CS3223 Lecture 9
Multiversion Concurrency Control

Multiversion Concurrency Control (MVCC)

T_1	T_2
$R_1(x)$	
$W_1(x)$	
	$R_2(x)$
	$W_2(y)$
$R_1(y)$	
$W_1(z)$	

Schedule S

T_1	T_2
$R_1(x)$	
$W_1(x)$	
$R_1(y)$	
	$R_2(x)$
	$W_2(y)$
$W_1(z)$	

Schedule S’

Multiversion Concurrency Control (MVCC)

- **Key idea**: maintain multiple versions of each object
  - $W_i(O)$ creates a new version of object O
  - $R_i(O)$ reads an appropriate version of O
- Advantages:
  - Read-only Xacts are not blocked by update Xacts
  - Update Xacts are not blocked by read-only Xacts
  - Read-only Xacts are never aborted
- Notation:
  - $W_i(x)$ creates a new version of x denoted by x_i
  - For each object x, its initial version is denoted by x_0

MVCC: Example 1

T_1	T_2	Comments
		$x_0 = 10$
	$R_2(x)$	10
$W_1(x)$		?

- In **2PL**, $W_1(x)$ will be blocked
- In **MVCC**,
  - T_1 creates a new version of x
  - Update Xacts are not blocked by read-only Xacts

MVCC: Example 2

T_1	T_2	Comments
		$x_0 = 10, y_0 = 20$
	$R_2(x)$	10
$W_1(y)$		$y_1 = 100$
	$R_2(y)$	?

- In **2PL**, $R_2(y)$ will be blocked
- In **MVCC**,
  - $W_1(y)$ creates a new version of y (with a value of 100)
  - $R_2(y)$ is not blocked
  - $R_2(y)$ returns 20 (the value of the version before $W_1(y)$)
  - Read-only Xacts are never blocked/aborted

Multiversion Schedules

- If there are multiple versions of an object x, a read action on x could return any version
- Thus, an interleaved execution could correspond to different multiversion schedules depending on the MVCC protocol
- **Example**: $R_1(x)$, $W_1(x)$, $R_2(x)$, $W_2(y)$, $R_1(y)$, $W_1(z)$

  S_1: $R_1(x_0)$, $W_1(x_1)$, $R_2(x_0)$, $W_2(y_2)$, $R_1(y_0)$, $W_1(z_1)$
  S_2: $R_1(x_0)$, $W_1(x_1)$, $R_2(x_0)$, $W_2(y_2)$, $R_1(y_2)$, $W_1(z_1)$
  S_3: $R_1(x_0)$, $W_1(x_1)$, $R_2(x_1)$, $W_2(y_2)$, $R_1(y_0)$, $W_1(z_1)$
  S_4: $R_1(x_0)$, $W_1(x_1)$, $R_2(x_1)$, $W_2(y_2)$, $R_1(y_2)$, $W_1(z_1)$

  - $R_1(x)$ returns x_0
  - $R_2(x)$ could return x_0 or x_1
  - $R_1(y)$ could return y_0 or y_2

Multiversion View Equivalence

- Two schedules, S and S', over the same set of transactions, are defined to be multiversion view equivalent ($S \equiv_{mv} S'$) if they have the same set of **read-from relationships**
  - i.e. $R_i(x_j)$ occurs in S iff $R_i(x_j)$ occurs in S'
- **Example**:

$$
\begin{array}{ll}
S_1: & R_3(x_0), W_3(x_3), Commit_3, W_1(x_1), Commit_1, R_2(x_1), W_2(y_2), Commit_2 \\
S_2: & R_3(x_0), W_3(x_3), Commit_3, W_1(x_1), R_2(x_3), Commit_1, W_2(y_2), Commit_2 \\
S_3: & W_1(x_1), Commit_1, R_2(x_1), R_3(x_0), W_2(y_2), W_3(x_3), Commit_3, Commit_2
\end{array}
$$

  - $S_1 \not\equiv_{mv} S_2$ because $R_2(x_1) \in S_1$ and $R_2(x_3) \in S_2$
  - $S_1 \equiv_{mv} S_3$

Monoversion Schedules

- A multiversion schedule S is called a monoversion schedule if each read action in S returns the most recently created object version
- **Example**: $R_1(x)$, $W_1(x)$, $R_2(x)$, $W_2(y)$, $R_1(y)$, $W_1(z)$

