Database Systems
Lecture16 – Chapter 17: Transactions

Beomseok Nam (남범석)
bnam@skku.edu

Outline

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

Transaction Concept

- **Transaction**
  - a **sequence of operations** performed as a **single logical unit** of work that satisfies **ACID** principles.

- E.g., transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

If the transaction fails after step 3 but before step 6, $50 will be "lost" leading to an inconsistent database state

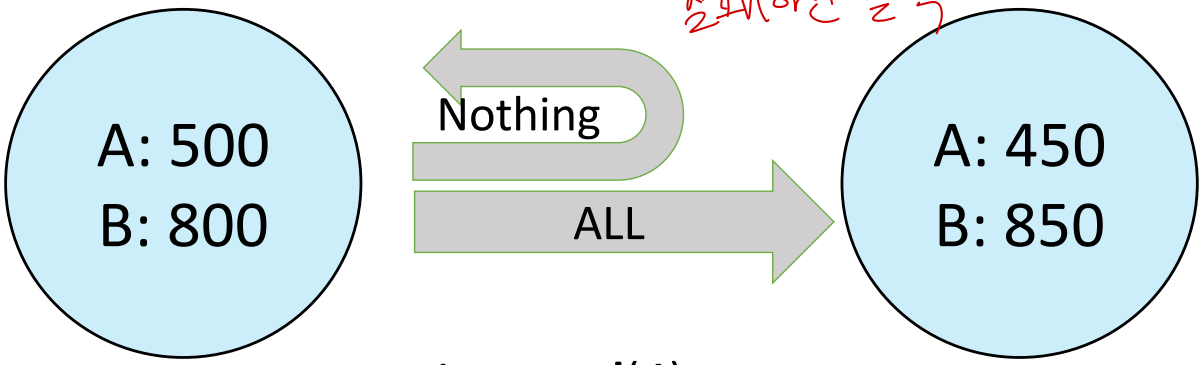
- Two main issues to deal with:
  - **Failures**: hardware failures and system crashes
  - **Concurrent execution** of multiple transactions

ACID Principles

- **Atomicity.** All or Nothing.
  - Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Serializability
  - Concurrent execution of multiple transactions preserves the consistency of the database.
- **Isolation.**
  - Each transaction must be unaware of other concurrently executing transactions.
  - Intermediate transaction results must be hidden from other concurrently executed transactions.
- **Durability.**
  - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Atomicity

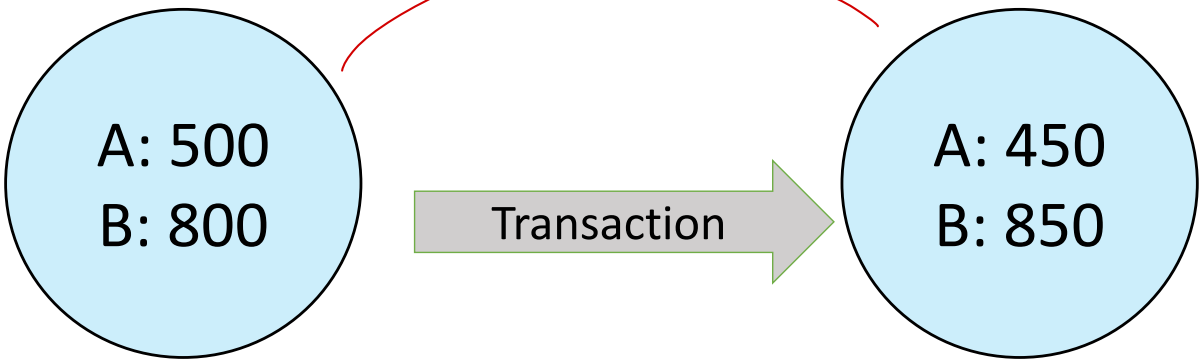
- **Atomicity = ALL or NOTHING**
  - All operations within the transaction are completed successfully, or none are. If any of the operations fails, the entire transaction is rolled back.



1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

Consistency

- **Consistency requirement** :
  - The database moves from one valid state to another.



- consistency requirements include
  - Explicit integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints such as
    - the sum of A and B is unchanged by the execution of the transaction

Isolation

- **Isolation requirement:** Prevent any concurrent transaction from accessing the partially updated DB

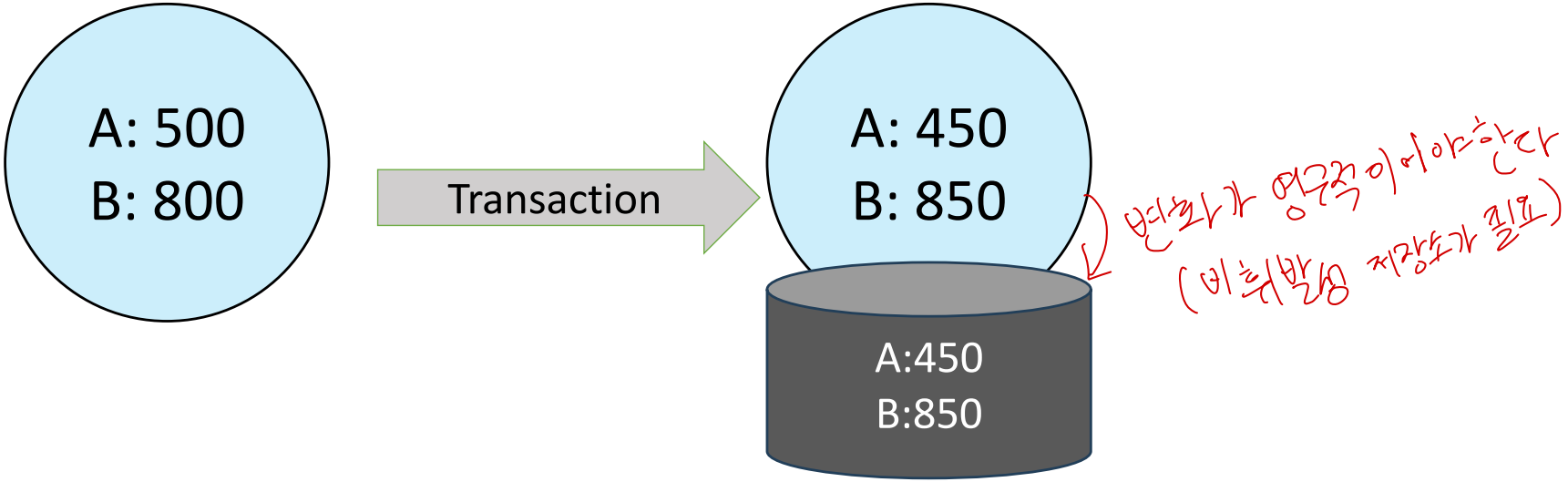
T1	T2
read(A)	
A := A-50	
write (A)	
	read(A), read(B), print(A+B)
read(B)	
B := B+50	
write(B)	

T2 will see an inconsistent DB
(the sum *A + B* will be less than it should be).

