CS3223 Lecture 7
Transaction Management

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

Interleaved Transaction Executions

T_1: Transfer(x,y,100) T_2: Transfer(y,x,100)

T_1	T_2	**Comments**
		$x = 400, \; y = 100$
$R_1(x)$		400
$R_1(y)$		100
$y = y + 100$		
$W_1(y)$		$y = 200$
	$R_2(y)$	200
	$R_2(x)$	400
$x = x - 100$		
$W_1(x)$		$x = 300$
	$x = x + 100$	
	$W_2(x)$	$x = 500$
	$y = y - 100$	
	$W_2(y)$	$y = 100$

Initial database state: x = 400, y = 100

Final database state: x = 500, y = 100

Correctness of Interleaved Xact Executions

An interleaved Xact execution schedule is **correct** if it is "equivalent" to some serial schedule over the same set of Xacts

Execution Schedule	**Serial Schedules** S_1	S_2
$R_1(x, 400)$	$R_1(x, 400)$	$R_2(y, 100)$
$R_1(y, 100)$	$R_1(y, 100)$	$R_2(x, 400)$
$W_1(y, 200)$	$W_1(y, 200)$	$W_2(x, 500)$
$R_2(y, 200)$	$W_1(x, 300)$	$W_2(y, 0)$
$R_2(x, 400)$	$R_2(y, 200)$	$R_1(x, 500)$
$W_1(x, 300)$	$R_2(x, 300)$	$R_1(y, 0)$
$W_2(x, 500)$	$W_2(x, 400)$	$W_1(y, 100)$
$W_2(y, 100)$	$W_2(y, 100)$	$W_1(x, 400)$

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: $R_1(\text{x})$, $R_1(\text{y})$, $W_1(\text{y})$, $W_1(\text{x})$
    - T_2: $R_2(\text{y})$, $R_2(\text{x})$, $W_2(\text{x})$, $W_2(\text{y})$
  - Serial schedules over $\{T_1, T_2\}$:
    - S_1: $R_1(\text{x})$, $R_1(\text{y})$, $W_1(\text{y})$, $W_1(\text{x})$, $R_2(\text{y})$, $R_2(\text{x})$, $W_2(\text{x})$, $W_2(\text{y})$
    - S_2: $R_2(\text{y})$, $R_2(\text{x})$, $W_2(\text{x})$, $W_2(\text{y})$, $R_1(\text{x})$, $R_1(\text{y})$, $W_1(\text{y})$, $W_1(\text{x})$
  - S_3 is not view serializable; S_4 is view serializable
    - S_3: $R_1(\text{x})$, $R_1(\text{y})$, $W_1(\text{y})$, $R_2(\text{y})$, $R_2(\text{x})$, $W_1(\text{x})$, $W_2(\text{x})$, $W_2(\text{y})$
    - S_4: $R_1(\text{x})$, $R_1(\text{y})$, $W_1(\text{y})$, $R_2(\text{y})$, $W_1(\text{x})$, $R_2(\text{x})$, $W_2(\text{x})$, $W_2(\text{y})$

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

Anomalies with Interleaved Xact Executions

- Anomalies can arise due to conflicting actions:
  1. Dirty read problem (due to WR conflicts)
     - ★ T_2 reads an object that has been modified by T_1 and T_1 has not yet committed
     - ★ T_2 could see an inconsistent DB state!
  2. Unrepeatable read problem (due to RW conflicts)
     - ★ T_2 updates an object that T_1 has previously read and T_2 commits while T_1 is still in progress
     - ★ T_1 could get a different value if it reads the object again!
  3. Lost update problem (due to WW conflicts)
     - ★ T_2 overwrites the value of an object that has been modified by T_1 while T_1 is still in progress
     - ★ T_1's update is lost!

Dirty Read Problem: Example

T_1	T_2	**Comments**
		Initial DB state: $x = 100$
R(x)		100
$x = x + 20$		
W(x)		$x = 120$
	R(x)	120
	$x = x \times 2$	
Abort		
	W(x)	$x = 240$

- For every serial schedule over $\{T_1, T_2\}$, the final value of x is 200

Unrepeatable Read Problem: Example

T_1	T_2	**Comments**
		Initial DB state: $x = 100$
R(x)		100
	R(x)	100
	$x = x - 20$	
	W(x)	$x = 80$
	Commit	
R(x)		80

- For every serial schedule over $\{T_1, T_2\}$, both values read by T_1 are the same

Lost Update Problem: Example

T_1	T_2	**Comments**
		Initial DB state: $x = 100$
R(x)		100
	R(x)	100
$x = x + 20$		
	$x = x \times 2$	
W(x)		$x = 120$
	W(x)	$x = 200$

