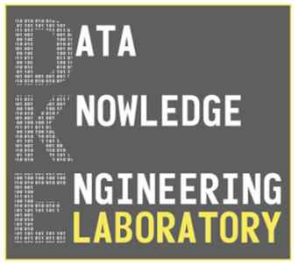


TRANSACTION PROCESSING I

Chapter 20

- Transaction Concepts

Prof. Young-Kyoon Suh

Introduction

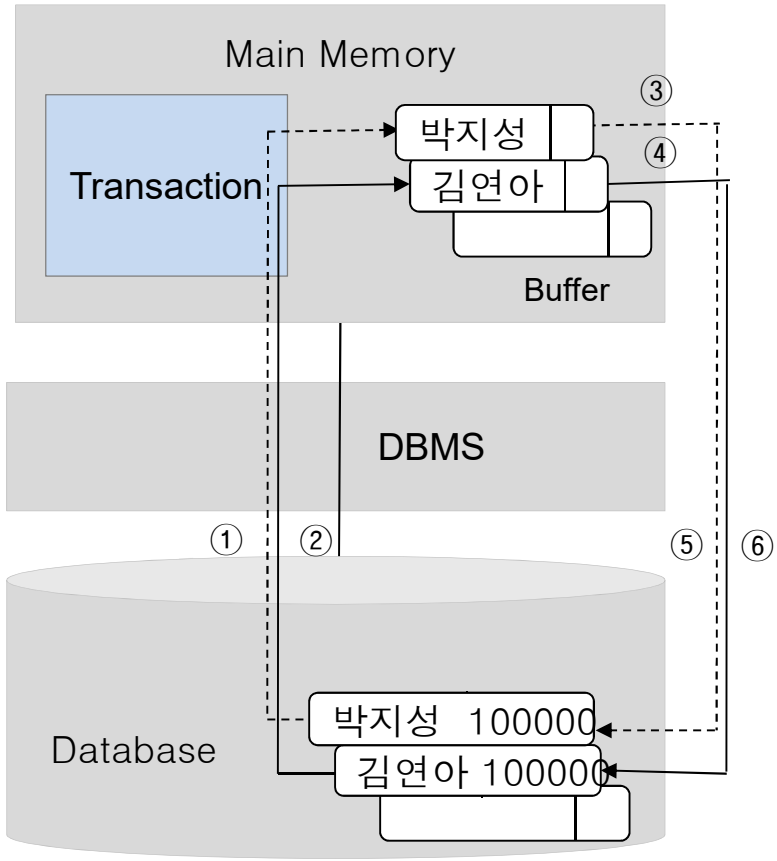
- *Transaction* processing systems (트랜잭션 처리 시스템)
  - Systems with "*large databases*" and "hundreds of *concurrent* users" executing database **transactions**.
  - Examples: *airline reservations*, *banking*, *credit card processing*, *online retail purchasing*, *stock markets*, *supermarket checkouts*, and *many other applications*.
- What is ***transaction***?
  - A **logical unit of database processing** that *must be completed in its entirety* to ensure "correctness and consistency"
  - "An executing (computer) program"
    - Typically includes database commands: *retrievals*, *insertions*, *deletions*, and *updates*.
    - E.g., Project Phase 3

Introduction – Transaction Example (Cont'd)

- Scenario: 박지성 wants to send some money to 김연아.

```
START TRANSACTION
① /* 박지성 계좌를 읽어온다 */
② /* 김연아 계좌를 읽어온다 */
  /* 잔고 확인 */
③ /* 예금인출 박지성 */
UPDATE Customer
SET balance=balance-10000
WHERE name='박지성';
④ /* 예금입금 김연아 */
UPDATE Customer
SET balance=balance+10000
WHERE name='김연아';
COMMIT /* 부분완료 */
⑤ /* 박지성 계좌를 기록한다 */
⑥ /* 김연아 계좌를 기록한다 */
```

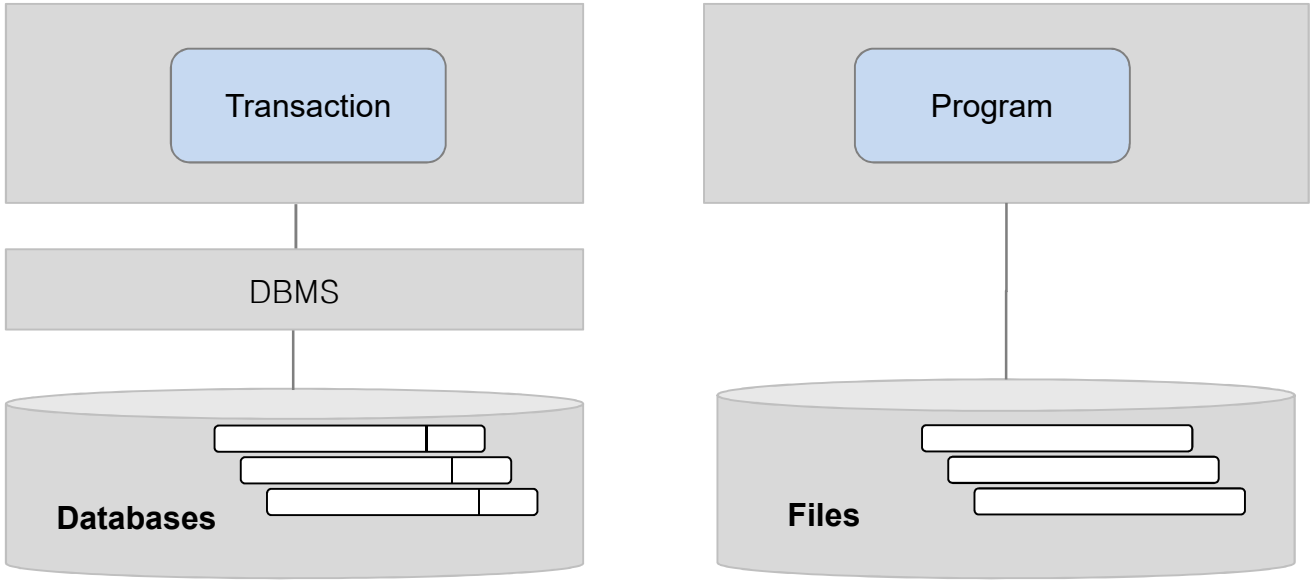
(a) 계좌이체 트랜잭션 (Balance Transfer Transaction)



