CS4224/CS5424 Lecture 6
Review of Concurrency Control

Transactions

- A transaction is an abstraction representing a logical unit of work
- **Example**: `Transfer` (x, y, amount)
  - transaction to transfer `amount` from account `x` to account `y`

```
BEGIN TRANSACTION;
SELECT balance INTO :Balx FROM Account WHERE accountId = :x;

SELECT balance INTO :Baly FROM Account WHERE accountId = :y;

If (:Balx < amount) then ABORT;

UPDATE Account SET balance = :Baly + :amount WHERE accountId = :y;

UPDATE Account SET balance = :Balx - :amount WHERE accountId = :x;

COMMIT;
```

Transaction Management

- Ensures four properties of transactions (Xacts) to maintain data in the face of concurrent access and system failures
  1. **Atomicity**: Either all or none of the actions in Xact happen
  2. **Consistency**: If each Xact is consistent, and the DB starts consistent, the DB ends up consistent
  3. **Isolation**: Execution of one Xact is isolated from other Xacts
  4. **Durability**: If a Xact commits, its effects persist
- The concurrency control manager component ensures isolation
- The recovery manager component ensures atomicity and durability

Transactions

- A transaction (Xact) T_i can be viewed as a sequence of actions:
  - $R_i(O)$ = T_i reads an object O
  - $W_i(O)$ = T_i writes an object O
  - $Commit_i$ = T_i terminates successfully
  - $Abort_i$ = T_i terminates unsuccessfully
- Each Xact must end with either a commit or an abort action
- An active Xact is a Xact that is still in progress (i.e., has not yet terminated)

Transactions

- $R(x), R(y), W(y), W(x)$, Commit
- $R(x), R(y)$, Abort

```
BEGIN TRANSACTION;
SELECT balance INTO :Balx FROM Account WHERE accountId = :x;

SELECT balance INTO :Baly FROM Account WHERE accountId = :y;

If (:Balx < amount) then ABORT;

UPDATE Account SET balance = :Baly + :amount WHERE accountId = :y;

UPDATE Account SET balance = :Balx - :amount WHERE accountId = :x;

COMMIT;
```

Transaction Schedules

- Schedule = a list of actions from a set of Xacts, where the order of the actions within each Xact is preserved
- **Example**: Consider two Xacts T_1 and T_2:
  - T_1: R_1(A), W_1(A), R_1(B), W_1(B), $Commit_1$
  - T_2: R_2(A), W_2(A), R_2(B), W_2(B), $Commit_2$
- Some schedules of T_1 and T_2:

  S_1: R_1(A),W_1(A),R_1(B),W_1(B),$Commit_1$, R_2(A),W_2(A),R_2(B),W_2(B),$Commit_2$

  S_2: R_2(A),W_2(A),R_2(B),W_2(B),$Commit_2$, R_1(A),W_1(A),R_1(B),W_1(B),$Commit_1$

  S_3: R_1(A),W_1(A),R_2(A),W_2(A),R_1(B), W_1(B),$Commit_1$,R_2(B),W_2(B),$Commit_2$
- A serial schedule is a schedule where the actions of Xacts are not interleaved

Read From & Final Write

- We say that T_j reads O from T_i in a schedule S if the last write action on O before $R_j(O)$ in S is $W_i(O)$
- We say that T_j reads from T_i if T_j has read some object from T_i
- We say that T_i performs the final write on O in a schedule S if the last write action on O in S is $W_i(O)$
- **Example**: Consider the following schedule:

$$R_1(\text{A}), W_1(\text{A}), R_2(\text{A}), W_2(\text{A}), R_1(\text{B}), W_1(\text{B}), Commit_1, R_2(\text{B}), W_2(\text{B}), Commit_2$$

  - T_1 reads A from the initial database (or T_1 reads A from T_0)
    - ★ Assume that initial database was created by dummy Xact T_0
  - T_2 reads A from T_1
  - T_1 reads B from T_0
  - T_2 reads B from T_1
  - T_2 performs the final write on A
  - T_2 performs the final write on B

View Serializable Schedules

- Two schedules S and S' (over the same set of Xacts) are view equivalent (denoted by $S \equiv_v S'$) if they satisfy all the following conditions:
  1. If T_i reads A from T_j in S, then T_i must also read A from T_j in S'
  2. For each data object A, the Xact (if any) that performs the final write on A in S must also perform the final write on A in S'

- **Example**:
  - S_3: R_1(x), R_1(y), W_1(y), R_2(y), R_2(x), W_1(x), W_2(x), W_2(y)
  - S_5: R_1(x), R_2(y), R_1(y), W_1(y), R_2(x), W_2(x), W_2(y), W_1(x)
  - S_6: R_1(x), R_2(y), R_1(y), R_2(x), W_1(y), W_2(x), W_1(x), W_2(y)
  - $S_3 \not\equiv_v S_5$ as T_2 reads y from T_1 in S_3 but T_2 reads y from T_0 in S_5
  - Similarly, $S_3 \not\equiv_v S_6$
  - $S_5 \equiv_v S_6$

View Serializable Schedules (cont.)