- Isolation can be ensured trivially by running transactions **serially**, i.e., one after the other.
- However, executing multiple transactions sequentially has performance problems.

Durability

- **Durability requirement**
  - Once a transaction is committed, its changes are permanent, even in the case of a system failure.



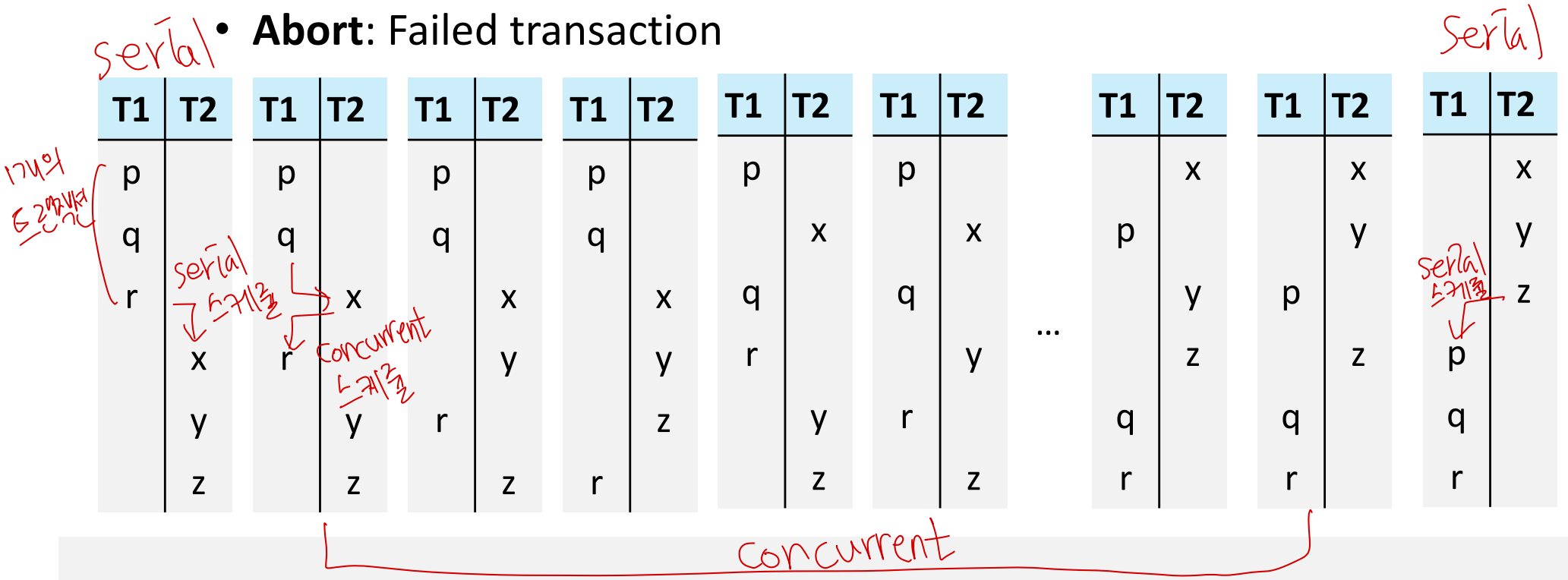
Isolation: Concurrent Executions

- Multiple transactions are allowed to run concurrently.
- Advantages
  - **Increased resource utilization** → higher ***throughput***
    - E.g., one transaction uses the CPU while another performs I/Os.
  - **Reduced waiting time** → low **response time** (latency)

- **Concurrency control schemes**
  - control the interaction among the concurrent transactions

Schedules

- **Schedule** 스케줄
  - Interleave instructions of concurrent transactions
  - Must preserve the order of instructions that appear in each individual transaction.
- Last instruction of transaction
  - **Commit**: Successful transaction
  - **Abort**: Failed transaction



Serial Schedule 1

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*.
- A serial schedule in which T_1 is followed by T_2 :