(b) 트랜잭션 수행 과정 (Transaction Execution Process)

Introduction - Difference Between Transaction and General Program (Cont'd)

Item	Transaction	Program (in C)
Program Structure	*BEGIN TRANSACTION* ... *COMMIT TRANSACTION*	`main()` `{ ... }`
Data to be dealt with	데이터베이스 저장된 데이터	파일에 저장된 데이터
Translator	DBMS	Compiler
Properties	ACID (to be discussed later)	–



Introduction – Terminology (Cont'd)

- Types of transaction
  - *Read-only* transaction: only reads; don't update database
  - *Read-write* transaction: otherwise.
- Major terms in transaction processing concepts
  - *Data item*: a "database record" (tuple), a "disk block", a "field value", a "whole file", or even the "whole database"
    - Has a unique name, used as means to uniquely identify each data item
  - *Granularity*: the size of the data item
  - *Database*: a collection of "named data items"
- Database access operations that a transaction can include:
  - `read_item(X)`: Reads a data item, X, into a program variable (X)
  - `write_item(X)`: Writes the value of X into the data item named X

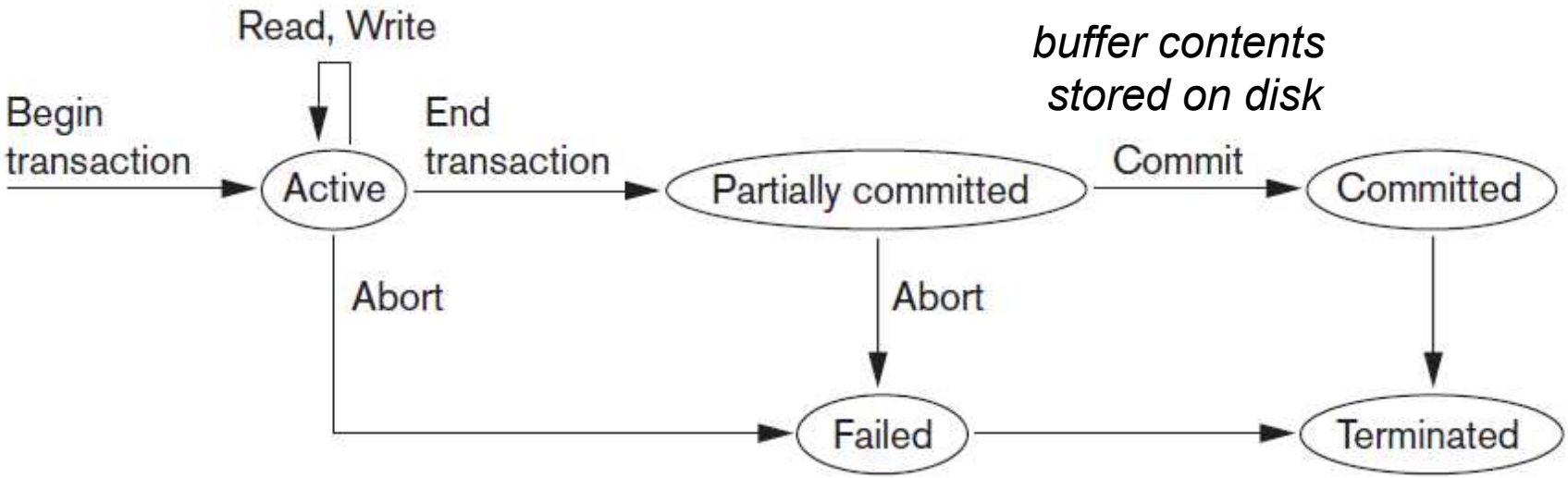
TRANSACTION AND SYSTEM CONCEPTS, AND DESIRABLE PROPERTIES OF TRANSACTIONS

Chapters 20.2 and 20.3

Transaction States (트랜잭션 상태)