S_1: $R_1(x_0)$, $W_1(x_1)$, $R_2(x_0)$, $W_2(y_2)$, $R_1(y_0)$, $W_1(z_1)$
S_2: $R_1(x_0)$, $W_1(x_1)$, $R_2(x_0)$, $W_2(y_2)$, $R_1(y_2)$, $W_1(z_1)$
S_3: $R_1(x_0)$, $W_1(x_1)$, $R_2(x_1)$, $W_2(y_2)$, $R_1(y_0)$, $W_1(z_1)$
S_4: $R_1(x_0)$, $W_1(x_1)$, $R_2(x_1)$, $W_2(y_2)$, $R_1(y_2)$, $W_1(z_1)$

  - S_4 is a monoversion schedule
  - S_1, S_2, and S_3 are not monoversion schedules

Serial Monoversion Schedules

- A monoversion schedule is defined to be a serial monoversion schedule if it is also a serial schedule
- **Example**:

$$S_1: \quad R_1(x_0), W_1(x_1), R_2(x_1), W_2(y_2), R_1(y_2), W_1(z_1)$$
$$S_2: \quad R_1(x_0), W_1(x_1), R_1(y_0), W_1(z_1), R_2(x_1), W_2(y_2)$$

  - S_1 is a non-serial monoversion schedule
  - S_2 is a serial monoversion schedule

Multiversion View Serializability

A multiversion schedule *S* is defined to be multiversion view serializable schedule (MVSS) if there exists a **serial monoversion schedule** (over the same set of Xacts) that is multiversion view equivalent to *S*

MVSS: Example 1

- Consider schedule S:

  $W_1(x_1)$, $Commit_1$, $R_2(x_1)$, $R_3(x_0)$, $W_2(y_2)$, $W_3(x_3)$, $Commit_3$, $Commit_2$

- S is multiversion view serializable as $S \equiv_{mv} (T_3, T_1, T_2)$:

  $R_3(x_0)$, $W_3(x_3)$, $Commit_3$, $W_1(x_1)$, $Commit_1$, $R_2(x_1)$, $W_2(y_2)$, $Commit_2$

MVSS: Example 2

- Consider the following schedule S:

T_1	T_2
$R_1(x_0)$	
	$R_2(x_0)$
$R_1(y_0)$	
	$R_2(y_0)$
$W_1(x_1)$	
$Commit_1$	
	$W_2(y_2)$
	$Commit_2$

- S is not multiversion view serializable
- S is not multiversion view equivalent to any serial monoversion schedule
  - $R_1(x_0), R_1(y_0), W_1(x_1), C_1, R_2(x_1), R_2(y_0), W_2(y_2), C_2$
  - $R_2(x_0), R_2(y_0), W_2(y_2), C_2, R_1(x_0), R_1(y_2), W_1(x_1), C_1$

MVSS: Example 3

- Consider the following schedule S:

T_1	T_2	T_3
$R_1(y_0)$		
	$R_2(x_0)$	
$W_1(y_1)$		
$Commit_1$		
	$R_2(y_0)$	
	$W_2(x_2)$	
		$R_3(x_0)$
		$R_3(y_1)$
		$Commit_3$
	$Commit_2$	

- S is not multiversion view serializable
  - Suppose S' is a serial monoversion schedule where $S' \equiv_{mv} S$
  - T_3 must precede T_2 in S' due to $R_3(x_0)$ & $W_2(x_2)$
  - T_2 must precede T_1 in S' due to $R_2(y_0)$ & $W_1(y_1)$
  - T_1 must precede T_3 in S' due to $W_1(y_1)$ & $R_3(y_1)$

Multiversion View Serializability

- **Theorem 1**: A view serializable schedule (VSS) is also a multiversion view serializable schedule (MVSS)
- A MVSS is not necessarily VSS
- **Example**:

T_1: $R_1(x_0)$, $R_1(y_0)$, $Commit_1$

T_2: $W_2(x_2)$, $W_2(y_2)$, $Commit_2$,

  - The above schedule is multiversion view equivalent to the serial monoversion schedule (T_1, T_2)
  - However, the schedule is not a valid monoversion schedule (due to $W_2(y_2)$ & $R_1(y_0)$) and is therefore not VSS

