



# Chapter 14: Transactions

Database System Concepts, 6<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Database System Concepts

- Chapter 1: Introduction
- **Part 1: Relational databases**
  - Chapter 2: Introduction to the Relational Model
  - Chapter 3: Introduction to SQL
  - Chapter 4: Intermediate SQL
  - Chapter 5: Advanced SQL
  - Chapter 6: Formal Relational Query Languages
- **Part 2: Database Design**
  - Chapter 7: Database Design: The E-R Approach
  - Chapter 8: Relational Database Design
  - Chapter 9: Application Design
- **Part 3: Data storage and querying**
  - Chapter 10: Storage and File Structure
  - Chapter 11: Indexing and Hashing
  - Chapter 12: Query Processing
  - Chapter 13: Query Optimization
- **Part 4: Transaction management**
  - [Chapter 14: Transactions](#)
  - Chapter 15: Concurrency control
  - Chapter 16: Recovery System
- **Part 5: System Architecture**
  - Chapter 17: Database System Architectures
  - Chapter 18: Parallel Databases
  - Chapter 19: Distributed Databases
- **Part 6: Data Warehousing, Mining, and IR**
  - Chapter 20: Data Mining
  - Chapter 21: Information Retrieval
- **Part 7: Specialty Databases**
  - Chapter 22: Object-Based Databases
  - Chapter 23: XML
- **Part 8: Advanced Topics**
  - Chapter 24: Advanced Application Development
  - Chapter 25: Advanced Data Types
  - Chapter 26: Advanced Transaction Processing
- **Part 9: Case studies**
  - Chapter 27: PostgreSQL
  - Chapter 28: Oracle
  - Chapter 29: IBM DB2 Universal Database
  - Chapter 30: Microsoft SQL Server
- **Online Appendices**
  - Appendix A: Detailed University Schema
  - Appendix B: Advanced Relational Database Model
  - Appendix C: Other Relational Query Languages
  - Appendix D: Network Model
  - Appendix E: Hierarchical Model



# Chapter 14: Transactions

- Transaction Concept
- A Simple Transaction Model
- Storage Structure
- Transaction Atomicity and Durability
- Transaction Isolation
- Serializability
- Transaction Isolation and Atomicity
- Transaction Isolation Level
- Implementation of Isolation Levels
- Transaction Definition in SQL



# Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items
- 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)**
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions



# ACID Properties

- A **transaction** is a unit of program execution that accesses and possibly updates various data items
- To preserve the integrity of data the database system must ensure:
  - **Atomicity:** Either **all** operations of the transaction are properly reflected in the database or **none** are
  - **Consistency:** Execution of a transaction in isolation preserves the consistency of the database
  - **Isolation:** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions
    - ▶ Intermediate transaction results must be hidden from other concurrently executed transactions
    - ▶ That is, for every pair of transactions  $T_i$  and  $T_j$ , it appears to  $T_i$  that **either**  $T_j$  finished execution before  $T_i$  started, **or**  $T_j$  started execution after  $T_i$  finished
  - **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures



# Chapter 14: Transactions

- Transaction Concept
- A Simple Transaction Model
- Storage Structure
- Transaction Atomicity and Durability
- Transaction Isolation
- Serializability
- Transaction Isolation and Atomicity
- Transaction Isolation Level
- Implementation of Isolation Levels
- Transaction Definition in SQL



# A Simple Transaction Model [1/3]

- We ignore operations other than **read** and **write** instructions
  - We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes
- 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)**
- **Atomicity requirement**
  - if the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
    - ▶ Failure could be due to software or hardware
  - the system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures



# A Simple Transaction Model [2/3]

- 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)**

- **Consistency requirement** in above example:

- the sum of A and B is unchanged by the execution of the transaction

- In general, consistency requirements include

- ▶ Explicitly specified integrity constraints such as primary keys and foreign keys
  - ▶ Implicit integrity constraints
    - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database
  - During transaction execution the database may be **temporarily inconsistent**
  - When the transaction completes successfully the database must be **consistent**
    - ▶ Erroneous transaction logic can lead to inconsistency