- The (recovery manager of the) DBMS must keep track of the following operations (`start`, `terminate`, `commit`, and/or `abort`):
  - **`BEGIN_TRANSACTION`**:
    - This marks the **beginning** of transaction execution.
  - **`READ`** or **`WRITE`**:
    - These specify **read** or **write** operations on the database items that are executed as part of a transaction.
  - **`END_TRANSACTION`**:
    - Transaction operations of `READ` or `WRITE` have ended and marks the end of transaction execution.
    - Necessary to check the status of that transaction: ***committed*** or ***aborted***.
  - **`COMMIT_TRANSACTION`**:
    - Signals a successful end of the transaction; changes **recorded permanently**
  - **`ROLL_BACK`** (or **`ABORT`**):
    - Signals that the transaction has ended unsuccessfully; must be **undone**

Transaction States (Cont'd)



- Partially committed: Why? For fast response to user
  - The DBMS reflects changes on disk at a time rather than each transaction.
  - Some *concurrency control* protocols additionally checks for possible commit.
  - Some *recovery protocols* check failure and its effect on database
- Failed: if one of the checks fails, or if the transaction is aborted
  - Failed or aborted transactions may be restarted (by user or automatically).
- Terminated: the transaction leaves the system.

Why *Recovery* Is Needed? To Handle Failure

- When a transaction is submitted to a DBMS for execution, the terminal status of that transaction is determined by the DBMS and is in **two** cases:
  - ***Committed***:
    - All the *operations* in the transaction are *completed successfully* and their *effect* is *recorded permanently* in the database, or
  - ***Aborted***:
    - The transaction has **NO** effect on the database or any other transactions.
- If a transaction **fails** after executing some of its operations but before executing all of them,
  - The operations already executed *must be undone* and have **NO** lasting effects.

Why *Recovery* Is Needed? (Cont'd)

- Failures: classified as transaction, system, and media failures
- Types of failures

  1) A *computer failure*: hardware/software/network error occurs in the computer system during transaction execution; memory crashes

  2) A *transaction* or *system error*: such as integer overflow, division by zero; or user interrupt

  3) *Local errors* or *exception conditions detected by the transaction*: an exception condition like insufficient balance

  4) *Concurrency control enforcement*: due to serializability violation or deadlock detection

  5) *Disk failure*: malfunctioning disk blocks

  6) *Physical problems and catastrophe*: fire, theft, power failures …

- *The DBMS must keep sufficient information to recover quickly from Types 1-4 failures (more common than the others).*

The **System Log** (시스템 로그)

- Maintained by the DBMS to recover from failure
- Keeps track of transaction operations that affect the values of database items
- *Sequential*, *append-only* file stored on disk
  - Safe, except disk or catastrophic failure
- ***Log buffer***: one (or more) main memory buffers used to record logs
  - When full with log entries, appended to “end” of log file on disk
- Log file is periodically backed up to archival storage
  - Against any catastrophic failures

The System Log (Cont'd)

- Types of "Log Records":
  1. [`start_transaction`, T]
  2. [`read_item`, T, X]
  3. [`write_item`, T, X, old_value, new_value]
  4. [`commit`, T]
  5. [`abort`, T]
     - T: a unique transaction identifier generated automatically
  - If the system *crashes*, we can recover to a consistent database state by examining the log and using a recovery technique like ARIES.
- UNDO operations: restore X to old_value
  - Undo the effect of `WRITE` operations of T by tracing *backward*
- REDO operations: updates X to new_value
  - Can be performed if a transaction has its updates recorded in the log but a failure occurs before recording new_value into the database

Commit Point of Transaction T

- Occurs when all its operations that access the database have *completed successfully* and *effect of operations recorded* in the log; Beyond this, T is said to be ***committed***.
- T then writes into the log a commit record: [`commit`, T].
- When failure occurs,

  1) If T has its commit record, *redo their effect* on the database from their log records.

  2) If T has no commit record but has `start_transaction` record, *undo their effect* on the database during the recovery process: *rollback*

- "Force-writing": flushes the log buffer to disk *before* committing a transaction.
  - Indicates writing to disk any portion of the log that hasn't been written disk yet before a transaction reaches its commit point
  - Allows us to consider only log entries stored on disk during recovery.

ACID Properties of Transaction

- Must be possessed by any transaction.
- **A**tomicity: responsibility of recovery subsystem
  - A transaction is an atomic unit of processing: *all* or *nothing*.
- **C**onsistency: responsibility of programmer
  - A transaction should be consistency preserving: consistent database -> consistent database if no interference
- **I**solation: enforced by concurrency control subsystem
  - A transaction should appear as if it is being executed in isolation from other transactions; its execution shouldn't be interrupted by others
- **D**urability: responsibility of recovery subsystem
  - Changes applied by a committed transaction must persist in the database.
    - Shouldn't be lost by because of any failure

CHARACTERIZING SCHEDULES BASED ON SERIALIZABILITY

Chapters 20.5

- Also, refer to the Wiki page:
https://en.wikipedia.org/wiki/Schedule_(computer_science)

Schedules of Transactions

- A *schedule* S of n (concurrent) transactions T_1, T_2, …, T_n:
  - An ordering of the operations of the transactions
  - Operations (reads/writes) from different transactions may be *serial* or *interleaved* in S.
  - Example of a schedule (S_a) of interleaving two transactions, T_1 and T_2:
    - S_a: $r_1(X)$; $w_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;
- *Total ordering* of operations in S
  - For any two operations in S, one operation must occur before the other operation.
- *Partial ordering* of operations in S
  - Used if order in time between certain operations is not determined by the system