- A schedule S is a view serializable schedule (VSS) if S is view equivalent to some serial schedule over the same set of Xacts
- **Example**:
  - Consider two Xacts T_1 and T_2:
    - ★ T_1: R_1(x), R_1(y), W_1(y), W_1(x)
    - ★ T_2: R_2(y), R_2(x), W_2(x), W_2(y)
  - Serial schedules over $\{T_1, T_2\}$:
    - ★ S_1: R_1(x), R_1(y), W_1(y), W_1(x), R_2(y), R_2(x), W_2(x), W_2(y)
    - ★ S_2: R_2(y), R_2(x), W_2(x), W_2(y), R_1(x), R_1(y), W_1(y), W_1(x)
  - S_3 is not view serializable; S_4 is view serializable
    - ★ S_3: R_1(x), R_1(y), W_1(y), R_2(y), R_2(x), W_1(x), W_2(x), W_2(y)
    - ★ S_4: R_1(x), R_1(y), W_1(y), R_2(y), W_1(x), R_2(x), W_2(x), W_2(y)

Conflicting Actions

- Two actions on the same object conflict if
  1. at least one of them is a write action, and
  2. the actions are from different Xacts
- **Examples**:
  - $R_1(x)$ and $R_2(x)$ do not conflict
  - $R_1(x)$ and $W_1(x)$ do not conflict
  - $W_1(x)$ and $R_2(y)$ do not conflict
  - $W_1(x)$ and $R_2(x)$ conflict
  - $W_1(x)$ and $W_2(x)$ conflict

Conflict Serializable Schedules

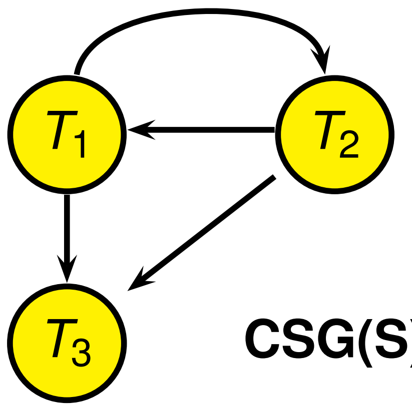
- Two schedules S & S' (over the same set of Xacts) are said to be conflict equivalent (denoted by $S \equiv_c S'$) if they order every pair of conflicting actions of two committed Xacts in the same way
- **Example**:
  - S_3: R_1(x), R_1(y), W_1(y), R_2(y), R_2(x), W_1(x), W_2(x), W_2(y)
  - S_5: R_1(x), R_2(y), R_1(y), W_1(y), R_2(x), W_2(x), W_2(y), W_1(x)
  - S_6: R_1(x), R_2(y), R_1(y), R_2(x), W_1(y), W_2(x), W_1(x), W_2(y)
  - $S_3 \not\equiv_c S_5$ as $W_1(y)$ precedes $R_2(y)$ in S_3 but not in S_5
  - Similarly, $S_3 \not\equiv_c S_6$
  - $S_5 \equiv_c S_6$

Conflict Serializable Schedules (cont.)

- A schedule is a conflict serializable schedule (CSS) if it is conflict equivalent to a serial schedule over the same set of Xacts
- **Example**:
  - Consider two Xacts T_1 and T_2:
    - ★ T_1: R_1(x), R_1(y), W_1(y), W_1(x)
    - ★ T_2: R_2(y), R_2(x), W_2(x), W_2(y)
  - Serial schedules over $\{T_1, T_2\}$:
    - ★ S_1: R_1(x), R_1(y), W_1(y), W_1(x), R_2(y), R_2(x), W_2(x), W_2(y)
    - ★ S_2: R_2(y), R_2(x), W_2(x), W_2(y), R_1(x), R_1(y), W_1(y), W_1(x)
  - S_3 is not conflict serializable; S_4 is conflict serializable
    - ★ S_3: R_1(x), R_1(y), W_1(y), R_2(y), R_2(x), W_1(x), W_2(x), W_2(y)
    - ★ S_4: R_1(x), R_1(y), W_1(y), R_2(y), W_1(x), R_2(x), W_2(x), W_2(y)