# A Simple Transaction Model [3/3]

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum  $A + B$  will be less than it should be)

T1

- **read(A)**
  - $A := A - 50$
  - **write(A)**
- 
- **read(B)**
  - $B := B + 50$
  - **write(B)**

T2

read(A), read(B), print(A+B)

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other
- However, executing multiple transactions concurrently has significant benefits, as we will see later



# Chapter 14: Transactions

- Transaction Concept
- A Simple Transaction Model
- [Storage Structure](#)
- Transaction Atomicity and Durability
- Transaction Isolation
- Serializability
- Transaction Isolation and Atomicity
- Transaction Isolation Level
- Implementation of Isolation Levels
- Transaction Definition in SQL



# Storage Structure

## ■ Volatile storage:

- does not survive system crashes
- examples: main memory, cache memory

## ■ Nonvolatile storage:

- survives system **crashes**
- examples: disk, tape, flash memory, non-volatile (battery backed up) RAM
- but may still fail, losing data

## ■ Stable storage:

- a **mythical form of storage** that survives **all failures**
  - approximated by maintaining **multiple copies on distinct nonvolatile media**
  - See book for more details on how to implement stable storage
- 
- For a transaction to be **durable**, data changes need to be written to stable storage
  - For a transaction to be **atomic**, log records need to be written to stable storage before any changes are made to the database on disk



# Chapter 14: Transactions

- Transaction Concept
- A Simple Transaction Model
- Storage Structure
- **Transaction Atomicity and Durability**
- Transaction Isolation
- Serializability
- Transaction Isolation and Atomicity
- Transaction Isolation Level
- Implementation of Isolation Levels
- Transaction Definition in SQL



# Transaction State

- **Active:** the initial state; the transaction stays in this state while it is executing
- **Partially committed:** after the final statement has been executed
- **Failed:** after the discovery that normal execution can no longer proceed
- **Aborted:** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction
  - Two options after it has been aborted:
    - ▶ restart the transaction (can be done only if no internal logical error)
    - ▶ kill the transaction
- **Committed:** after successful completion

Maintaining a log is critical!

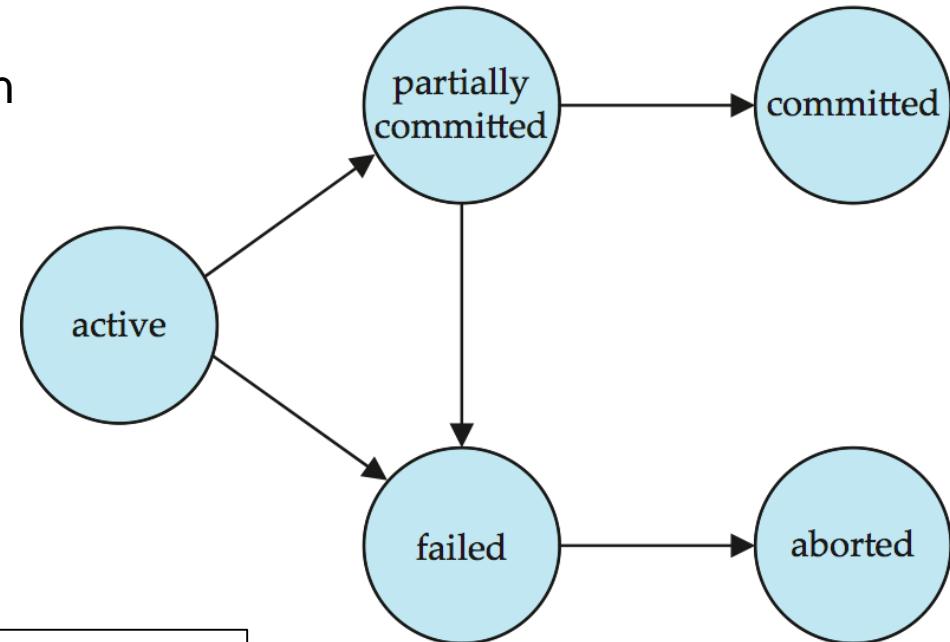


Fig 14.01



# Chapter 14: Transactions

- Transaction Concept
- A Simple Transaction Model
- Storage Structure
- Transaction Atomicity and Durability
- **Transaction Isolation**
- Serializability
- Transaction Isolation and Atomicity
- Transaction Isolation Level
- Implementation of Isolation Levels
- Transaction Definition in SQL



# Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system
- Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - ▶ E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - ▶ Will study various CC schemes in Chapter 16, after studying notion of correctness of concurrent executions



# Schedules

- **Schedule:** a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- **Serial schedule:** instruction sequences from **one by one** transactions
- A transaction that successfully completes its execution will have **a commit instruction** as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have **an abort instruction** as the last statement



# Schedule 1 and 2

- Let  $T_1$  transfer \$50 from  $A$  to  $B$ , and  $T_2$  transfer 10% of the balance from  $A$  to  $B$ .
- Serial schedules:  $\{ T_1 ; T_2 \}$        $\{ T_2 ; T_1 \}$

$T_1$	$T_2$
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

Fig 14.02

$T_1$	$T_2$
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

Fig 14.03



# Schedule 3

- Let  $T_1$  and  $T_2$  be the transactions defined previously
- The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1
- We call it a *serializable schedule*

$T_1$	$T_2$
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

Fig 13.04

In Schedules 1, 2 and 3, the sum  $A + B$  is preserved.



# Schedule 4

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

T <sub>1</sub>	T <sub>2</sub>
read (A) $A := A - 50$	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B)
write (A) read (B) $B := B + 50$ write (B) commit	$B := B + temp$ write (B) commit

Fig 14.05



# Chapter 14: Transactions

- Transaction Concept
- A Simple Transaction Model
- Storage Structure
- Transaction Atomicity and Durability
- Transaction Isolation
- **Serializability**
- Transaction Isolation and Atomicity
- Transaction Isolation Level
- Implementation of Isolation Levels
- Transaction Definition in SQL



# Serializability

- **Basic Assumption** – Each transaction preserves database consistency
- Thus serial execution of a set of transactions preserves database consistency
- A (possibly concurrent) schedule is **serializable** if it is equivalent to a serial schedule.
- Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**



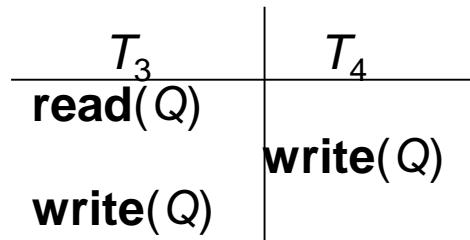
# Conflicting Instructions

- Instructions  $I_i$  and  $I_j$  of transactions  $T_i$  and  $T_j$  respectively, **conflict** if and only if there exists some item  $Q$  accessed by both  $I_i$  and  $I_j$ , and at least one of these instructions wrote  $Q$ 
  1.  $I_i = \text{read}(Q)$ ,  $I_j = \text{read}(Q)$  :  $I_i$  and  $I_j$  don't conflict
  2.  $I_i = \text{read}(Q)$ ,  $I_j = \text{write}(Q)$  : They conflict.
  3.  $I_i = \text{write}(Q)$ ,  $I_j = \text{read}(Q)$  : They conflict
  4.  $I_i = \text{write}(Q)$ ,  $I_j = \text{write}(Q)$  : They conflict
- Intuitively, a conflict between  $I_i$  and  $I_j$  forces a (logical) temporal order between them
  - If  $I_i$  and  $I_j$  are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule



# Conflict Serializability [1/3]

- If a schedule  $S$  can be transformed into a schedule  $S'$  by a series of swaps of non-conflicting instructions, we say that  $S$  and  $S'$  are **conflict equivalent**
- We say that a schedule  $S$  is **conflict serializable** if it is conflict equivalent to a serial schedule
- 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  $< T_4, T_3 >$ .