- For serial schedule (T_1, T_2), the final value of x is 240
- For serial schedule (T_2, T_1), the final value of x is 220

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

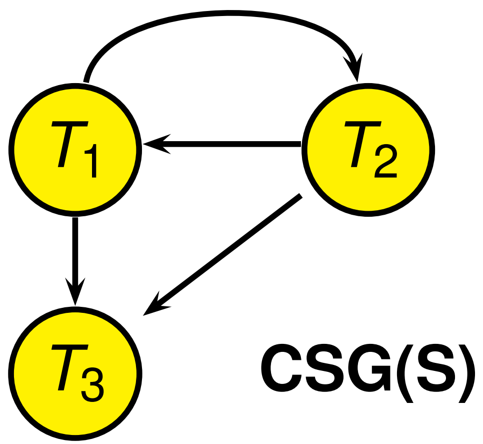
- **Example**:
  - S_3: R_1(x), R_1(y), W_1(y), R_2(y), R_2(x), W_1(x), W_2(x), W_2(y)
  - S_5: R_1(x), R_2(y), R_1(y), W_1(y), R_2(x), W_2(x), W_2(y), W_1(x)
  - S_6: R_1(x), R_2(y), R_1(y), R_2(x), W_1(y), W_2(x), W_1(x), W_2(y)
  - $S_3 \not\equiv_c S_5$ as $W_1(y)$ precedes $R_2(y)$ in S_3 but not 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
  $R_1(A)$, $W_2(A)$, $Commit_2$, $W_1(A)$, $Commit_1$, $W_3(A)$, $Commit_3$



- **Theorem 1**: A schedule is conflict serializable iff its conflict serializability graph is acyclic

Conflict Serializable Schedules

- **Theorem 2**: A schedule that is conflict serializable is also view serializable
- For convenience, we use serializable to mean conflict 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

Cascading Aborts

- For correctness, if T_i has read from T_j, then T_i must abort if T_j aborts

$\mathbf{T_1}$	$\mathbf{T_2}$
$W_1(x)$	
	$R_2(x)$
	$W_2(y)$
$Abort_1$	

$\mathbf{T_1}$	$\mathbf{T_2}$
$W_1(x)$	
	$R_2(x)$
	$W_2(y)$
$Abort_1$	
	$Abort_2$

- We say that T_1's abort is cascaded to T_2
- Recursive aborting process is known as cascading abort

Recoverable Schedules

$\mathbf{T_1}$	$\mathbf{T_2}$
$W_1(x)$	
	$R_2(x)$
	$W_2(y)$
	$Commit_2$

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'

Cascadeless Schedules

- While recoverable schedules guarantee that committed Xacts will not be aborted, cascading aborts of active Xacts are possible
  - **Example**: if T_i reads from T_j and T_j aborts, T_i must also abort
- Cascading aborts are undesirable because of the cost of bookkeeping to identify them and the performance penalty incurred
- To avoid cascading aborts (or to be cascadeless), DBMS must permit reads only from committed Xacts
- A schedule S is a cascadeless schedule if whenever T_i reads from T_j in S, $Commit_j$ must precede this read action
- **Theorem 4**: A cascadeless schedule is also a recoverable schedule

Recovery using Before-Images

- An efficient approach to undo the actions of aborted Xacts is to restore *before-images* for writes
- **Example**: Consider the following schedule S:

  $$W_1(A),\ W_2(A),\ Abort_2$$

  where T_1 updates A to be 100 and T_2 updates A to be 200
  - Assume that the initial value of A is 50
  - Before performing $W_1(A)$, its before-image "A=50" is logged
  - Before performing $W_2(A)$, its before-image "A=100" is logged
  - To recover from $Abort_2$, $W_2(A)$ is undone by restoring the before-image of A (i.e., the value of A is restored to 100)

Recovery using Before-Images (cont.)

- However, before-image recovery doesn't always work!
- **Example**:

$$W_1(A),\ W_2(A),\ Abort_1$$

  - Here, undoing $W_1(A)$ by restoring A to its before-image of 50 is incorrect!

Strict Schedules

- To enable the use of before-images for recovery, we use *strict schedules*
- A schedule S is a strict schedule if for every $W_i(O)$ in S, O is not read or written by another Xact until T_i either aborts or commits
- **Example**:

$$S: \quad W_1(A), W_2(A), Abort_2$$
$$S': \quad W_1(A), W_2(A), Abort_1$$

  Both S and S' are not strict schedules
- Performance Tradeoff:
  - recovery (using before-images) is more efficient
  - concurrent executions become more restrictive
- **Theorem 5**: A strict schedule is also a cascadeless schedule