Testing for Conflict Serializability

- A conflict serializability graph for a schedule S (denoted by $CSG(S)$) is a directed graph $G = (V, E)$ such that
  - V contains a node for each committed Xact in S
  - E contains (T_i, T_j) if an action in T_i precedes and conflicts with one of T_j's actions
- **Example**: Conflict serializability graph for schedule
  R_1(A), W_2(A),$Commit_2$, W_1(A),$Commit_1$, W_3(A),$Commit_3$



CSG(S)

- **Theorem 1**: A schedule is conflict serializable iff its conflict serializability graph is acyclic
- **Theorem 2**: A schedule that is conflict serializable is also view serializable

Blind Writes

- A write on object O by T_i is call a blind write if T_i did not read O prior to the write
  - Consider the schedule: $R_1(x)$, $W_2(y)$, $W_1(x)$
  - $W_2(y)$ is a blind write
  - $W_1(x)$ is a non-blind write
- **Theorem 3**: If S is view serializable and S has no blind writes, then S is also conflict serializable

Recoverable Schedules

T_1	T_2
$W_1(x)$	
	$R_2(x)$
	$W_2(y)$
	$Commit_2$

Non-Recoverable
Schedule

- A schedule S is said to be a recoverable schedule if for every Xact T that commits in S, T must commit after T' if T reads from T'

Lock-Based Concurrency Control

- Each Xact needs to request for an appropriate lock on an object before the Xact can access the object
- Locking modes
  - Shared (S) locks for reading objects
  - Exclusive (X) locks for writing objects
- Lock compatibility:

Lock Requested	Lock Held		
	-	S	X
S	$\surd$	$\surd$	$\times$
X	$\surd$	$\times$	$\times$

$\surd$: Compatible
Lock request is granted

$\times$: Incompatible
Lock request is blocked

Lock-Based Concurrency Control (cont.)

1. To **read an object** *O*, a Xact must request for a shared/exclusive lock on *O*
2. To **update an object** *O*, a Xact must request for an exclusive lock on *O*
3. A **lock request is granted** on *O* if the requesting lock mode is compatible with the lock modes of existing locks on *O*
4. If *T*'s **lock request is not granted** on *O*, *T* becomes blocked: its execution is suspended & *T* is added to *O*'s request queue
5. When a **lock is released** on *O*, the lock manager checks the request of the first Xact *T* in the request queue for *O*. If *T*'s request can be granted, *T* acquires its lock on *O* and resumes execution after its removal from the queue
6. When a Xact **commits/aborts**, all its locks are released & T is removed from any request queue it's in

Two Phase Locking (2PL) Protocol

- 2PL Protocol:
  1. To read an object O, a Xact must hold a S-lock or X-lock on O
  2. To write to an object O, a Xact must hold a X-lock on O
  3. Once a Xact releases a lock, the Xact can't request any more locks
- Xacts using 2PL can be characterized into two phases:
  - Growing phase: before releasing 1^{st} lock
  - Shrinking phase: after releasing 1^{st} lock
- **Theorem 4**: 2PL schedules are conflict serializable

Strict Two Phase Locking (strict 2PL) Protocol

- 2PL Protocol:
  1. To read an object O, a Xact must hold a S-lock or X-lock on O
  2. To write to an object O, a Xact must hold a X-lock on O
  3. Once a Xact releases a lock, the Xact can't request any more locks
- **Theorem 4**: 2PL schedules are conflict serializable
- Strict 2PL Protocol:
  1. To read an object O, a Xact must hold a S-lock or X-lock on O
  2. To write to an object O, a Xact must hold a X-lock on O
  3. A Xact must hold on to locks until Xact commits or aborts
- **Theorem 5**: Strict 2PL schedules are strict & conflict serializable

Lock Management

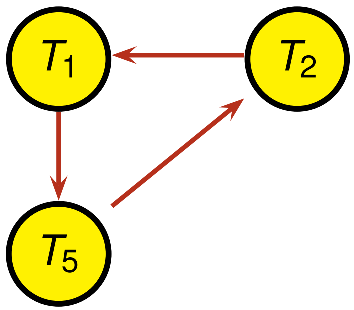
- Handling deadlocks
- Lock conversion

Deadlocks

- Deadlock: cycle of Xacts waiting for locks to be released by each other
- **Example**:
  T_1 requests X-lock on A and is granted;
  T_2 requests X-lock on B and is granted;
  T_1 requests X-lock on B and is blocked;
  T_2 requests X-lock on A and is blocked;
- Dealing with deadlocks:
  - deadlock detection
  - deadlock prevention

How to Detect Deadlocks?

- Waits-for graph (WFG)
  - Nodes represent active Xacts
  - Add an edge $T_i \rightarrow T_j$ if T_i is waiting for T_j to release a lock



- Lock manager
  - adds an edge when it queues a lock request
  - updates edges when it grants a lock request
- Deadlock is detected if WFG has a cycle
- Breaks a deadlock by aborting a Xact in cycle
- Alternative to WFG: timeout mechanism

How to Prevent Deadlocks?