A: 400
B: 800
↓
A: 350
B: 850

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

↓
A: 315
B: 885
⇒ 1200

DBMS가 트랜잭션의 순서를 질의 고려하지 않음
⇒ 순서가 중요하다면 app이 유의해야 한다

Serial Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

A: 400
B: 800
↓
A: 360
B: 8400

A: ↓ 310
B: 890
⇒ 1200

Serializable Schedule 3

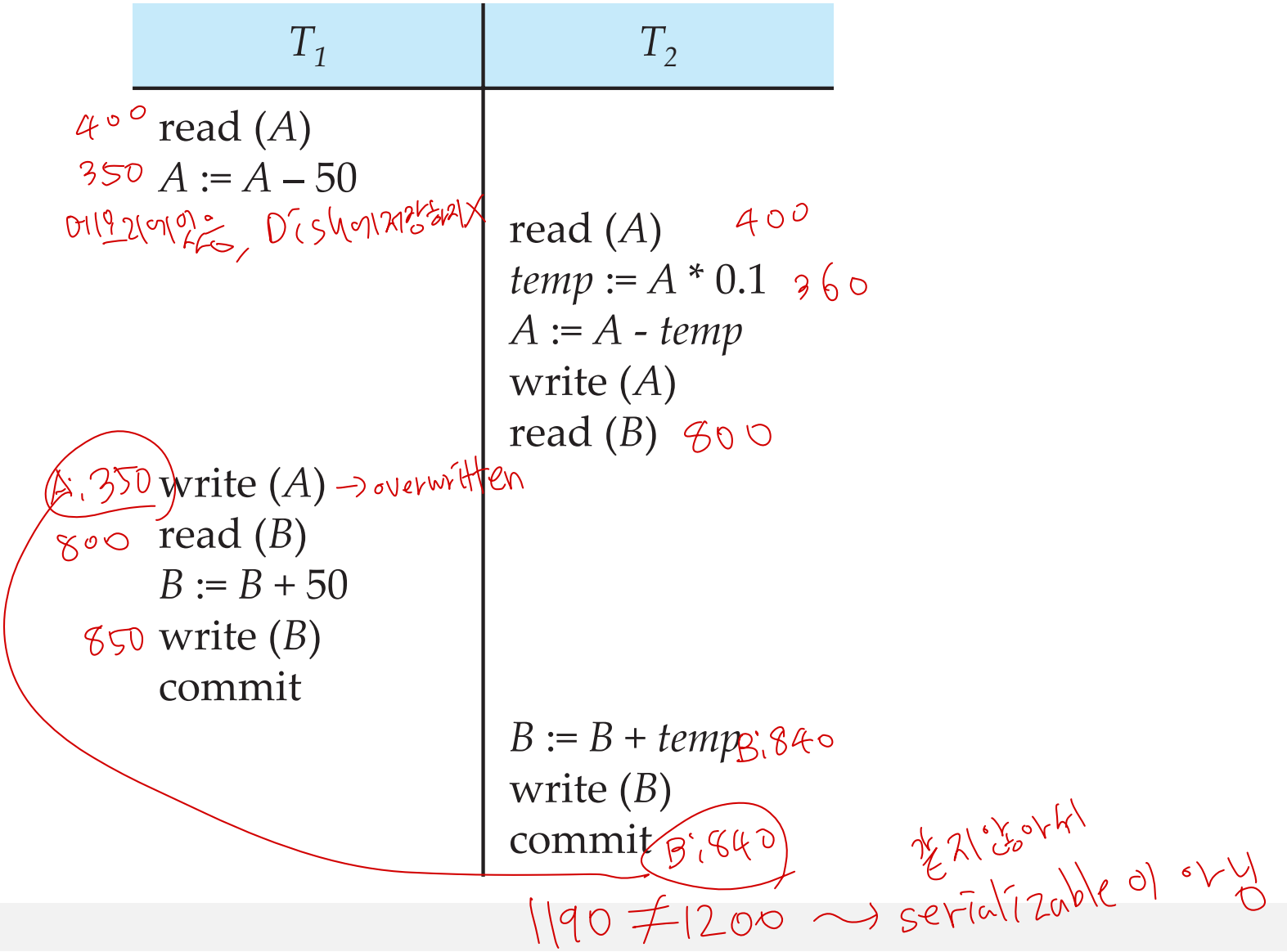
- This is not a serial schedule, but it is *equivalent* to Schedule 1

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

the sum A + B is preserved.

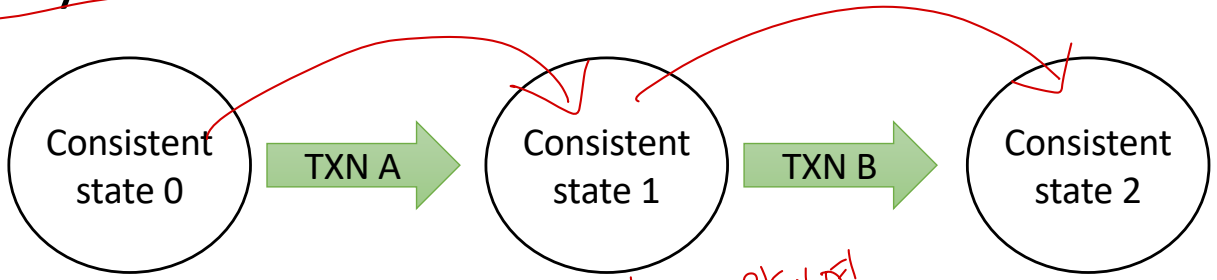
Schedule 4 (Not Serializable)

- The following schedule does not preserve the value of $(A + B)$.

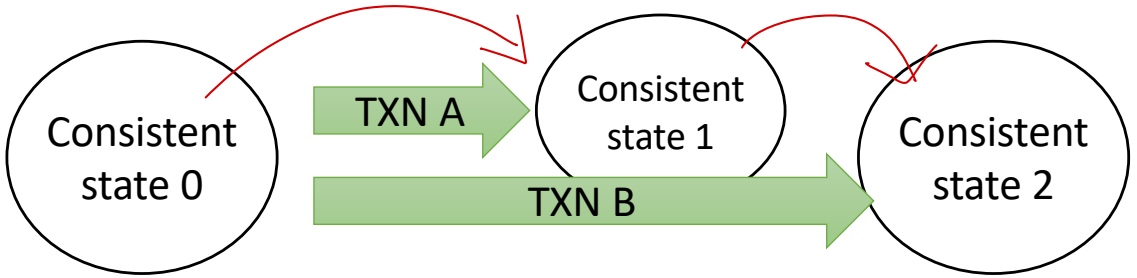


Serializability

- Serial execution of a set of transactions preserves database consistency.



- A schedule is ***serializable*** if it is equivalent to a serial schedule.



- Different forms of serializability:
  1. **Conflict serializability**
  2. **View serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions **perform computations using local buffers** in between reads and writes.
- Modified **data is exposed** to other transactions **only when write is called.**