MVCC Protocols

- Multiversion two-phase locking
- Multiversion timestamp ordering
- Snapshot isolation

Snapshot Isolation (SI)

- Widely used (e.g., Oracle, PostgreSQL, SQL Server, Sybase IQ)
- Each Xact T sees a snapshot of DB that consists of updates by Xacts that committed before T starts
- Each Xact T is associated with two timestamps:
  - start(T): the time that T starts
  - commit(T): the time that T commits

Concurrent Transactions

- Two Xacts T and T' are defined to be concurrent if they overlap
  - i.e., $[start(T), commit(T)] \cap [start(T'), commit(T')] \neq \emptyset$
- **Example**:

Timestamp	T_1	T_2	T_3
1	$R_1(B)$		
2		$R_2(A)$	
3	$W_1(B)$		
4	$Commit_1$		
5		$R_2(B)$	
6		$W_2(A)$	
7			$R_3(A)$
8			$R_3(B)$
9			$Commit_3$
10		$Commit_2$	

Snapshot Isolation (SI)

- $\mathbf{W_i(O)}$ creates a version of O denoted by O_i
- O_i is a more recent (or newer) version compared to O_j if $commit(T_i) > commit(T_j)$
- $\mathbf{R_i(O)}$ reads either its own update (if $W_i(O)$ precedes $R_i(O)$) or the latest version of O that is created by a Xact that committed before T_i started; i.e., If $R_i(O)$ returns O_j, then
  1. Either $j = i$ if $W_i(O)$ precedes $R_i(O)$;
  2. Or
     - 2.1 $commit(T_j) < start(T_i)$, and
     - 2.2 For every Xact T_k, $k \neq j$, that has created a version O_k of O, if $commit(T_k) < start(T_i)$, then $commit(T_k) < commit(T_j)$

Example

T_1	T_2	T_3	**Comments**
$R_1(x)$			x_0
$W_1(x)$			x_1
$R_1(y)$			y_0
	$R_2(x)$		x_0
$W_1(y)$			y_1
$Commit_1$			
	$R_2(y)$		y_0
	$W_2(x)$		x_2
		$R_3(x)$	x_1
		$R_3(y)$	y_1
		$W_3(y)$	y_3
		$R_3(y)$	y_3
		$Commit_3$	

Snapshot Isolation

- **Concurrent Update Property**: If multiple concurrent Xacts updated the same object, only one of Xacts is allowed to commit
- If not, the schedule may not be serializable
- **Example**: Consider the following schedule S

T_1:	$R_1(x_0)$		$W_1(x_1)$			$Commit_1$
T_2:		$R_2(x_0)$		$W_2(x_2)$	$Commit_2$	

  S is not serializable!
- Two approaches to enforce the concurrent update property:
  - First Committer Wins (FCW) Rule
  - First Updater Wins (FUW) Rule

First Committer Wins (FCW) Rule

- Before commiting a Xact T, the system checks if there exists a committed concurrent Xact T' that has updated some object that T has also updated
- If T' exists, then T aborts
- Otherwise, T commits
- **Example 1**:

T_1:	$R_1(x)$		$W_1(x)$			$Abort_1$
T_2:		$R_2(x)$		$W_2(x)$	$Commit_2$	

- **Example 2**:

T_1:	$R_1(x)$		$W_1(x)$		$Commit_1$	
T_2:		$R_2(x)$		$W_2(x)$		$Abort_2$

First Updater Wins (FUW) Rule

- Whenever a Xact T needs to update an object O, T requests for a X-lock on O
- If the X-lock is not held by any concurrent Xact, then
  - T is granted the X-lock on O
  - If O has been updated by any concurrent Xact, then T aborts
  - Otherwise, T proceeds with its execution
- Otherwise, if the X-lock is being held by some concurrent Xact T', then T waits until T' aborts or commits
  - If T' aborts, then
    - ⋆ Assume that T is granted the X-lock on O
    - ⋆ If O has been updated by any concurrent Xact, then T aborts
    - ⋆ Otherwise, T proceeds with its execution
  - If T' commits, then T is aborted
- When a Xact commits/aborts, it releases its X-lock(s)