- Assume older Xacts have higher priority than younger Xacts
  - Each Xact is assigned a timestamp when it starts
  - An older Xact has a smaller timestamp
- Suppose T_i requests for a lock that conflicts with a lock held by T_j
- Two possible deadlock prevention policies:
  - Wait-die policy: lower-priority Xacts never wait for higher-priority Xacts
  - Wound-wait policy: higher-priority Xacts never wait for lower-priority Xacts

Prevention Policy	T_i has higher priority	T_i has lower priority
Wait-die	T_i waits for T_j	T_i aborts
Wound-wait	T_j aborts	T_i waits for T_j

- Wait-die policy
  - non-preemptive: only a Xact requesting for a lock can get aborted
  - a younger Xact may get repeatedly aborted
  - a Xact that has all the locks it needs is never aborted
- Wound-wait policy
  - preemptive
- To avoid starvation, a restarted Xact must use its original timestamp!

Lock Conversion

- Consider two Xacts:

  T_1: $R_1(A)$, $R_1(B)$, $W_1(A)$
  T_2: $R_2(A)$, $R_2(B)$

- Since T_1 needs to both read & update A, T_1 acquires an exclusive lock to read A
- All possible schedules are serial

  S_1: $R_1(A)$, $R_1(B)$, $W_1(A)$, $R_2(A)$, $R_2(B)$
  S_2: $R_2(A)$, $R_2(B)$, $R_1(A)$, $R_1(B)$, $W_1(A)$

- Increase concurrency by allowing **lock conversions**
- Two types of lock conversions
  - T_i upgrades its S-lock on object A to X-lock
  - T_i downgrades its X-lock on object A to S-lock

Lock Conversion (cont.)

- Interleaved executions become possible with lock upgrading:

  S_3: $R_1(A)$, $R_2(A)$, $R_2(B)$, $R_1(B)$, $W_1(A)$
  S_4: $R_1(A)$, $R_1(B)$, $R_2(A)$, $R_2(B)$, $W_1(A)$
  S_5: $R_1(A)$, $R_2(A)$, $R_1(B)$, $R_2(B)$, $W_1(A)$

- **Lock upgrade** on object A
  - Upgrade request is blocked if another Xact is holding a shared lock on A
  - Upgrade request is allowed if T_i has not released any lock
- **Lock downgrade** on object A
  - Downgrade request is allowed if
    1. T_i has not modified A, and
    2. T_i has not released any lock

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

$\mathbf{T_1}$	$\mathbf{T_2}$	**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**: $R_1(x), W_1(x), R_2(x), W_2(y), R_1(y), W_1(z)$

$$\begin{array}{ll} S_1: & R_1(x_0), W_1(x_1), R_2(x_0), W_2(y_2), R_1(y_0), W_1(z_1) \\ S_2: & R_1(x_0), W_1(x_1), R_2(x_0), W_2(y_2), R_1(y_2), W_1(z_1) \\ S_3: & R_1(x_0), W_1(x_1), R_2(x_1), W_2(y_2), R_1(y_0), W_1(z_1) \\ S_4: & R_1(x_0), W_1(x_1), R_2(x_1), W_2(y_2), R_1(y_2), W_1(z_1) \end{array}$$

  - $R_1(x)$ 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**: $R_1(x), W_1(x), R_2(x), W_2(y), R_1(y), W_1(z)$

$$
\begin{array}{ll}
S_1: & R_1(x_0), W_1(x_1), R_2(x_0), W_2(y_2), R_1(y_0), W_1(z_1) \\
S_2: & R_1(x_0), W_1(x_1), R_2(x_0), W_2(y_2), R_1(y_2), W_1(z_1) \\
S_3: & R_1(x_0), W_1(x_1), R_2(x_1), W_2(y_2), R_1(y_0), W_1(z_1) \\
S_4: & R_1(x_0), W_1(x_1), R_2(x_1), W_2(y_2), R_1(y_2), W_1(z_1)
\end{array}
$$

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

$$\begin{array}{llll} T_1: & R_1(x_0), & & R_1(y_0),\ Commit_1 \\ T_2: & & W_2(x_2),\ W_2(y_2),\ Commit_2, & \end{array}$$

  - 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)

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
- A schedule S is defined to be a serializable snapshot isolation (SSI) schedule if S is produced by the snapshot isolation protocol (i.e., S is a SI schedule) and S is MVSS

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,					
		$W_4(x_4)$,		C_4,		
			$R_3(y_0)$,			
					$W_5(x_5)$, C_5,	
						$R_6(z_0)$

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