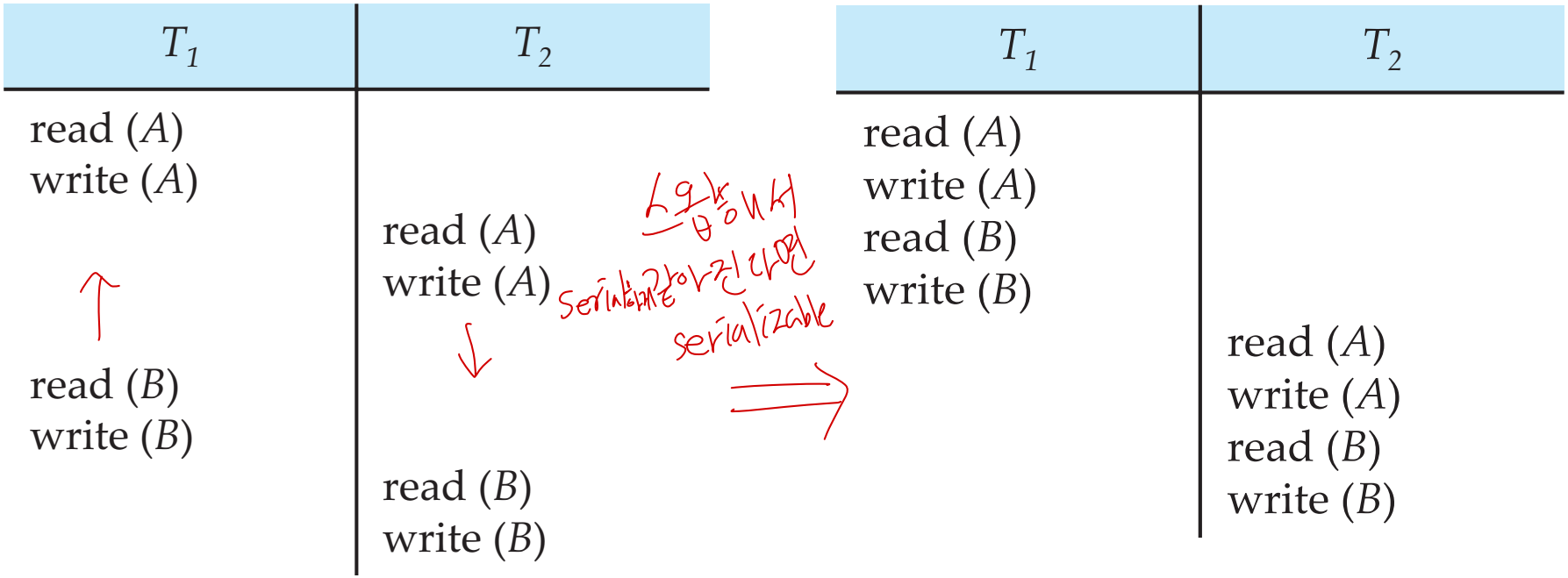
Conflicting Instructions

- Instructions I_a and I_b of transactions T_a and T_b **conflict** if and only if at least one of the instructions write a shared data item.
  1. I_a = **read**(Q), I_b = **read**(Q). I_a and I_b don't conflict.
  2. I_a = **read**(Q), I_b = **write**(Q). They conflict.
  3. I_a = **write**(Q), I_b = **read**(Q). They conflict
  4. I_a = **write**(Q), I_b = **write**(Q). They conflict

- If two contiguous *instructions* do not conflict, we can swap them
- If a schedule S can be transformed into a schedule S' by swapping non-conflicting instructions, we say that S and S' are **conflict equivalent**.

Conflict Serializability

- Schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule
  - Schedule 1 can be transformed into a serial schedule 2 by series of swaps of non-conflicting instructions.

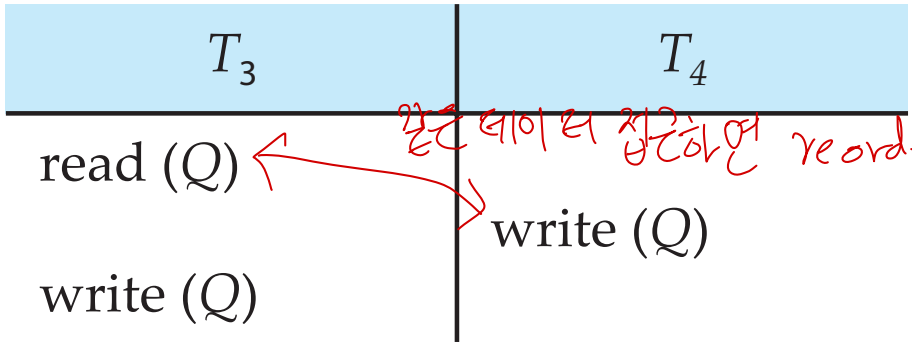


Conflict Serializable Schedule 1

Serial Schedule 2

Conflict Serializability (Cont.)

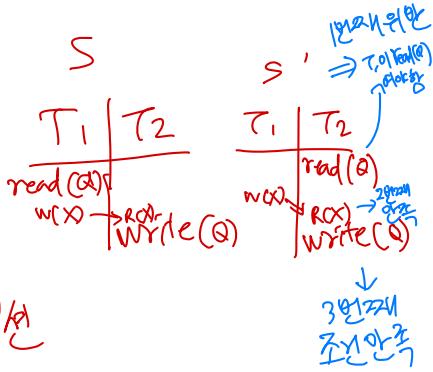
- Example of a schedule that is not conflict serializable:



- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

View Serializability

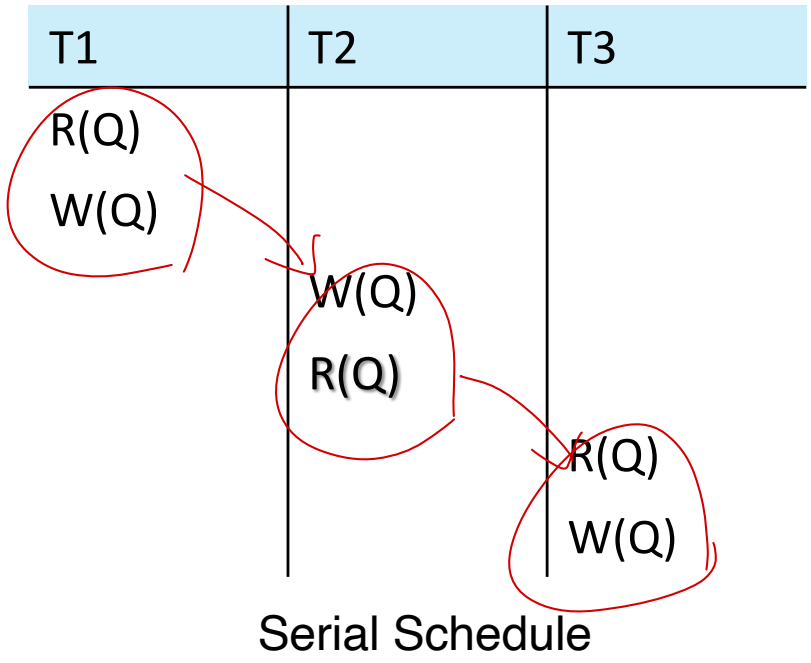
- Two schedules *S* and *S'* with the same set of transactions are **view equivalent** if the following three conditions hold