$T_3$	$T_4$
read(Q)	write(Q)
write(Q)	read(Q)



# Conflict Serializability

[2/3]

- Schedule 3 can be transformed into Schedule 6, a serial schedule where  $T_2$  follows  $T_1$ , by a series of swaps of non-conflicting instructions
  - Therefore Schedule 3 is conflict serializable

**Schedule 3**

$T_1$	$T_2$
read( $A$ )	
write( $A$ )	
	read( $A$ )
	write( $A$ )
<u>read(<math>B</math>)</u>	
<u>write(<math>B</math>)</u>	
	read( $B$ )
	write( $B$ )

**Schedule 5 –**  
After Swapping a Pair of non  
conflicting Instructions in  
schedule 3

$T_1$	$T_2$
read( $A$ )	
write( $A$ )	
<u>read(<math>B</math>)</u>	read( $A$ )
	<u>write(<math>A</math>)</u>
write( $B$ )	
	read( $B$ )
	write( $B$ )

**Schedule 6 –**  
A Serial Schedule That is  
Equivalent to Schedule 3

$T_1$	$T_2$
read( $A$ )	
write( $A$ )	
read( $B$ )	
write( $B$ )	
	read( $A$ )
	write( $A$ )
	read( $B$ )
	write( $B$ )



# Conflict Serializability

[3/3]

- Example of a schedule that is **not conflict serializable**:

Fig 14.09

	$T_3$	$T_4$
read ( $Q$ )		write ( $Q$ )
write ( $Q$ )		

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

$T_3$	$T_4$
read( $Q$ )	
write( $Q$ )	read( $Q$ )
write( $Q$ )	write( $Q$ )



# View Serializability

[1/2]

- Let  $S$  and  $S'$  be two schedules with the same set of transactions.  $S$  and  $S'$  are **view equivalent** if the following 3 conditions are met, for each data item  $Q$ ,
  - If in schedule  $S$ , transaction  $T_i$  reads the initial value of  $Q$ , then in schedule  $S'$  also transaction  $T_i$  must read the initial value of  $Q$
  - If in schedule  $S$  transaction  $T_i$  executes **read( $Q$ )**, and that value was produced by transaction  $T_j$  (if any), then in schedule  $S'$  also transaction  $T_i$  must read the value of  $Q$  that was produced by the same **write( $Q$ )** operation of transaction  $T_j$
  - The transaction (if any) that performs **the final write( $Q$ ) operation** in schedule  $S$  must also perform **the final write( $Q$ ) operation** in schedule  $S'$

As can be seen, view equivalence is also based purely on **reads** and **writes** alone

---

Schedule A

$T_3$	$T_4$
read ( $Q$ )	
	write ( $Q$ )

Schedule B

$T_3$	$T_4$
read( $Q$ )	
write( $Q$ )	
	write( $Q$ )

Schedule C

$T_3$	$T_4$
	write( $Q$ )
read( $Q$ )	

- 스케줄 A는 Conflict serializable schedule이 아님, View serializable schedule도 아님



# View Serializability [2/2]

- A schedule  $S$  is **view serializable** if it is **view equivalent** to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is **view-serializable** but **not** conflict serializable.

$T_{27}$	$T_{28}$	$T_{29}$
read ( $Q$ )	write ( $Q$ )	
write ( $Q$ )		write ( $Q$ )

- Non-conflicting operations를 swap할 여지가 없다
- $T_{27} T_{28} T_{29}$  의 어떤 combination도 만들수가 없다
- Conflict serializable schedule이 될수 없다
- 그러나 왼쪽의 schedule은  $\langle T_{27}, T_{28}, T_{29} \rangle$  와 View-serialibility의 3가지 조건을 만족하므로 View serializable schedule이다

- Every view serializable schedule that is not conflict serializable has **blind writes**
- Intuition: 두개 schedule에서 read들이 같은값을 읽었고, final write가 같다면 Database에 반영되는 결과가 serial execution중의 한가지와 같게되는 view serializable schedule이다 → we can have more potential concurrency