FUW Rule: Example 1

T_1	T_2
Begin	
$R_1(O)$	
$X_1(O)$	
$W_1(O)$	
	Begin
	$R_2(O)$
Commits	
$U_1(O)$	
	$X_2(O)$
	T_2 is aborted

FUW Rule: Example 2

T_1	T_2
Begin	
$R_1(O)$	
$X_1(O)$	
$W_1(O)$	
	Begin
	$R_2(O)$
	$X_2(O)$
	T_2 is blocked by T_1
Aborts	
$U_1(O)$	
	X-lock on O is granted to T_2
	$W_2(O)$
	Commits
	$U_2(O)$

Garbage Collection

- A version O_i of object O may be deleted if there exists a newer version O_j (i.e., $commit(T_i) < commit(T_j)$) such that for every active Xact T_k that started after the commit of T_i (i.e., $commit(T_i) < start(T_k)$), we have $commit(T_j) < start(T_k)$
- **Example**:

$$W_1(x_1),\ C_1,\ W_2(x_2),\ C_2,\ W_4(x_4),\ R_3(y_0),\ C_4,\ W_5(x_5),\ C_5,\ R_6(z_0)$$

- Active transactions: T_3 & T_6
- Versions that can be deleted: x_1 & x_4

Snapshot Isolation Tradeoffs

- Performance of SI often similar to Read Committed
- Unlike Read Committed, SI does not suffer from lost update or unrepeatable read anomalies
- But SI is vulnerable to some non-serializable executions
  - Write Skew Anomaly
  - Read-Only Transaction Anomaly
- Snapshot isolation does not guarantee serializability

Write Skew Anomaly

T_1	T_2
$R_1(x_0)$	
	$R_2(x_0)$
$R_1(y_0)$	
	$R_2(y_0)$
$W_1(x_1)$	
$Commit_1$	
	$W_2(y_2)$
	$Commit_2$

The above is a SI schedule that is not a MVSS

Read-Only Transaction Anomaly

T_1	T_2	T_3
$R_1(y_0)$		
	$R_2(x_0)$	
$W_1(y_1)$		
$Commit_1$		
	$R_2(y_0)$	
	$W_2(x_2)$	
		$R_3(x_0)$
		$R_3(y_1)$
		$Commit_3$
	$Commit_2$	

The above is a SI schedule that is not a MVSS

Serializable Snapshot Isolation (SSI) Protocol

- This is a stronger protocol that guarantees serializable SI schedules
- Here's an approach to guarantee serializable SI schedules:
  - Keep track of rw dependencies among concurrent Xacts
  - Detect the formation of T_j involving two rw dependencies:

$$T_i \xrightarrow{\text{rw}} T_j \xrightarrow{\text{rw}} T_k$$

  - Once detected, abort one of T_i, T_j, or T_k
  - May result in unnecessary rollbacks due to false positives of SI anomalies

Transactional Dependencies

- ww dependency from T_1 to T_2
  - T_1 writes a version of some data item x, and
  - T_2 later writes the immediate successor version of x
- wr dependency from T_1 to T_2
  - T_1 writes a version of some data item x, and
  - T_2 reads this version of x
- rw dependency from T_1 to T_2
  - T_1 reads a version or some data item x, and
  - T_2 later creates the immediate successor version of x

x_j is the immediate successor of x_i if (1) T_i commits before T_j, and (2) no transaction that commits between T_i's and T_j's commits produces a version of x

Dependency Serialization Graph (DSG)

- Consider a schedule S consisting of a set of committed transactions $\{T_1, \cdots, T_k\}$
- DSG(S) is an edge-labelled directed graph (V, E)
- V represents transactions $\{T_1, \cdots, T_k\}$
- E represents transactional dependencies
  - $T_i \stackrel{ww}{\rightarrow} T_j$
  - $T_i \stackrel{wr}{\rightarrow} T_j$
  - $T_i \stackrel{rw}{\rightarrow} T_j$
- Edge types:
  - $\dashrightarrow$ if transaction pair is concurrent
  - $\longrightarrow$ if transaction pair is non-concurrent