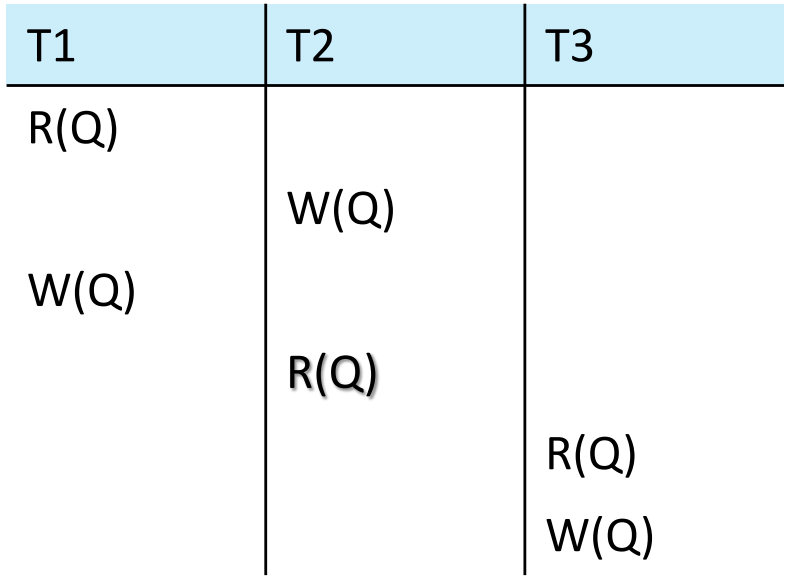
1. **Same initial Reads**
   - If T_a reads the initial value of a data item Q in S, it must do the same in S'.
2. **Same Read-From**
   - If T_a reads Q written by transaction T_b in S, T_i must read the same value of Q written by T_b in S'. (W(Q)→R(Q))
3. **Same Final Write**
   - Final write(*Q*) must be done by the same transaction in both S and *S'*.

- *S* is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.

Q1: Is this a view serializable schedule?



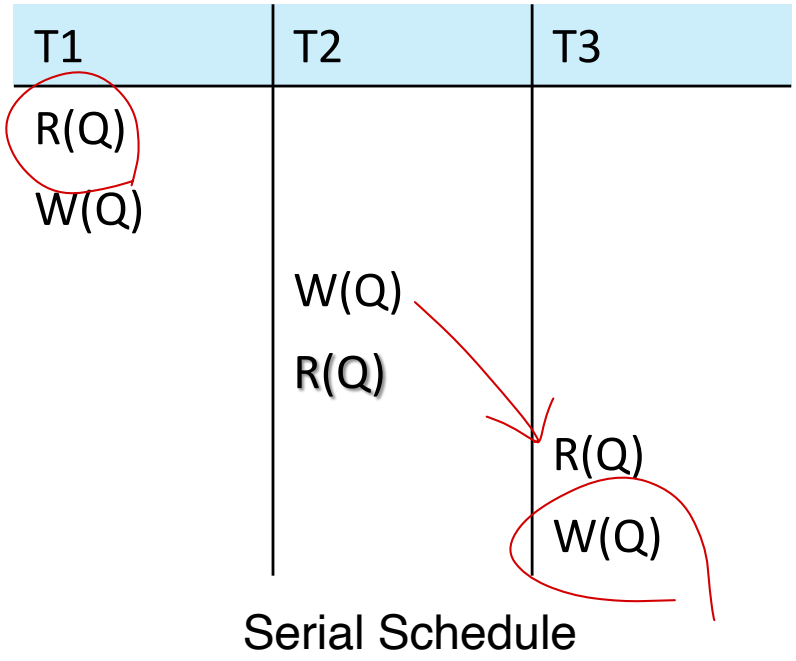
Serial Schedule



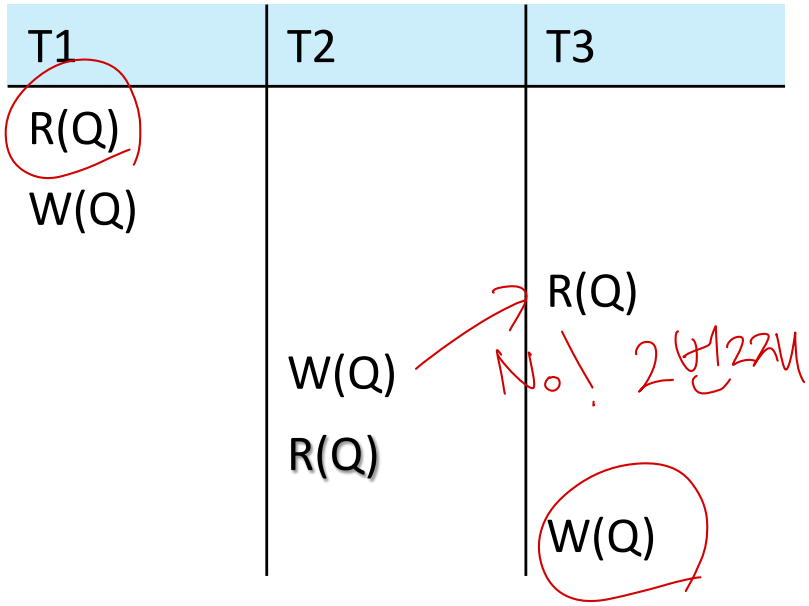
View-Serializable?

- In both schedules, T1 reads the initial value of Q
- In both schedules, T3 performs the final write

Q2: Is this a view serializable schedule?



Serial Schedule



View-Serializable?

- In both schedules, T1 reads the initial value of Q
- In both schedules, T3 performs the final write

Q3: Is this a view serializable schedule?

