Lecture-18: Transaction Isolation

CS211 - Introduction to Database

Serializability in the real world

- **Serializability** is a useful concept which ensures that **concurrent transaction executions maintain consistency**.
- However, the **protocols implementing serializability** may **allow too little concurrency** for certain applications.
- In some practical applications, **weaker levels of consistency** are used.

- Online shop
  - An item - In stock when browsing, and added to cart
  - Go through the checkout process – no longer available
- Seat selection for air travel
  - It is possible to enforce serializability by allowing only one traveler to do seat selection for a particular flight at a time
  - However … a traveler who takes a long time to make a choice could cause serious problems for other travelers

Weaker level consistency

1. Places **additional burdens on programmers** for ensuring database correctness.

2. For the **benefit of long transactions** whose results do not need to be precise.

3. **Avoid causing delays** due to transactions waiting for others to complete.

Recoverable Schedules

We need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** —For each pair of transactions T_i and T_j such that T_j ***reads*** a data item previously ***written*** by T_i , the ***commit*** operation of T_i appears ***before*** the ***commit operation*** of T_j .
- The following schedule is **non-recoverable** if T_9 commits immediately after *read(A)*.

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- If T_8 should abort, T_9 is already committed and cannot be aborted.
- Hence, DBMS must ensure that such schedules are not permitted.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks.

- Consider the following schedule where none of the transactions has yet committed (***so the schedule is recoverable***).

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

If T_{10} fails, T_{11} and T_{12} must also be rolled back

- Cascading rollback can lead to the undoing of a significant amount of work.

Cascading Rollbacks

- **Cascadeless schedules** —For each pair of transactions T_i and T_j such that T_j ***reads*** a data item previously ***written*** by T_i, the ***commit*** operation of T_i appears before the ***read operation*** of T_j.

- Every **Cascadeless** schedule is also **Recoverable**.

It is desirable to restrict the schedules to those that are Cascadeless.

Isolation levels

Concurrency-control protocols allow concurrent schedules, but **ensure** that the schedules are **conflict serializable**, **recoverable** ,and **cascadeless**.

The **isolation levels** specified by the SQL standard are as follows:

1. **Serializable**
   - usually ensures serializable execution. It is the highest isolation level allowed by SQL.
2. **Repeatable read**
   - allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it.
3. **Read committed**
   - allows only committed data to be read, but does not require repeatable reads.
4. **Read uncommitted (dirty read)**
   - allows uncommitted data to be read. It is the lowest isolation level allowed by SQL.

Dirty Writes

- All the isolation levels additionally **disallow dirty writes.**

Dirty write: write to a data item that has already been written by another transaction that has not yet committed or aborted.

T_1

Update account Set amount = amount + 1000;

T_2

Update account Set amount = amount + 2000;

Commit;

Commit;

Repeatable Read

Dirty read is not allowed

Start transaction T_1 ;

insert into TEST values(10);

Start transaction T_2 ;

.

.

Commit;

Select * from TEST;

Start transaction T_3 ;

Select * from TEST;

Commit;

insert into TEST values(20);

Commit;

Allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it

Read Committed

Dirty read is not allowed

Allows only committed data to be read, but does not require repeatable reads

Start transaction T_1 ;

insert into TEST values(10);

Start transaction T_2 ;

.

.

Commit;

Select * from TEST;

.

.

Start transaction T_3 ;

insert into TEST values(20);

Commit;

.

Select * from TEST;

Commit;

Read Uncommitted

Dirty read is allowed

Start transaction T_1 ;

insert into TEST values(10);

Start transaction T_2 ;

Select * from TEST;

Commit;

Commit;

Allows uncommitted data to be read

Read Phenomena

We refer to three different *read phenomena* when **Transaction-1** reads data that **Transaction-2** might have changed:

1. Non-repeatable Read
2. Dirty Read
3. Phantom Read

Non-repeatable Read

- A **non-repeatable read** occurs, when during the course of a transaction, a row is retrieved **twice** and the values **within the row** differ between **reads.**

Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE id = 1;
```

Transaction 2

```
/* Query 2 */
UPDATE users SET age = 21
WHERE id = 1;
COMMIT;
```

Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE id = 1;
COMMIT; */
```

This happens due to 2nd transaction **updating** the same row in between the execution of 1st transaction. The 2nd transaction **commits** after its execution. There is **no cascading rollback**.

