Transaction Processing Concepts

↳ 쪼갤수없는작업단위

↱ undo

transaction fail난경우? (transaction 중도에) ⇒ 없던것으로 처리해야함!

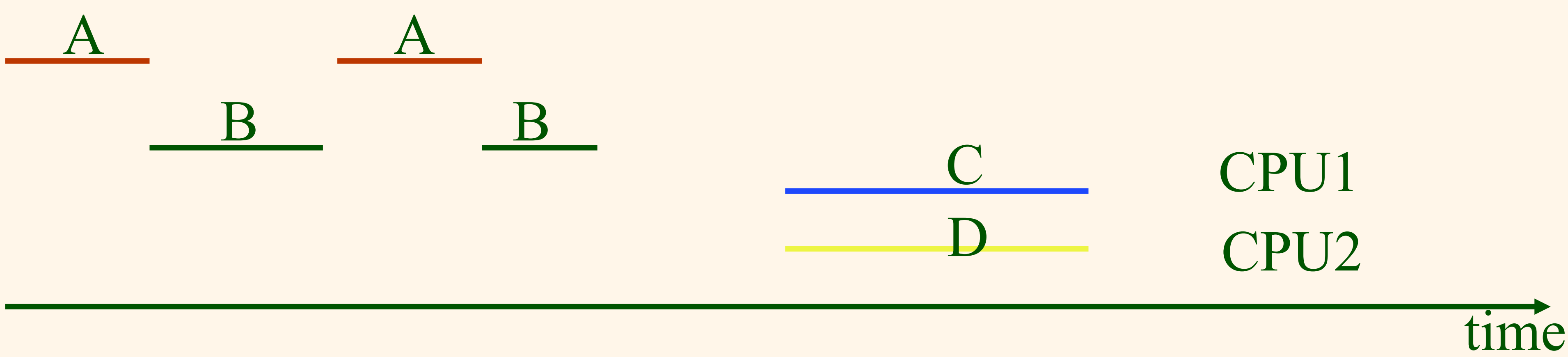
read/write 후 commit을 call해야 하나의 transaction이 끝남

log에 commit이 없는경우 transaction fail → recovery작업 필요

↳ roll back

Single-User VS Multi-user Systems

- Single-user vs multi-user
  - A DBMS is single-user if at most one user at a time can use the DBMS.
  - A DBMS is multi-user if many users can use the DBMS simultaneously.
- Interleaved processing vs parallel processing



Transactions

read
write

- Concurrent execution of user programs is essential for good DBMS!

동시성 제공 이유

  - Usability: Multiple users or applications need to (can) access databases simultaneously.
  - Performance : Since disk access speed is relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- A *transaction* is the DBMS's abstract view of a user program: a logical unit of database processing that includes the sequence of reads and writes.

Transactions

- Conventional notations
  - T_1, T_2 , T_n : transactions
  - R(X) : Reads a database item named X into a variable X.
  - W(X) : Write the value of X into the database item named X.
  - Read-set(T_1) : set of items that the transaction T_1 reads.
  - Write-set(T_1) : set of items that the transaction T_1 writes.

11/21 [

transaction
: sequence를 모아둔 project unit

종윤. 진수

Transactions

- Conventional notations Cont'd
  - Example:
    - 계좌이체 T1 is a transaction that transfers the balance (N) from an account (X) to the other account (Y)
    - 입금 T2 is a transaction that deposits the amount (M) into an account (X).

operation이 아님 임의 X에 저장

T_1: BEGIN R(X), X=X-N, W(X), R(Y), Y=Y+N, W(Y) END
T_2: BEGIN R(X), X=X+M, W(X) END

- Read-set(T1)? Write-set(T1) ? Read-set(T2)? Write-set(T2) ?

Read-set(T1) = {X, Y}, Write-set(T1) = {X,Y}
Read-set(T2) = {X}, Write-set(T2) = {X}

순차적 serial : 하나끝난다음실행

↔ 동시수행 interleaving

Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - integrity constraints
    - DBMS will enforce ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- *Issues:* Effect of *interleaving* transactions, and *crashes*.

Anomalies with Interleaved Execution

- The Lost Update Problem 고친게 없어짐!

T1: BEGIN R(X), X=X-N, W(X), R(Y), Y=Y+N, W(Y) END
T2: BEGIN R(X), X=X+M, W(X) END

→ time

- Let's say that X = 100, Y=0 at the start and N = M = 50. Then after processing T1 and T2, what are the values of X and Y ?
  - X should be 100.
  - Y should be 50.