Serial Schedule

- A schedule S is *serial* if, for every transaction T in S, all the operations of T are *executed consecutively* without any interleaved operations from the other operations
  - Otherwise, called *nonserial*.
- Note that every serial schedule is **CORRECT**, but …
- "Problems" with serial schedules
  - They limit concurrency by prohibiting interleaving of operations
  - Cannot wait for another transaction to be done; *Unacceptable* in practice
- Solution: determine which schedules are "equivalent" to a serial schedule and allow those schedules to occur
- But not easy to determine which one is equivalent…

Conflicting Operations in a (Nonserial) Schedule

- Two operations in a schedule are said to be ***conflict*** if
  (1) Operations belong to *different transactions* .
  (2) Operations access the *same* item X, and
  (3) *At least one* of the operations is `write_item`(X).
  (All the above three conditions are to be satisfied to be conflict.)

- Intuitively, two operations are ***conflict*** if changing their order results in a different outcome.

- Two types of conflict: i) *read-write* or ii) *write-write* conflicts
  - E.g., S_a: $r_1(X)$; $w_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;

- Therefore, we need a sophisticated *concurrency control* (동시성 제어) *technique* to resolve not only these conflicts but also “others.”

Why **Concurrency Control** Needed in a Schedule?

- Consider the following transactions that may run concurrently

T_1

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);

T_2

read_item(X);
$X := X + M$;
write_item(X);

Why **Concurrency Control** Needed in a Schedule (Cont'd)

$X = 90;\ Y = 90;$
$N = 3;\ M = 2;$

Serial

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule B

$X = 89;\ Y = 93;$

Nonserial: interleaves operations of T_1 and T_2

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule C: $X = 92;\ Y = 93;$

T_1	T_2

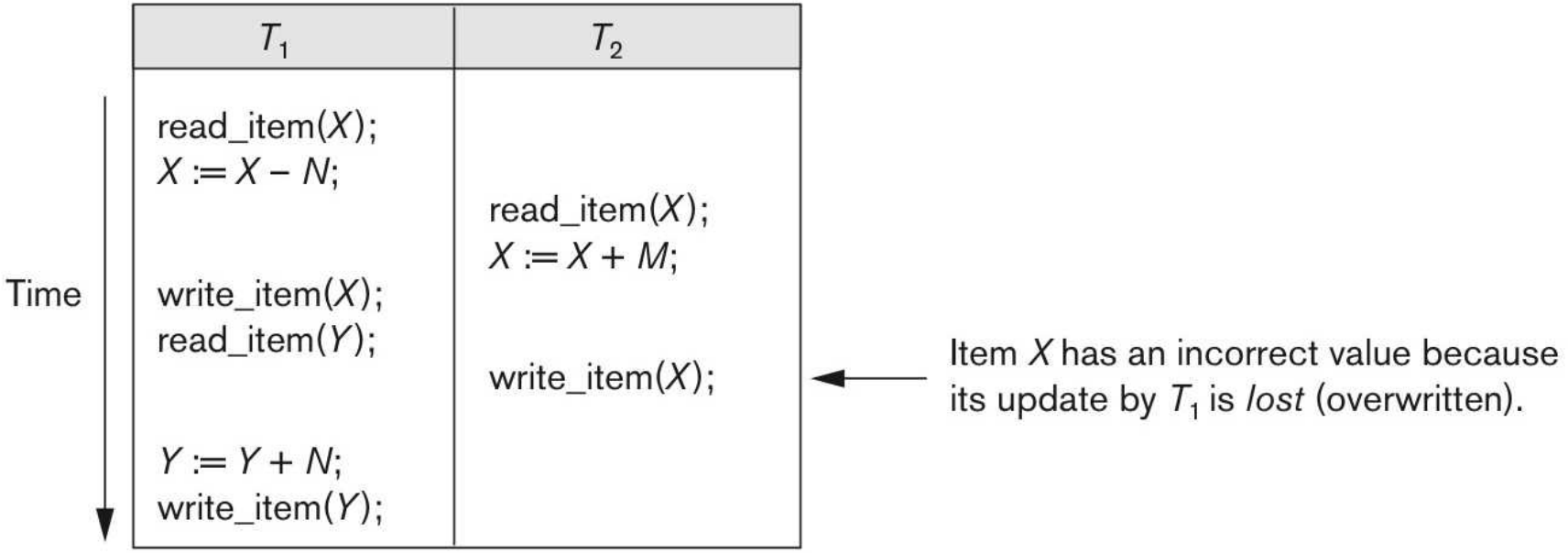
read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule D

Why **Concurrency Control** Needed in a Schedule - Types of Problems (Cont'd)