Dirty Read

- A ***dirty read*** occurs when a transaction is allowed to read data that has been **modified by another running transaction** which is **not yet committed**.

- Dirty reads work similarly to non-repeatable reads; however, the **second transaction would not need to be committed**.

Transaction 1	Transaction 2
`/* Query 1 */ SELECT age FROM users WHERE id = 1; /* will read 20 */`	
	`/* Query 2 */ UPDATE users SET age = 21 WHERE id = 1; /* No commit here */`
`/* Query 1 */ SELECT age FROM users WHERE id = 1; /* will read 21 */`	
	`ROLLBACK;`

This happens due to 2nd transaction **updating** the same row in between the execution of 1st transaction. The 2nd transaction **doesn't commit**. This **may lead to cascading rollback** if the 2nd transaction is aborted.

Phantom Read

- A ***phantom read*** occurs when, in the course of a transaction, new rows are **inserted** or **deleted** by another transaction to the records being read.

OR

- A ***phantom read*** occurs when, in the course of a transaction, two identical queries are executed, and the **collection of rows** returned by the second query is **different** from the first.

Transaction 1 | **Transaction 2**

```sql
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10
AND 30;
```

```sql
/* Query 2 */
INSERT INTO users(id, name, age)
VALUES (3, 'Bob', 27);
COMMIT;
```

```sql
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10
AND 30;
COMMIT;
```

This happens due to 2nd transaction **inserting/deleting** rows in between the execution of 1st transaction. The 2nd transaction **commits** after its execution. There is **no cascading rollback**.

Quiz-1

R=

A
1
2

Consider a table R(A) containing {(1),(2)}.

- **Transaction T1 is "update R set A = 2*A" and**
- **Transaction T2 is "select avg(A) from R"**
- **If transaction T2 executes using "read uncommitted", what are the possible values it returns?**

A. 1.5, 3, 2, 2.5

B. 1.5, 2, 3

C. 1.5, 2.5, 3

D. 1.5, 3

Read Uncommitted: **Allows uncommitted data to be read**

Quiz-1

Consider a table R(A) containing {(1),(2)}.

R=

A
1
2

- **Transaction T1 is "update R set A = 2*A" and**
- **Transaction T2 is "select avg(A) from R"**
- **If transaction T2 executes using "read uncommitted", what are the possible values it returns?**

A. 1.5, 3, 2, 2.5

B. 1.5, 2, 3

C. 1.5, 2.5, 3

D. 1.5, 3

Transaction execution order: **T2 T1**, **T1 T2**, **T1.1 T2**, **T1.2 T2**

Read Uncommitted: **Allows uncommitted data to be read**

Quiz-2

R=

A
1
2

S=

B
1
2

Consider a table R(A) and S(B), both containing {(1),(2)}.

- **Transaction T1 is "update R set A = 2*A, update S set B = 2*B" and**
- **Transaction T2 is "select avg(A) from R; select avg(B) from S"**
- **If transaction T2 executes using "read committed", is it possible for T2 to return two different values?**

Read Committed: **Allows only committed data to be read, but does not require repeatable reads**

Quiz-2

R=

A
1
2

S=

B
1
2

Consider a table R(A) and S(B), both containing {(1),(2)}.

- Transaction T1 is "update R set A = 2*A, update S set B = 2*B" and
- Transaction T2 is "select avg(A) from R; select avg(B) from S"
- If transaction T2 executes using "read committed", is it possible for T2 to return two different values?

YES

It can return T2,T1 (1.5 , 1.5) OR T1,T2(3 , 3) OR T2.1 (1.5) T1 T2.2 (3)

Read Committed: Allows only committed data to be read, but does not require repeatable reads

Quiz-3

R=

A
1
2

S=

B
1
2

Consider a table R(A) and S(B), both containing {(1),(2)}.