View Serializable Schedules

Conflict Serializable Schedules



# Other Notions of Serializability

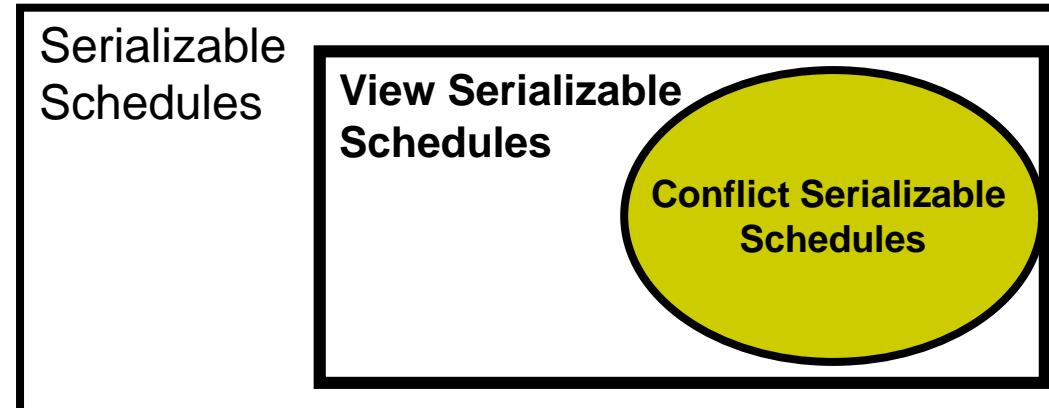
- The schedule below produces same outcome as the serial schedule  $< T_1, T_5 >$ , yet is not conflict equivalent or view equivalent to it

- Result Equivalent Schedule

- 우연에 의한 경우
- 실제 사용은 부적합

- Determining such equivalence requires analysis of operations other than read and write

$T_1$	$T_5$
read ( $A$ ) $A := A - 50$ write ( $A$ )	
	read ( $B$ ) $B := B - 10$ write ( $B$ )
read ( $B$ ) $B := B + 50$ write ( $B$ )	
	read ( $A$ ) $A := A + 10$ write ( $A$ )



- Still we can pursue potential concurrency!

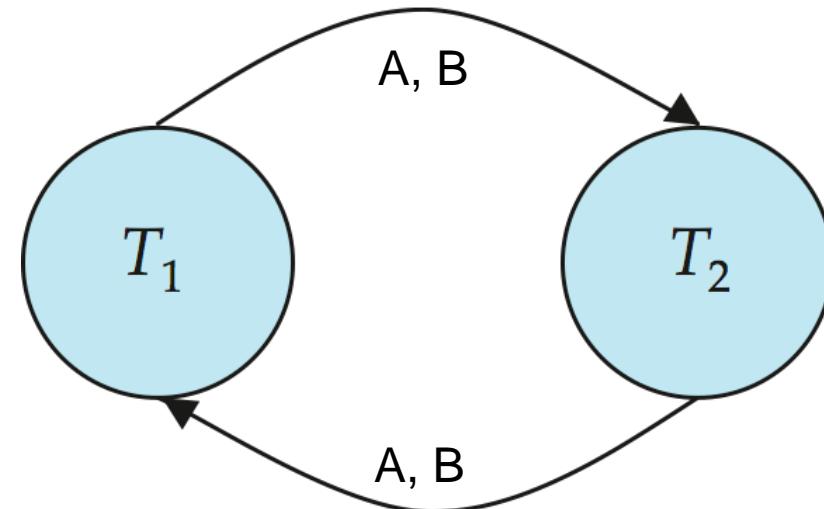


# Testing for Serializability

- Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$
- **Precedence graph** — a direct graph where the vertices are the transactions (names)
- We draw an arc from  $T_i$  to  $T_j$  if the two transaction conflict, and  $T_i$  accessed the data item on which the conflict arose earlier
- We may label the arc by the item that was accessed
- **Example 1**

$T_1$	$T_2$
<code>read(A) <math>A := A - 50</math></code>	<code>read(A) <math>temp := A * 0.1</math> <math>A := A - temp</math> <code>write(A)</code> <code>read(B)</code></code>
<code>write(A)</code> <code>read(B)</code> $B := B + 50$ <code>write(B)</code>	$B := B + temp$ <code>write(B)</code>

