, 50)$ (can see its own updates)	
	$r_2(Y_0, 0)$ (update by T_1 not seen)

$X_2 = 50,\ Y_1 = 50$

Snapshot Write: First Committer Wins

$X_0 = 100$

T_1 deposits 50 in X	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(X_1, 150)$	
$commit_1$	
	$commit_2$ (Serialization Error T_2 is rolled back)

$X_1 = 150$

- Variant: "**First-updater-wins**"
  - Check for concurrent updates when write occurs by locking item
    - – But lock should be held till all concurrent transactions have finished
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent

Benefits of Snapshot Isolation

- Reads are *never* blocked,
  - and also don’t block other transaction activities
- Performance similar to Read Committed
- Avoids several anomalies
  - **No dirty read**, i.e. no read of uncommitted data
  - **No lost update**
    - Lost update is the update overwritten by another transaction that did not see the update
  - **No non-repeatable read**
    - I.e., if read is executed again, it will see the same value
- Problems with Snapshot Isolation
  - Snapshot Isolation does **not always** give **serializable** executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated

Snapshot Isolation

- **Example of problem** with Snapshot Isolation
  - Initially A = 3 and B = 17
    - Serial execution: A = ??, B = ??
    - if both transactions start at the same time, with snapshot isolation: A = ?? , B = ??

T_i	T_j
read(A)	
read(B)	
	read(A)
	read(B)
A=B	
	B=A
write(A)	
	write(B)

- Called **Skew Write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1
    - Two transaction can both create order with same number
      - Is an example of **phantom phenomenon**

Snapshot Isolation Anomalies

- **SI breaks serializability** when transactions modify *different* items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when transactions conflict due to modifying different data, there is usually also a shared item they both modify, so SI will abort one of them
  - But problems do occur
    - Application developers should be careful about write skew
    - *Warning: Oracle & old versions of PostgreSQL provide "serializable" mode, which actually runs in SI mode.*
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details
- **Serializable snapshot isolation (SSI):** extension of snapshot isolation that ensures serializability

Working Around SI Anomalies

- Can work around SI anomalies for specific queries by using **select .. for update** (supported e.g. in Oracle)
  - Example
    - **select max**(orderno) **from** orders <u>**for update**</u>
    - read value into local variable maxorder
    - insert into orders (maxorder+1, …)
- **select for update (SFU) clause** treats all data read by the query as if it were also updated, preventing concurrent updates
- Can be added to queries to ensure serializability in many applications
  - Does not handle phantom phenomenon/predicate reads though

Weak Levels of Concurrency

Weak Levels of Consistency

- **Degree-two consistency**: differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]
- **Cursor stability**:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - **Serializable**: is the default
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even uncommitted data to be read
- In most database systems, read committed is the default consistency level
  - Can be changed as database configuration parameter, or per transaction
    - **set isolation level serializable**