- **Transaction T1 is "update R set A = 2*A, update S set B = 2*B" and**
- **Transaction T2 is "select avg(A) from R; select avg(B) from S"**
- **If transaction T2 executes using "read committed", is it possible for T2 to return a smaller avg(B) than avg(A)?**

Read Committed: Allows only committed data to be read, but does not require repeatable reads

Quiz-3

R=

A
1
2

S=

B
1
2

Consider a table R(A) and S(B), both containing {(1),(2)}.

- **Transaction T1 is "update R set A = 2*A, update S set B = 2*B" and**
- **Transaction T2 is "select avg(A) from R; select avg(B) from S"**
- **If transaction T2 executes using "read committed", is it possible for T2 to return a smaller avg(B) than avg(A)?**

NO

It can return T2,T1 (1.5 , 1.5) OR T1,T2(3 , 3) OR T2.1 (1.5) T1 T2.2 (3)

Read Committed: Allows only committed data to be read, but does not require repeatable reads

Quiz-4

R=

A
1
2

Consider table R(A) containing {(1),(2)}.

- **Transaction T1 is "update R set A = 2*A, insert into R values(6)" and**
- **Transaction T2 is "select avg(A) from R; select avg(A) from R;"**
- **If transaction T2 executes using "repeatable read", what are the possible values returned by its SECOND statement?**

A. 4, 1.5

B. 1.5, 2, 4

C. 1.5, 4, 3

D. 1.5, 2, 3, 4

Repeatable Read: **Allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it.**

Quiz-4

R=

A
1
2

Consider table R(A) containing {(1),(2)}.

- **Transaction T1 is "update R set A = 2*A, insert into R values(6)" and**
- **Transaction T2 is "select avg(A) from R; select avg(A) from R;"**
- **If transaction T2 executes using "repeatable read", what are the possible values returned by its SECOND statement?**

A. **4, 1.5**

B. 1.5, 2, 4

C. 1.5, 4, 3

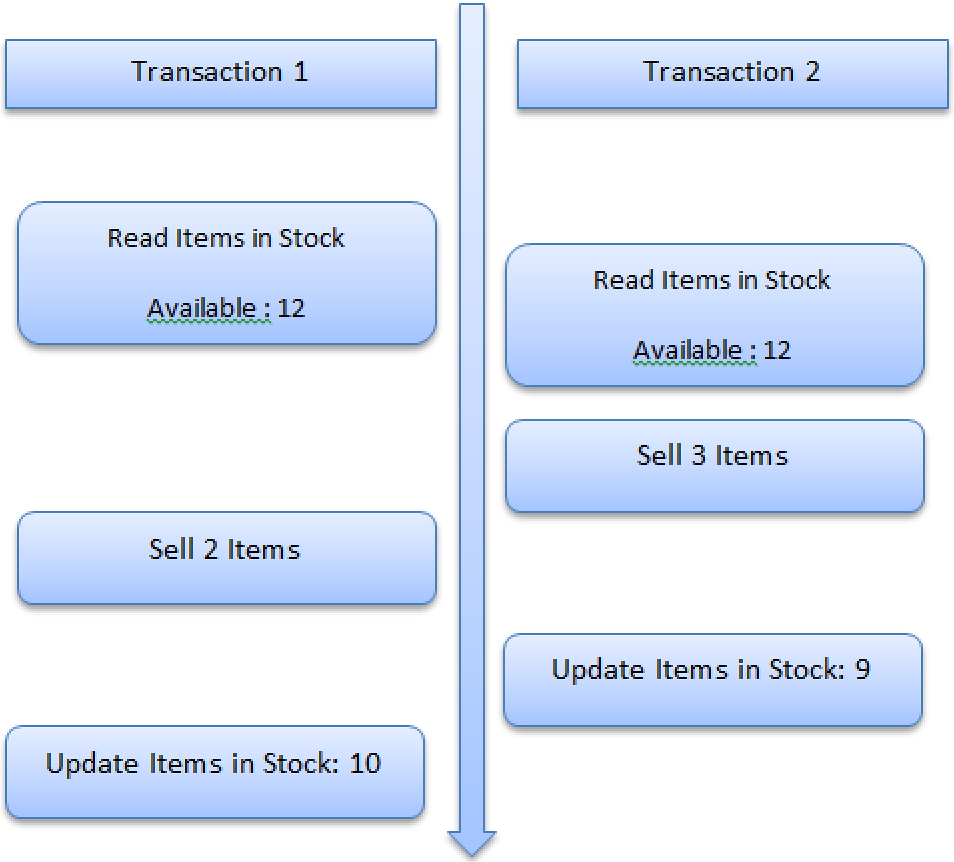
D. 1.5, 2, 3, 4

T1 T2 , T2 T1

Repeatable Read: Allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it.