1) The ***lost update*** (갱신 손실) problem

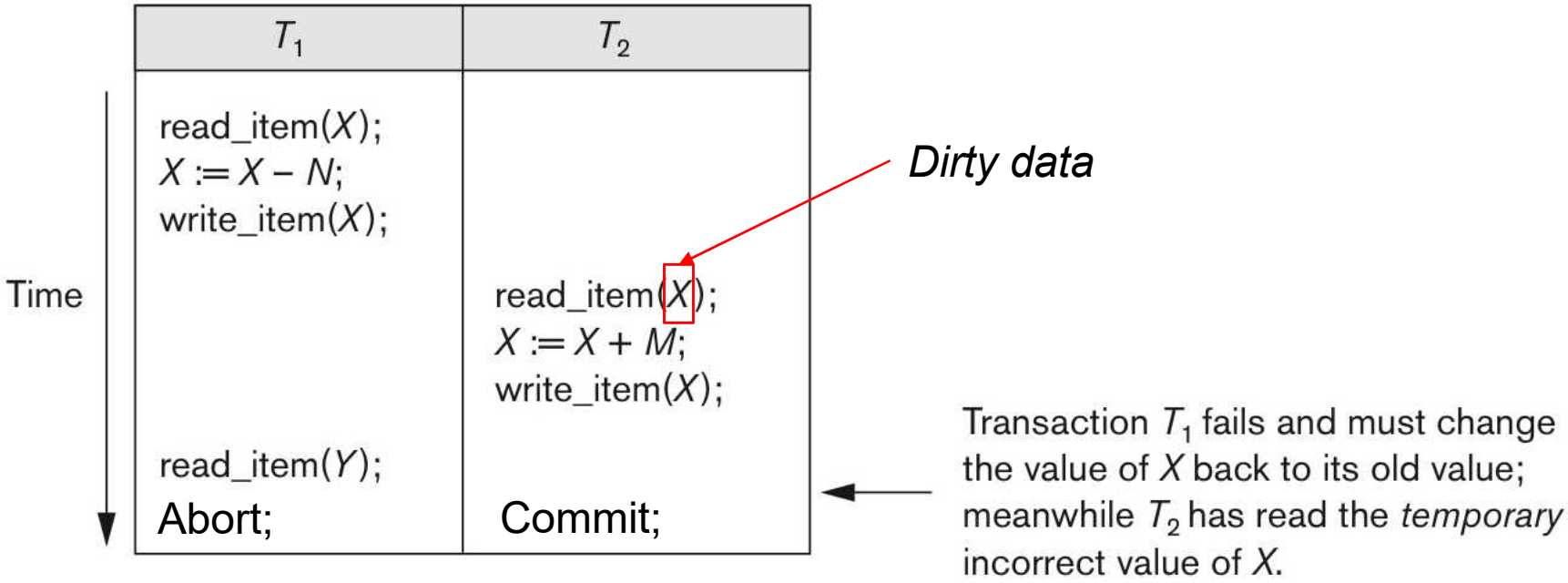
- Occurs when two transactions accessing the "same" database items have their operations *interleaved* in a way that makes the value of some database items incorrect



Why **Concurrency Control** Needed in a Schedule - Types of Problems (Cont'd)

2) The ***dirty read*** (or *temporary update*) problem

- Occurs when one transaction *updates* a database item and then the transaction *fails* for some reasons.
- Meanwhile, the updated item is *read* by another transaction before it is changed (or rolled back) to its original value.



Why **Concurrency Control** Needed in a Schedule - Types of Problems (Cont'd)

3) The ***incorrect summary*** problem (called phantom read)

- If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, then the aggregate function may calculate summary *before* and *after* the values are updated.

T_1	T_3

	sum := 0;
	read_item(*A*);
	sum := *sum* + *A*;
	. . .

read_item(*X*);
X := *X* − *N*;
write_item(*X*);	

	read_item(*X*);
	sum := *sum* + *X*;
	read_item(*Y*);
	sum := *sum* + *Y*;

read_item(*Y*);
Y := *Y* + *N*;
write_item(*Y*);	

T_3 would like to sum up all the values of A, …, X, and Y.

← T_3 reads *X* after *N* is subtracted and reads *Y* before *N* is added; a wrong summary is the result (off by *N*).

Or, it could be the case for T_1 to insert a new record matching a selection condition by T_3.

Why **Concurrency Control** Needed in a Schedule - Types of Problems (Cont'd)

4) The ***unrepeatable read*** problem

- Occurs when a transaction T *reads* the same item *twice* and the item is changed by another transaction T' between the two reads.
- Hence, T receives *different* values for its two reads.

To solve these problems and produce a nonserial but *serializable* schedule, we need a good concurrency control.

Serializable (직렬가능한) Schedule

- *Equivalent* to some serial schedule of the same n transactions
- Places *simultaneous transactions* in series
  - Suppose S_1 is a serial schedule ($T_1;T_2 \mid T_2;T_1$), and S_2 is a nonserial one.
  - If S_1 and S_2 produce the same final state of the database, then S_1 and S_2 may be *equivalent*.
    - S_2 is said to be *serializable*.
- Always considered to be correct when concurrent transactions are executing; gives benefit of *concurrent execution*
- [Caution] two schedules that are result equivalent for the initial value of X = 100 but are not result equivalent in general