T1	T2	T3
R(Q)		
W(Q)		
	W(Q)	
		W(Q)

Serial Schedule

T1	T2	T3
R(Q)		
	W(Q)	
W(Q)		
		W(Q)

View-Serializable? Yes

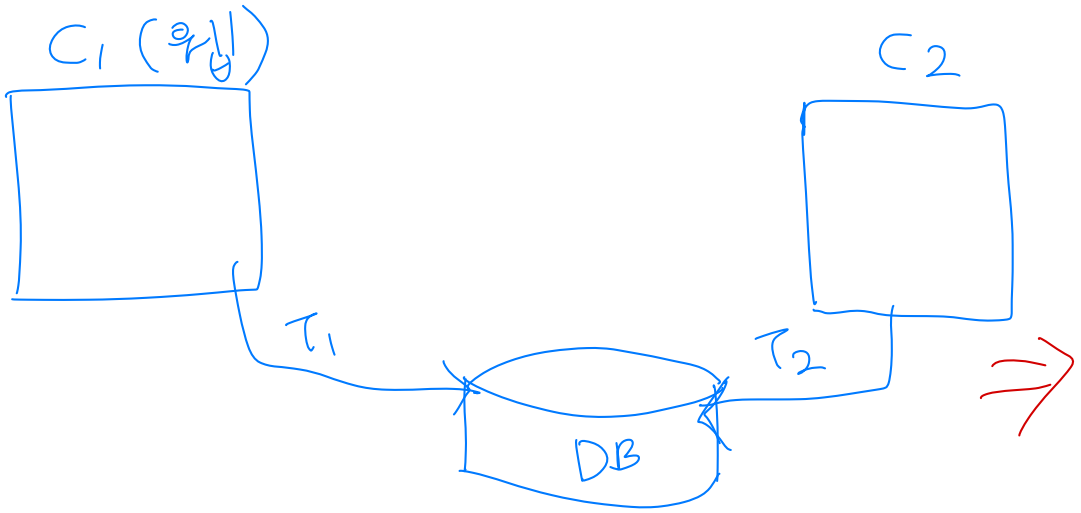
But, conflict serializable? No

- Above is a schedule which is view-serializable but *not* conflict serializable.
- Every view serializable schedule that is not conflict serializable has **blind writes**.

Blind Write

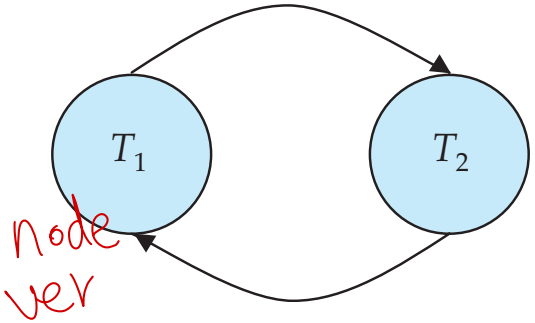
- **Blind Write** occurs when a transaction **writes a value to a data item without reading it.**
- E.g.

```
UPDATE customers
SET reward_points = 0
WHERE customer_id = 123;
```



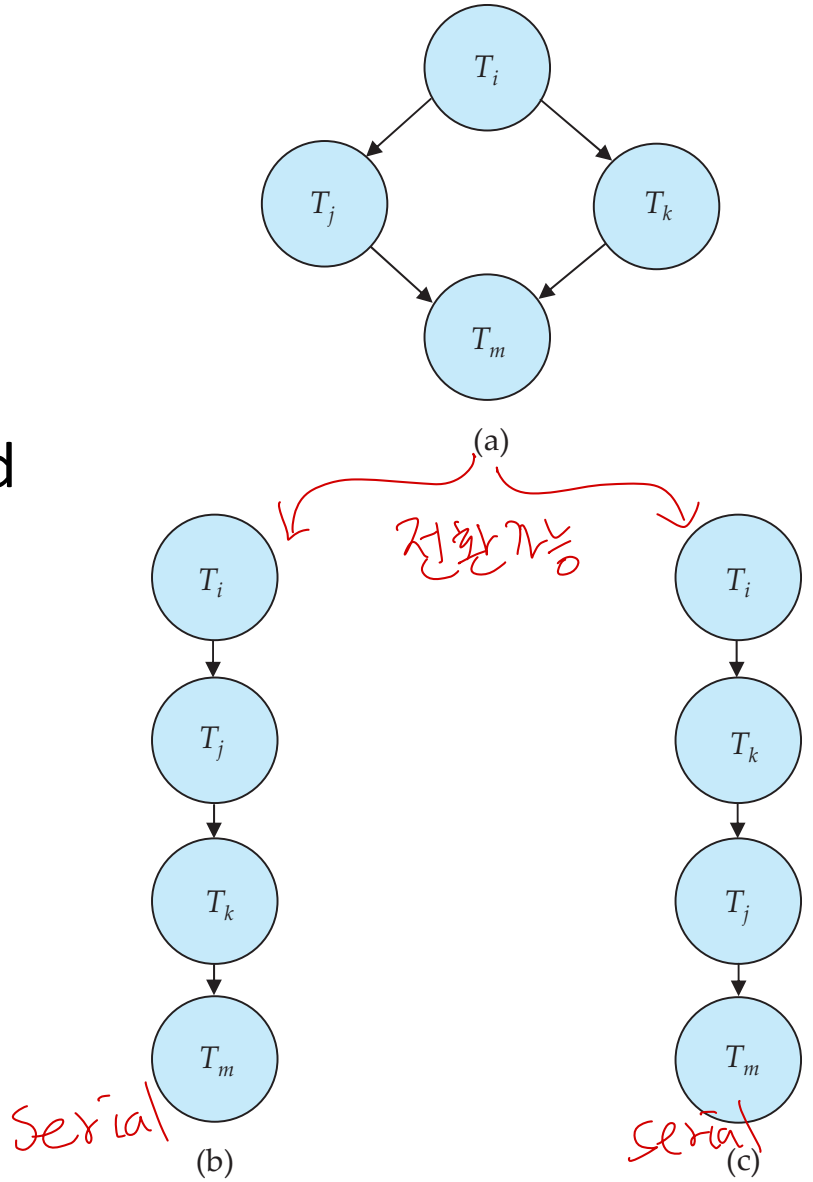
Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_a to T_b if the two transactions conflict and T_a accessed the conflicting data item earlier than T_b.
- We may label the arc by the item that was accessed.
- Example of a precedence graph