Lost updates

A lost update occurs when two processes read the same data and then try to update the data with a different value.



The correct result is:

T2=12 - 3 =9
T1= 9 - 2 = 7

Isolation levels vs Read phenomena

Isolation level \ Read phenomena	Dirty reads	Lost updates	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur	may occur
Read Committed	don't occur	may occur	may occur	may occur
Repeatable Read	don't occur	don't occur	don't occur	may occur
Serializable	don't occur	don't occur	don't occur	don't occur

Range locks are a mechanism that's in between **locking a row** and **locking a table**:
if you're running a SELECT query with a WHERE clause, range locks will lock some of the rows that exist *close* to your selected rows (before and after).

Range locks not maintained

Implementation of Isolation Levels

- **Locking**
  - Lock on whole database **vs** lock on items
  - How long to hold lock?
  - Shared **vs** exclusive locks
- **Timestamps**
  - Transaction timestamp is assigned when a transaction begins
  - Each data items store two timestamps:
    1) The **read timestamp** of a data item holds the largest (that is, the most recent) timestamp of those transactions that read the data item.
    2) The **write timestamp** of a data item holds the timestamp of the transaction that wrote the current value of the data item.
  - Timestamps are used to ensure that transactions access each data item in order of the transactions' timestamps if their accesses **conflict**.
- **Multiple versions of each data item**
  - "**snapshot isolation**" is widely used in practice

Lock-Based Protocols

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item.

- Data items can be locked in two modes :

1. **exclusive *(X) mode***. Data item can be both **read** as well as **written**. X-lock is requested using **lock-X** instruction.
2. **shared *(S) mode***. Data item can only be **read**. S-lock is requested using **lock-S** instruction.

- Lock requests are made to **concurrency-control manager**. **Transaction can proceed only after request is granted**.

Lock-Based Protocols (Cont.)

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be **granted** a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item, but if any transaction holds an **exclusive lock** on the item no other transaction may hold any lock on the item.
- If a **lock cannot be granted**, the requesting transaction is made to **wait** till all incompatible locks held by other transactions have been released. The lock is then granted.

Lock-Based Protocols (Cont.)

- Transaction *T*1 transfers $50 from account *B* to account *A*.
- Transaction *T*2 displays the total amount of money in accounts *A* and *B*—that is, the sum *A* + *B*

A=$100 and B=$200

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

display(A+B) → 300

If T_1 and T_2 are executed serially, in any order, then transaction T_2 will display the value $300.

Lock-Based Protocols (Cont.)

A=$100 and B=$200

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

T1 & T2 are run Concurrently

display(A+B) → 250

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

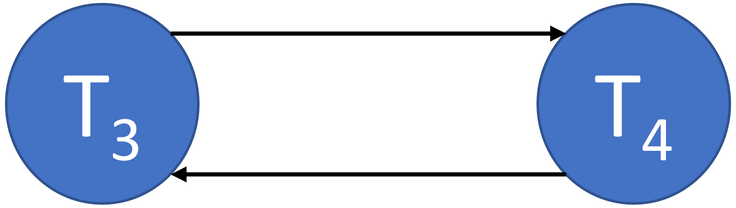
Schedule-1

- If, however, T_1 and T_2 transactions are executed concurrently as in **schedule-1**, then transaction T_2 displays ***$250***, which is incorrect.
- The reason for this mistake is that the transaction T_1 unlocked data item ***B*** **too early**, as a result of which T_2 saw an inconsistent state.