Fig 14.11

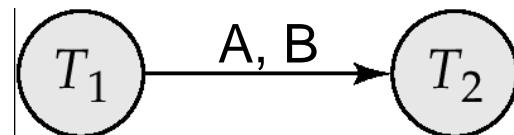




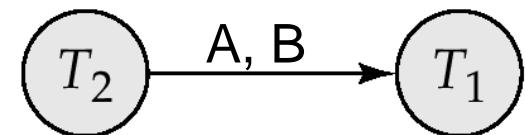
# Precedence Graph

## for (a) Schedule 1 and (b) Schedule 2

Fig 14.10



(a)



(b)

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

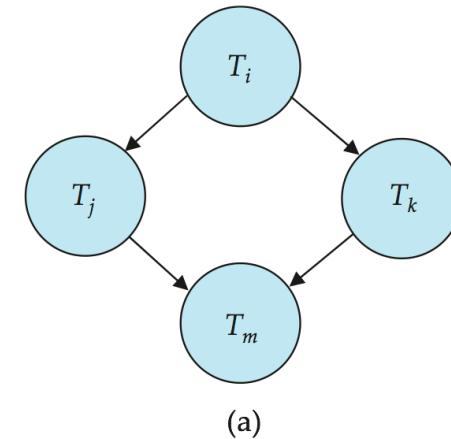
$T_1$	$T_2$
	read( $A$ )
	$temp := A * 0.1$
	$A := A - temp$
	write( $A$ )
	read( $B$ )
	$B := B + temp$
	write( $B$ )
	read( $A$ )
	$A := A - 50$
	write( $A$ )
	read( $B$ )
	$B := B + 50$
	write( $B$ )



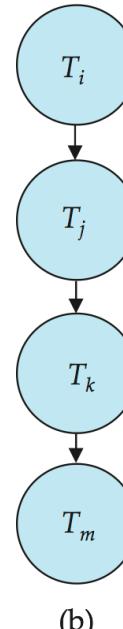
# Test for Conflict Serializability

Fig 14.12

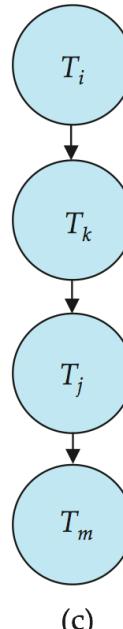
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order  $n^2$  time, where  $n$  is the number of vertices in the graph.
  - (Better algorithms take order  $n + e$  where  $e$  is the number of edges.)
- If precedence 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.
  - For example, a serializability order for Schedule A would be
$$T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$$
    - ▶ Are there others?



(a)



(b)



(c)

(참고) Cycle detection in DAG

- Depth-First Search 기반 알고리즘:  $O(n^2)$
- Kahn's 알고리즘 :  $O(v+e)$
- Shortest path 방식:  $O(n+e)$  (단, 노드에 weight 필요)



# Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability
  - Extension to test for view serializability has **cost exponential in the size of the precedence graph**
- The problem of checking if a schedule is view serializable falls in the class of **NP-complete problems**.
  - Thus existence of an efficient algorithm is *extremely* unlikely
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used



# Chapter 14: Transactions

- Transaction Concept
- A Simple Transaction Model
- Storage Structure
- Transaction Atomicity and Durability
- Transaction Isolation
- Serializability
- **Transaction Isolation and Atomicity**
- Transaction Isolation Level
- Implementation of Isolation Levels
- Transaction Definition in SQL



# Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions

- **Recoverable schedule** — if a transaction  $T_j$  reads a data item previously written by a transaction  $T_i$ , then the commit operation of  $T_i$  appears before the commit operation of  $T_j$
- The following schedule (Schedule 11) is not recoverable if  $T_9$  commits immediately after the read

Fig 14.14

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



$T_8$	$T_9$
read (A) write (A)	
	read (A)
	commit

$T_8$	$T_9$
	read (A)
	read (B)
	commit

- If  $T_8$  should abort,  $T_9$  would have read (and possibly shown to the user) an inconsistent database state
  - Hence, DBMS must ensure that schedules are recoverable