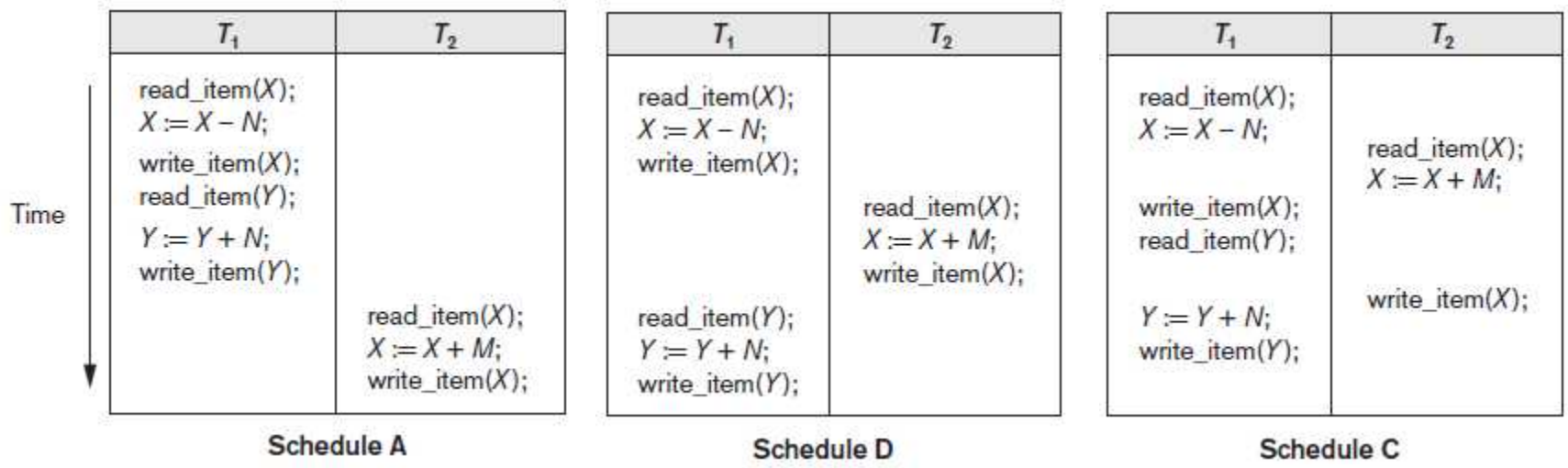
S_1

read_item(X);
$X := X + 10$;
write_item(X);

S_2
read_item(X); $X := X * 1.1$; write_item (X);

Serializable Schedule (Cont'd)

$X = 90;\ Y = 90;$
$N = 3;\ M = 2;$



(Serial)

(**Serializable**)

(Non-serializable)

AFTER: $X = 89;\ Y = 93;$

$\boldsymbol{X = 89;\ Y = 93;}$

$X = 92;\ Y = 93;$

How to Ensure *Serializablity*?

- Use ***concurrency control*** (CC) ***protocols***:
  - *Two-phase locking* (2PL): most common technique;
    - *Locks* data items to prevent concurrent transactions from interfering with one another
    - Two phases: growing (acquiring) and shrinking (releasing) phases on locks
    - Enforces an additional condition that guarantees serializablity
  - *Snapshot isolation*: used by some DBMSes; less overhead than 2PL
    - Transaction sees data items that it reads based on the committed values of the items in the database snapshot when transaction starts
    - Ensures phantom record problem will not occur
  - *Timestamp ordering*
    - Each transaction is assigned a unique timestamp
    - The protocol ensures that any conflicting operations are executed in the order of the transaction timestamps

How to Ensure *Serializablity*?

- *Multiversion concurrent control* (MVCC) *protocols*
  - Based on maintaining multiple versions of data items

- *Optimistic protocols*:
  - Checking for possible serializability violations after the transactions terminate but before they are permitted to commit.

TRANSACTION SUPPORT IN SQL

Chapter 20.6

- We'll cover more in labs.

Transaction Support in SQL

- No explicit `Begin_Transaction` statement, in general.
- Instead, every transaction must have an explicit end statement:
  - **`COMMIT`** | **`ROLLBACK`**.
- Every transaction has certain characteristics associated with it.
  - Can by specified by its **isolation level**.
  - How to specify isolation level in SQL: **`ISOLATION LEVEL`** <*isolation*>
    - *isolation* = `READ UNCOMMITTED` | `READ COMMITTED` (by default in Oracle)
      `REPEATABLE READ` | `SERIALIZABLE` (by default in most DBMSes)
- If a transaction executes at a lower level than `SERIALIZABLE`,
  - One or more of the following three violations may occur.
  - See next slides.

What Happens if Serializability is Violated?

- ***Dirty read*** (오손 읽기)
  - T_1 may read the update of T_2, which has not yet committed.
  - If T_2 aborts, then would have read a value that doesn't exist and is incorrect.
- ***Nonrepeatable read*** (반복 불가능)
  - T_1 may read a given value from a table.
  - If T_2 later updates that value, then T_1 reads that value again, T_1 will see a different value.
- ***Phantoms*** (유령 데이터)
  - T_1 may read a set of rows from a table, perhaps based on some condition specified in the SQL `WHERE` clause.
  - T_2 inserts a new record that satisfies the condition.
  - T_1 then "may" see a new record, called *phantom*, into the same table.
  - T_1 didn't see the record when it started.

What Happens if Serializability is Violated? (Cont'd)

- Possible violations based on isolation levels as defined in SQL

	Type of Violation		
Isolation Level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

If isolation level is set to be higher, any problem?

Summary

- Uncontrolled execution of concurrent transactions
  - Reveals dirty read, lost updates, non-repeatable read, incorrect summary or phantom read
- Committed transaction
- Schedule of transactions
  - Serial/nonserial (serializable/nonserializable)
- Serializability of schedules
- Concurrency control
- Failure recovery

APPENDIX

Recoverable Schedule

- Transactions (T_1) commit only after *all the transactions (T_2)* that change item A that the transactions (T_1) read, commit.
- Example: S_1 and S_2 are recoverable.