Pitfalls of Lock-Based Protocols

- Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	



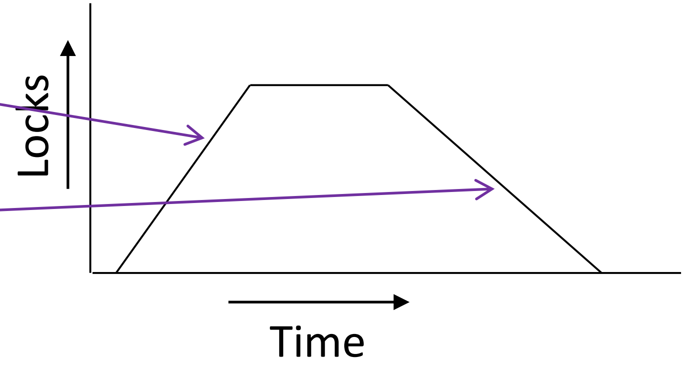
- Neither T_3 nor T_4 can make progress — executing ***lock-S(B)*** causes T_4 to wait for T_3 to release its lock on B, while executing ***lock-X(A)*** causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 **must be rolled back** and its locks released.

Pitfalls of Lock-Based Protocols

- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an **X-lock** on an item, while a sequence of other transactions request and are granted an **S-lock** on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol

- This protocol which ensures conflict-serializable schedules.
- Phase 1: **Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: **Shrinking Phase**
  - Transaction may release locks
  - Transaction may not obtain locks



- Initially, a transaction is in the **growing phase**. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the **shrinking phase**, and it can issue no more lock requests.

The Two-Phase Locking Protocol

Transactions ***T3*** and ***T4*** are **two phase**. On the other hand, transactions ***T1*** and ***T2*** are **not two phase**.

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

Not Two-Phase locking

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

Two-Phase locking

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking also *does not* ensure freedom from **deadlocks.**
- **Cascading roll-back** is possible under two-phase locking.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Deadlock

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Cascading roll-back

The Two-Phase Locking Protocol (Cont.)

- Extensions to basic two-phase locking needed to ensure recoverability of freedom from cascading roll-back:
  - **Strict two-phase locking:** a transaction must hold all its ***exclusive locks*** till it commits/aborts.
    - Ensures recoverability and avoids cascading roll-backs
  - **Rigorous two-phase locking**: a transaction must hold ***all locks*** till commit/abort.
    - Transactions can be serialized in the order in which they commit.

Most databases implement rigorous two-phase locking, *but refer to it as simply two-phase locking*

Rigorous Two-Phase Locking Protocol

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

Two-Phase Locking

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).
commit

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).
commit

Rigorous Two-Phase Locking

Two-phase locking protocol with lock conversions

T_8: read(a_1);
read(a_2);
. . .
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$)

1. Growing Phase:
- can acquire a lock-S on item
- can acquire a lock-X on item
- can **convert** a lock-S to a lock-X (**upgrade**)

2. Shrinking Phase:
- can release a lock-S
- can release a lock-X
- can convert a lock-X to a lock-S (**downgrade**)

Snapshot Isolation

Snapshot Isolation

- snapshot isolation involves giving a transaction a **“snapshot” of the database** at the time when it begins its execution.

- The transaction then operates on that snapshot in **complete isolation** from concurrent transactions.

- The data values in the snapshot consist only of **values written by committed transactions**.

- Updates are kept in the transaction’s **private workspace** until the transaction successfully commits, at which point the updates are written to the database.

Snapshot Isolation

- A transaction T2 executing with Snapshot Isolation
  - Takes snapshot of committed data at start
  - Always reads/modifies data in its own snapshot
  - Updates of concurrent transactions are not visible to T2
  - Writes of T2 completes only if it commits
  - **First-committer-wins rule**:
    - T2 aborts if other concurrent transactions has already written an update to data that T2 intends to write.
    - T2 Commits only if no other concurrent transaction has already written an update to data that T2 intends to write.

X=Y=Z=0

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) → 0 W(Y:=5) 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
    1. If the item has been updated by any concurrent transaction, then T aborts.
    2. Otherwise T may proceed with its execution including possibly committing.