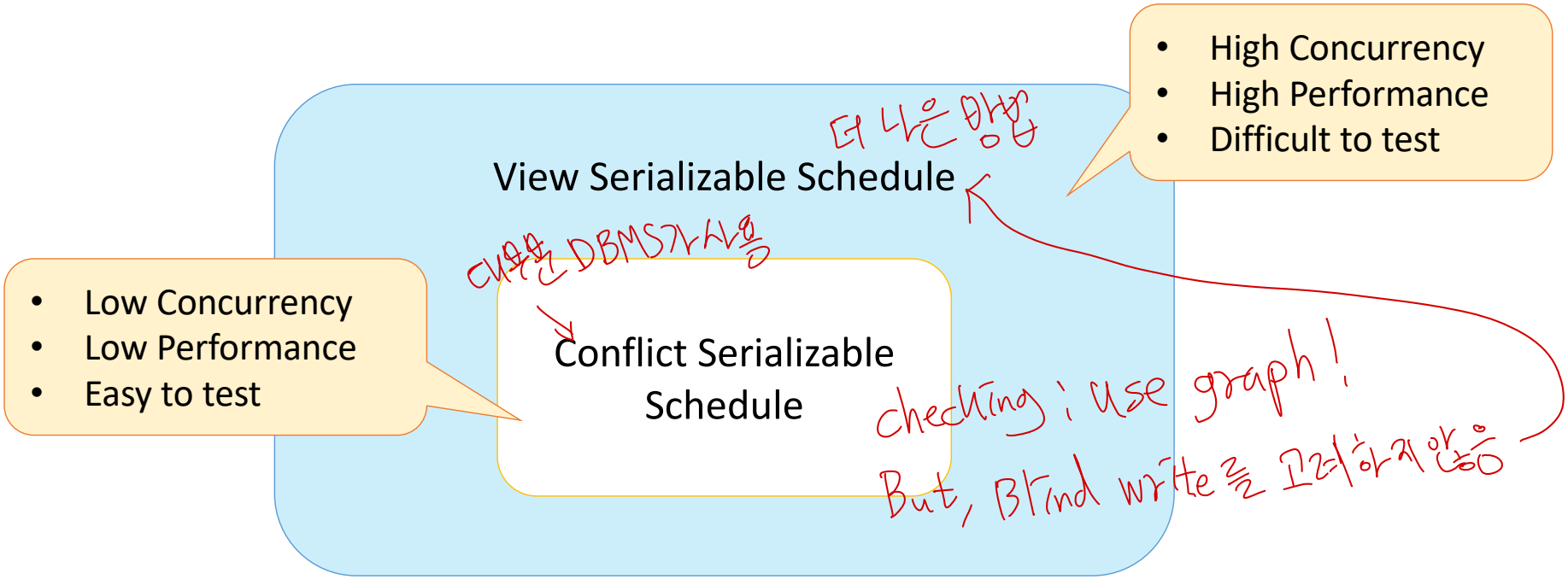
Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist
- If precedence graph is acyclic, the ***serializability order*** can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.



Conflict Serializability vs. View Serializability

- Conflict serializability is more popular in DBMS in practice
- But, conflict serializability actually permit only a subset of serializable schedules.
- The view serializability is expensive (*NP*-complete) to test.
  - Practical algorithms that just check some **sufficient conditions**



Recoverable Schedules

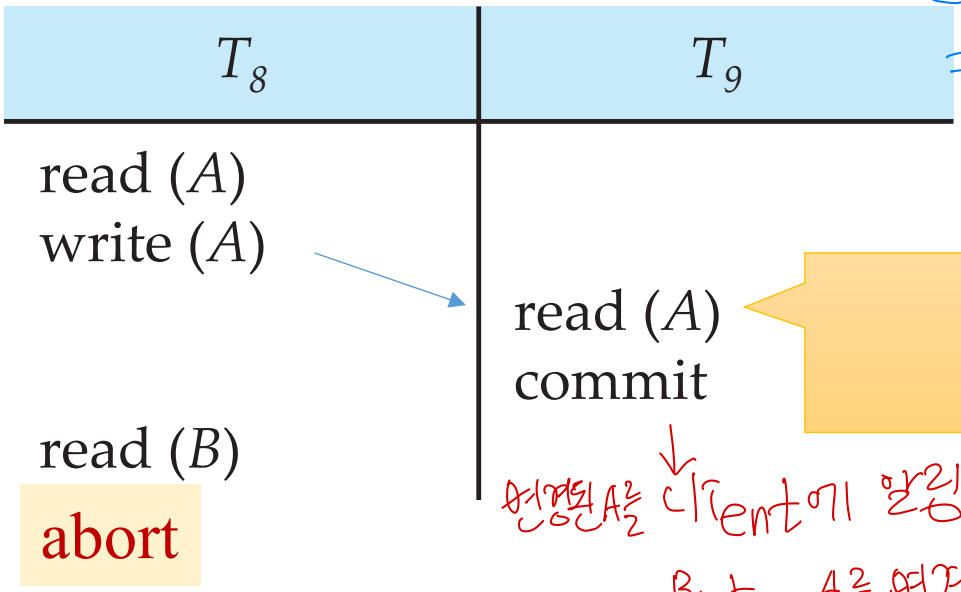
- Schedules must be recoverable

모든 concurrent schedule은 recoverable하다.

- **Recoverable schedule** — if a transaction T_b reads a data written by a transaction T_a, then T_a must commit before T_b.

T_a가 쓴값을 T_b가 읽었다면 T_a는 T_b전에 commit해야함 = recoverable

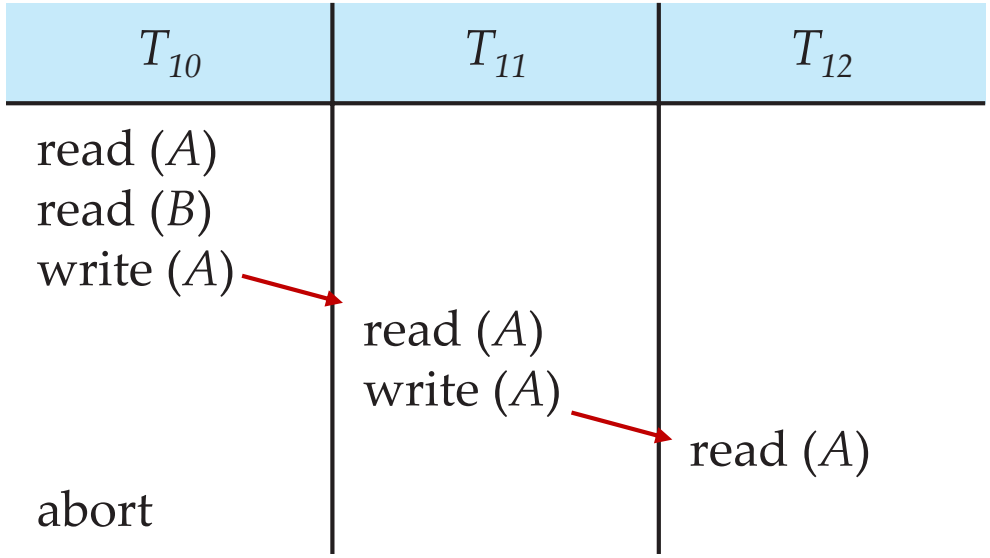
- The following schedule is not recoverable



- If T_8 aborts, T_9 short abort too.