DSG: Example

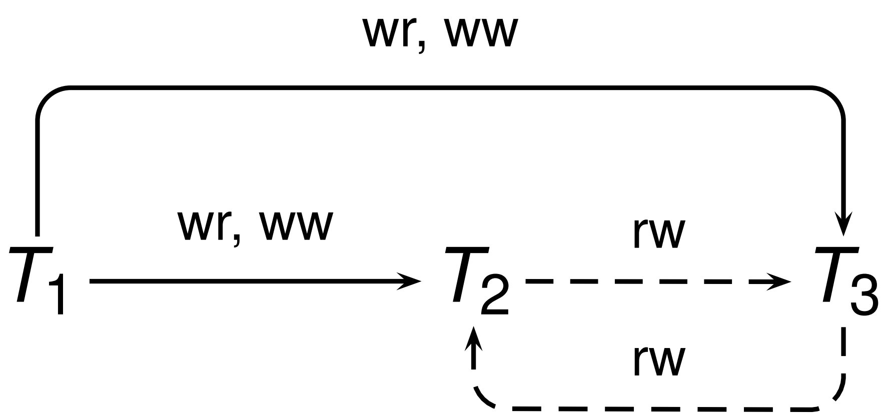
Schedule S:

$W_1(x)$, $W_1(y)$, $W_1(z)$, C_1,

$R_2(x)$, $W_2(y)$, C_2,

$W_3(x)$, $R_3(y)$, C_3

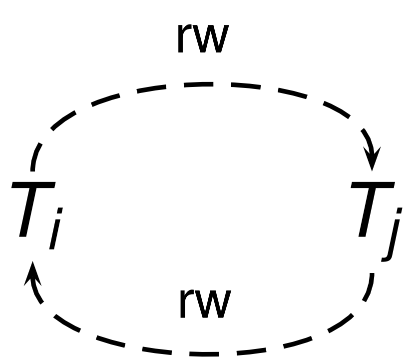
DSG(S):

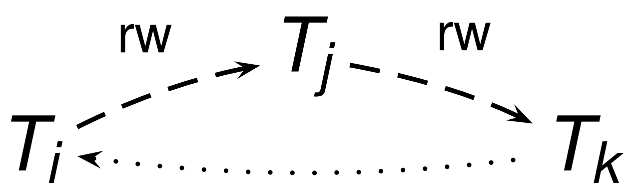


Non-MVSS SI schedules

Theorem 2: If S is a SI schedule that is not MVSS, then

1. There is at least one cycle in $DSG(S)$, and
2. For each cycle in $DSG(S)$, there exists three transactions, T_i, T_j, and T_k such that
   - T_i & T_k are possibly the same transaction,
   - T_i & T_j are concurrent with an edge $T_i \stackrel{rw}{\rightarrow} T_j$, and
   - T_j & T_k are concurrent with an edge $T_j \stackrel{rw}{\rightarrow} T_k$.





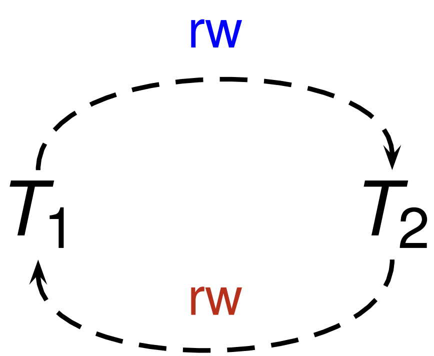
SI Anomalies Revisited

Schedule S_1: Write Skew Anomaly

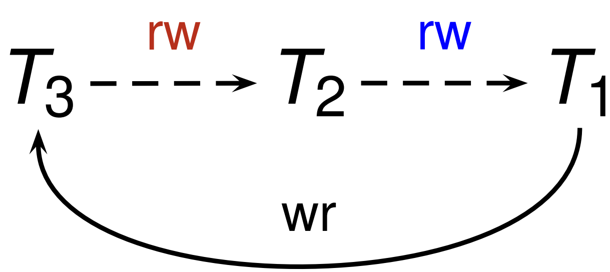
$R_1(a)$, $R_2(a)$, $R_1(b)$, $R_2(b)$, $W_1(a)$, C_1, $W_2(b)$, C_2

Schedule S_2: Read-only Xact Anomaly

$R_1(b)$, $R_2(a)$, $W_1(b)$, C_1, $R_2(b)$, $W_2(a)$, $R_3(a)$, $R_3(b)$, C_3, C_2



DSG(S_1)



DSG(S_2)