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
$Com.$	
	$Com.$

S_1

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
$Abort$	
	$Abort$

S_2

Unrecoverable Schedule

- If a transaction T_1 aborts, and a transaction T_2 commits, but T_2 relied on T_1, then the schedule is *unrecoverable*.
  - An unrecoverable schedule should NOT be permitted by the DBMS.
  - Example: S_1 are unrecoverable.

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$Com.$
$Abort$	

S_1

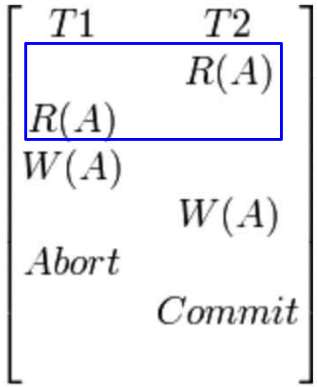
Cascading Rollback (or *Aborts*)

- An uncommitted transaction has to be rolled back.
  - Note that no committed transaction ever needs to be rolled back.
- May occur in some recoverable schedule; but should not occur
  - Example: S_e leads to cascading rollback of T_2
    - S_e: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1; a_2

Cascadeless Schedule

- Avoids cascading rollback if
  - Every transaction in S reads only item that *were written by committed transactions*.
  - Example 1: $S_{e'}$ is a <u>cascadeless</u> schedule (avoiding cascading-aborts)

$S_{e'}$



  - Example 2: $S_{e''}$ is not a <u>cascadeless</u> schedule
    - $S_{e''}$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; a_1; a_2
      - If T_1 aborts, T_2 must be *rolled back*, as T_2 reads X written by T_1 that aborts now.

Strict Schedule

- Transactions can *neither read nor write* an item X *until* the *last transaction* that wrote X has *committed* or *aborted*
  - Simpler recovery process
  - Example: $S_{e''''}$ are a strict schedule
    - $S_{e''''}$: $r_1(X)$; $w_1(X)$; c_1; $r_3(X)$; $w_3(X)$; c_3; $r_2(X)$; $w_2(X)$; c_2

Serializable Schedule: Another Example

- Q: How can we run these transactions concurrently?

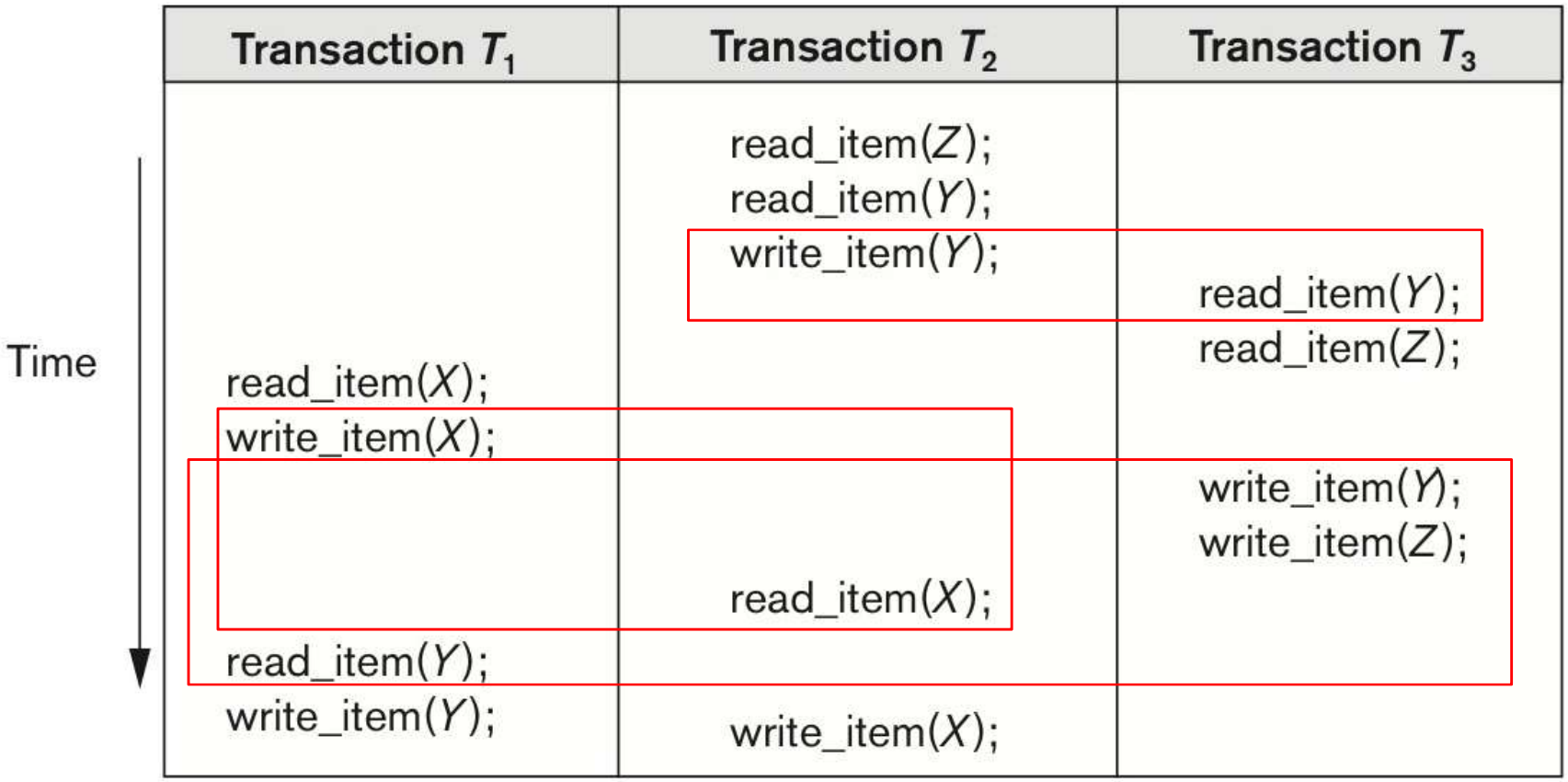
Transaction T_1
read_item(X);
write_item(X);
read_item(Y);
write_item(Y);