# 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)	
abort		read (A)

Fig 14.15

불행의 쌍오:  $T_{10}$ 이 commit하기 전에 unlock을 했기에

- \*\* Suppose the lock of A is released without executing the commit instruction in  $T_{10}$ . If  $T_{10}$  fails,  $T_{10}$  is supposed to be rolled back, then  $T_{11}$  and  $T_{12}$  must also be rolled back
- Can lead to the **undoing** of a significant amount of work



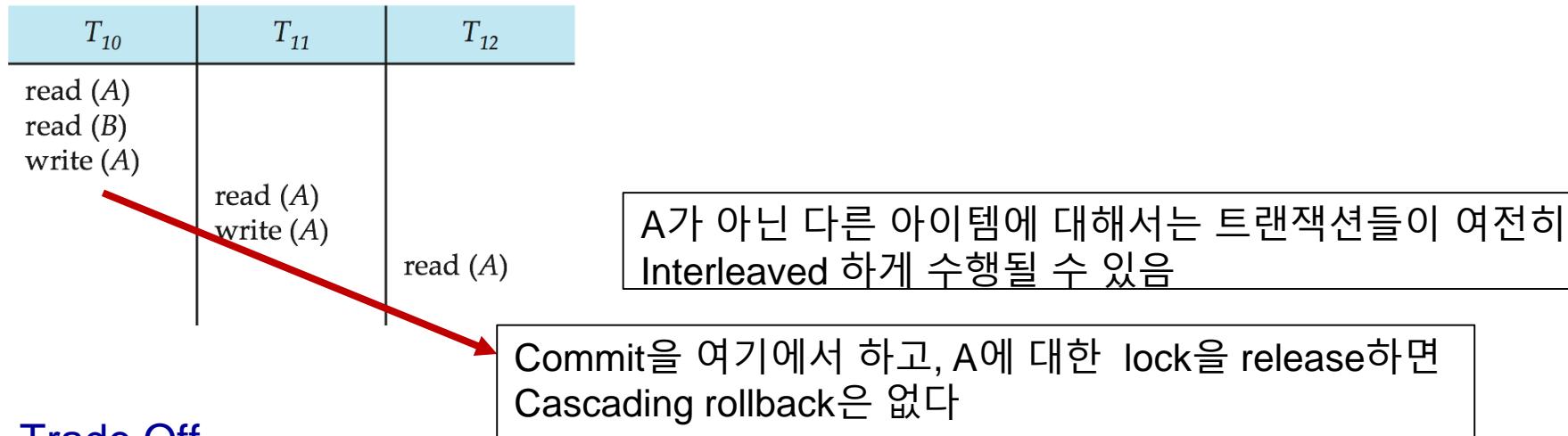
# Cascadeless Schedules

## ■ Cascadeless schedules: cascading rollbacks cannot occur

- 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

## ■ Idea: hold locks until executing the commit instruction!



## ■ Trade Off

- More concurrency → More cascading rollback
- Less cascading rollback → Less concurrency

## ■ 연구주제: Cascading Rollback을 줄이면서 Concurrency를 높이는 방법



# Chapter 14: Transactions

- Transaction Concept
- A Simple Transaction Model
- Storage Structure
- Transaction Atomicity and Durability
- Transaction Isolation
- Serializability
- Transaction Isolation and Atomicity
- **Transaction Isolation Level**
- Implementation of Isolation Levels
- Transaction Definition in SQL



# Transaction Isolation Levels in SQL-92

- **Serializable**: default
- **Repeatable read**: only committed records to be read, 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**: only committed records can be read, but successive reads of record may return different (but committed) values
- **Read uncommitted**: even uncommitted records may be read
- Lower degrees of consistency are useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
  - MSSQL server는 default로 **read committed** 지원
  - Oracle and PostgreSQL by default support a level of consistency called **snapshot isolation** (not part of the SQL standard)



# Weak Levels of DB Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - Read-only transactions need not be serializable with respect to other transactions
    - ▶ A read-only transaction that wants to get an approximate total balance of all accounts
    - ▶ Database statistics computed for query optimization can be approximate (why?)
- Tradeoff accuracy for performance
  - Weak consistency → More serializable schedules → High performance
  - Strong consistency → Less serializable schedules → Low performance