Cascading Rollbacks

- **Cascading rollback –** a transaction failure may lead to a series of transaction rollbacks.
- Consider the following schedule where none of the transactions has yet committed



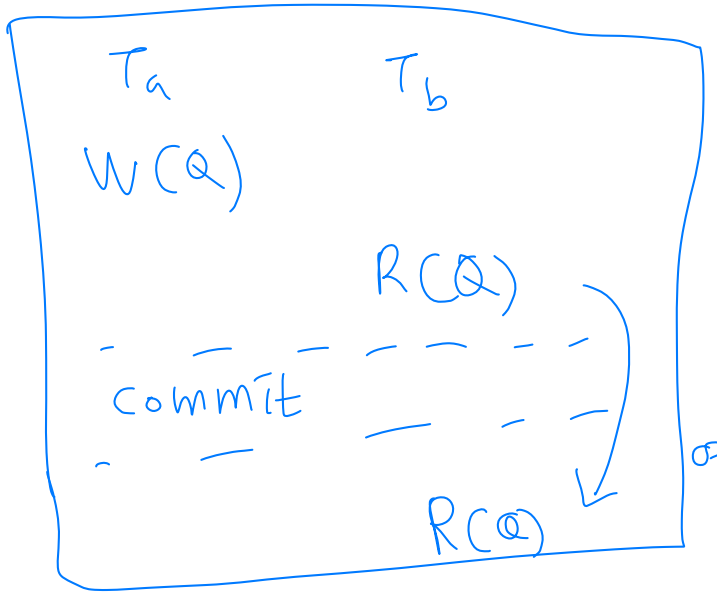
This schedule is recoverable.
If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work (overhead ↑↑)

Cascadeless Schedules

- **Cascadeless schedules** — prevents cascading rollbacks;
  - If T_b reads a data written by T_a, the commit operation of T_a appears before the read operation of T_b.
  - Read committed data only!
- Every Cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

concurrent 트랜잭션의 실행을 제한을 부여하기에 (자유를 제한하기에) concurrency를 해친다 ⇒ 성능↓



여기로 이동시키면 (Ta가 commit한 후에 Tb가 R(Q)를 한다는 규칙을 정하면 롤백의 연쇄 작용을 막을 수 있다)
= Cascadeless 스케줄링

Concurrency Control

- Goal: A DBMS must make scheduling decisions that are
  - either *conflict* or *view* **serializable**, and
  - **recoverable** and **preferably cascadeless**
- DBMS scheduler should provide **a high degree of concurrency**
  - We study concurrency control protocols in Chapter 18.

Weak Levels of Consistency

- Some applications live with weak levels of consistency
  - Tradeoff accuracy for performance

- **On Line Transaction Processing –OLTP**
  - Transactions need to be serializable
    - Short simple transactions
    - Transactions access only a small fraction of the database
- **On Line Analytic Processing –OLAP**
  - Transactions do not need to be serializable
  - Uses database to guide strategic decisions.
    - E.g., A CEO wants to get an approximate total balance of all customers' accounts
    - Transactions access a large fraction of the database

Phenomena caused by Concurrent Transactions

- **dirty read**
  - A transaction reads data written by an uncommitted transaction.
- **nonrepeatable read**
  - A transaction runs the same *point query* twice and finds that a data has been modified by another transaction.
- **phantom read**
  - A transaction runs the same *range query* twice and finds that the result set has new records due to another recently-committed transaction.
- **serialization anomaly**
  - The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

Phantom Phenomenon

- E.g., Transaction 1:
  select *ID, name* **from** *instructor* **where** *salary* > 90000
- E.g., Transaction 2:
  insert into *instructor* **values** ('11111', 'James', 'Marketing', 100000)
- Suppose
  - T1 starts, finds tuples salary > 90000 using index and locks them
  - And then T2 executes.
  - Do T1 and T2 conflict? Does tuple level locking detect the conflict?
  - Instance of the **phantom phenomenon**

Transaction Isolation Level

- **Serializable** — default in SQL-92, **SQLite**
- **Repeatable read** —default in **MySQL**
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — default in **PostgreSQL & Oracle**
  - Successive reads may return different (but committed) data.
- **Read uncommitted** — even uncommitted records may be read.

Weak ↕ Strong

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Repeatable Read vs. Read Committed

T1	T2	T3
TS_start		
	TS_start	
	W(Q←1)	
	TS_commit	
1 ← R(Q):		
		TS_start
		W(Q←2)
1 ← R(Q):		
		TS_commit
1 ← R(Q):		
TS_commit		

Repeatable Read

T1	T2	T3
TS_start		
	TS_start	
	W(Q←1)	
	TS_commit	
1 ← R(Q):		
		TS_start
		W(Q←2)
1 ← R(Q):		
		TS_commit
2 ← R(Q):		
TS_commit		

Read Committed

Phantom Read!!!
(a.k.a. Non-Repeatable Read)
This problem happens in PostgreSQL & Oracle

Phantom Phenomenon in Repeatable Read

T1	T2
TS_start	
select name from instructor where ID='1'; → empty table	
	TS_start
	insert into instructor values ('1', 'James', 'CS', 1000);
	TS_commit
select name from instructor where ID='1'; → empty table // GOOD (Repeatable Read)	
update instructor set salary = 2000 where ID = '1';	
select name from instructor where ID='1'; → 'James' Phantom Read here!!!!	
TS_commit	

This anomaly happens in MySQL

Transaction Definition in SQL

- A transaction in SQL ends by:
  - **Commit work** commits current transaction
  - **Rollback work** causes current transaction to abort.
- **Auto Commit**
  - Each SQL statement commits implicitly
  - Implicit commit can be turned off by a database directive
    - E.g., in JDBC -- `connection.setAutoCommit(false);`
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction
  - E.g. In SQL

    ```
    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
    ```

  - E.g. in JDBC - `connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)`

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 assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a “snapshot” of the database