Transaction T_2
read_item(Z);
read_item(Y);
write_item(Y);
read_item(X);
write_item(X);

Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);

Serializable Schedule: Another Example

- Schedule E: Not serializable



Schedule E

Serializable Schedule: Another Example

- Schedule F: Serializable

Time

Transaction T_1	Transaction T_2	Transaction T_3
		read_item(Y);
		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(Z);	
read_item(Y);		
write_item(Y);		
	read_item(Y);	
	write_item(Y);	
	read_item(X);	
	write_item(X);	

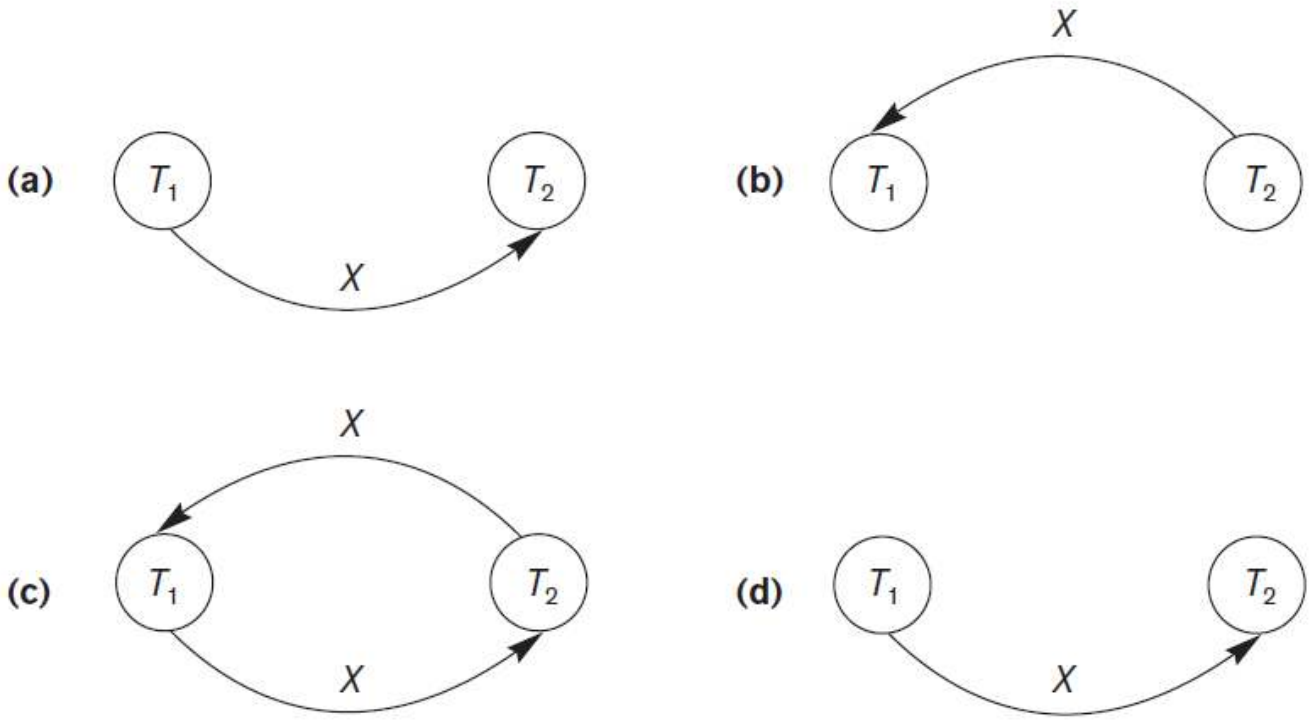
Schedule F

Testing for Serializability of a Schedule

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

Constructing the Precedence graphs

- For schedules A to D to test for conflict serializability



Levels of Isolation of a Transaction

- Level 0 (READ UNCOMMITTED)
  - Doesn't overwrite the dirty reads of higher-level transactions
- Level 1 (READ COMMITTED)
  - Has no lost updates
- Level 2 (REPEATABLE READ)
  - No lost updates and no dirty reads
- Level 3 (SERIALIZABILITY)
  - Has repeatable reads, in addition to 2 properties
- Snapshot isolation
  - Another type of isolation