Transaction Isolation Level == DB Consistency Level



# Chapter 14: Transactions

- Transaction Concept
- A Simple Transaction Model
- Storage Structure
- Transaction Atomicity and Durability
- Transaction Isolation
- Serializability
- Transaction Isolation and Atomicity
- Transaction Isolation Level
- **Implementation of Isolation Levels**
- Transaction Definition in SQL



# Implementation of Isolation Levels by Concurrency Control Schemes

- A database must provide a CC scheme ensuring all possible schedules are
  - either [conflict](#) or [view serializable](#), and
  - are [recoverable](#) and [preferably cascadeless](#)
  - We study such protocols in [Chapter 16](#)
    - ▶ [Locking based CC Schemes](#)
    - ▶ [Timestamps based CC Schemes](#)
    - ▶ [Multiple Version CC Schemes](#)
    - ▶ [Snapshot Isolation](#)
- **Contradictory Goal of CC Schemes:** High Concurrency and High Isolation Level
- Concurrency-control schemes tradeoff between [the amount of concurrency](#) they allow and [the amount of overhead](#) that they incur
  - A policy in which only one transaction can execute at a time generates serial schedules in a simple way, but provides a poor degree of concurrency
  - Some schemes allow only [conflict-serializable schedules](#) to be generated, while others allow [view-serializable schedules](#) that are not conflict-serializable



# Concurrency Control vs. Serializability Tests

- Concurrency control protocols generally do not examine **the precedence graph** as it is being created
  - Testing a schedule for serializability *after* it has executed is a little too late!
  - Instead a protocol imposes a discipline that avoids nonserializable schedules
  - Tests for serializability help us understand **why a concurrency control protocol is correct**
  
- Serializable schedule만을 만들어내는 CC Scheme이 없이 그냥 concurrent하게 transaction들을 수행하고 precedence graph로 serializability를 판단한다면 어떨까?
  - Precedence Graph로 Serializability를 판단하는 것은 run-time에 어떤 트랜잭션들이 들어올지 모두 알고 있어야 하므로 비현실적
  - 만약 동시에 들어오는 트랜잭션들을 실시간으로 precedence 그래프를 그려서 serializability 판단한다고 해도 unserializable schedule들은 다 abort시켜야 하므로 비용이나 성능측면에서 비현실적임



# Chapter 14: Transactions

- Transaction Concept
- A Simple Transaction Model
- Storage Structure
- Transaction Atomicity and Durability
- Transaction Isolation
- Serializability
- Transaction Isolation and Atomicity
- Transaction Isolation Level
- Implementation of Isolation Levels
- **Transaction Definition in SQL**



# Transaction Definition in SQL

- Data manipulation language (DML) must include a construct for specifying the set of actions that comprise a transaction
- In SQL, **a transaction begins implicitly**
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one
  - **Rollback work** causes current transaction to abort
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
  - In DBMS Schell, `setAutoCommit(false)`, 개발자가 원하는 시점에 `commit()` 가능
  - In JDBC, `connection.setAutoCommit(false); connection.commit()`



# Transaction as SQL Statements

- The current transaction model is based on `read(Q)` and `write(Q)` operations
- Concurrent execution of SQL statements may cause some strange situation
- T1: `select ID, name from instructor where salary > 90000;`
- T2: `insert into instructor values ('11111', 'James', 'Marketing', 100000)`
- T3: `update instructor set salary = salary * 0.9 where name = 'Wu'`
  
- Clearly there is a conflict in T1 and T2, but our current txn model cannot capture the conflict → T1 and T2를 Interleaving해서 실행시킨다
  - { T1; T2} 혹은 { T2; T1} 으로 2개의 transaction을 독립적으로 수행해야
  - Phantom Phenomenon: A conflict exists on phantom data
- T1과 T3도 interleaving하면 안되는데, simple CC scheme들은 interleaving 시킴
- SQL Statement 수준에서의 CC Schemes (Real World DBMS의 CC)
  - Index Locking Protocol: relation에 update를 할때에 index structure에도 locking
  - Predicate Locking Protocol: query의 predicate (where clause)와 conflict이 나는 update들에 locking