Anomalies with Interleaved Execution (Cont'd)

- The Lost Update Problem (Cont'd)

T1: R(X), X=X-N, W(X), R(Y), Y=Y+N, W(Y)
T2: R(X), X=X+M, W(X)

time

Let's say that X = 100, Y=0 at the start and N = M = 50.
X : 50 150
100 이어야됨!
Y: 50

Anomalies with Interleaved Execution (Cont'd)

- The Lost Update Problem (Cont'd)

X has incorrect value because its update by T1 is "lost".

lost!

T1: R(X), X=X-N,		W(X),	R(Y),		Y=Y+N, W(Y)
T2:	R(X), X=X+M,			W(X)	

time

Let's say that X = 100, Y=0 at the start and N = M = 50.

X :	50	150	
Y:			50

Anomalies with Interleaved Execution (Cont'd)

cf) dirty bit : 수정된 것 표시 (상태보관)
↳ 일어났는지 판별 중요

아직 작업이 안 끝남

2 ▫ Reading Uncommitted Data ("dirty reads")

↳ read만 해도 dirty해졌다고 간주
cf) write했지만 실제 바뀐 내용은 없는 경우

failure

T1: BEGIN R(X), X=X-N, W(X), R(Y), Y=Y+N, W(Y) END
T2: BEGIN R(X), X=X+M, W(X) END

time

- Let's say that X = 100, Y=0 at the start and N = 50, M = 10. Then after processing T1 and T2,
  - X should be 110.
  - Y should be 0.

Anomalies with Interleaved Execution (Cont'd)

- Reading Uncommitted Data ("dirty reads") Cont'd

얘도 undo

ex) 입금하는 중에 정전

T1: R(X), X=X-N, W(X)	R(Y), *fails then abort.*
T2:	R(X), X=X+M, W(X)

time →

Let's say that X = 100, Y=0 at the start and N = 50, M = 10.

X : 50 60

Y: 0

Anomalies with Interleaved Execution (Cont'd)

- Reading Uncommitted Data ("dirty reads") Cont'd

T2 has read the "uncommited value" of X.

T1: R(X), X=X-N, W(X) R(Y), *fails then abort.*
T2: R(X), X=X+M, W(X)

time

Let's say that X = 100, Y=0 at the start and N = 50, M = 10.
X : 50 60
Y: 0

fail됐는데도 값이 바뀜
110이 아니라 60

Anomalies with Interleaved Execution (Cont'd)

- Incorrect summary 결과값은 맞음

T1: BEGIN Sum=0, R(X), Sum=Sum+X, R(Y), Sum+=Y, print(Sum) END
T2: BEGIN R(X), X=X-N, W(X), R(Y), Y=Y+N, W(Y) END

- T1 is a transaction that reports the sum of the balance of two accounts (X and Y).
- T2 is a transaction that transfers the balance (N) from an account (X) to the other account (Y)
- Let's say that X = 100, Y=0 at the start and N = 50. Then after processing T1 and T2,
  - Sum should be either 100.

Anomalies with Interleaved Execution (Cont'd)

- Incorrect summary (Cont'd)

T1: Sum=0, R(X), Sum=Sum+X, R(Y), Sum+=Y, print(Sum)
T2: R(X), X=X-N, W(X), R(Y), Y=Y+N, W(Y)

Let's say that X = 100, Y=0 at the start and N = 50.

X : 50

Y: 50

Sum: 0 50 50 **50**

Anomalies with Interleaved Execution (Cont'd)

- Incorrect summary (Cont'd)

T1 reads X after N is subtracted but before N is added to Y

T1: Sum=0, R(X), Sum=Sum+X, R(Y), Sum+=Y, print(Sum)
T2: R(X), X=X-N, W(X), R(Y), Y=Y+N, W(Y)

Let's say that X = 100, Y=0 at the start and N = 50.

X : 50

Y: 50

Sum: 0 50 50 50

Wait for a second ……

- Why do we learn about several different kinds of anomalies ?
  - DBMS should allow multiple transactions can be processed concurrently.
  - Operations in multiple transactions can be interleaved.
  - Interleaving the operations may cause such anomalies.
  - So we need to have a concurrency control to avoid such anomalies.

Atomicity of Transactions

- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are *atomic.* That is, a user can think of a transaction as
  1. always executing all its operations, or
  2. not executing any operations at all.

  all or none
- What if something wrong during the processing of a transaction ? In other words, what if some failure occurs in the middle of execution transactions.

잘 → commit
잘못 → abort

11/21

Recovery is needed in DBMS

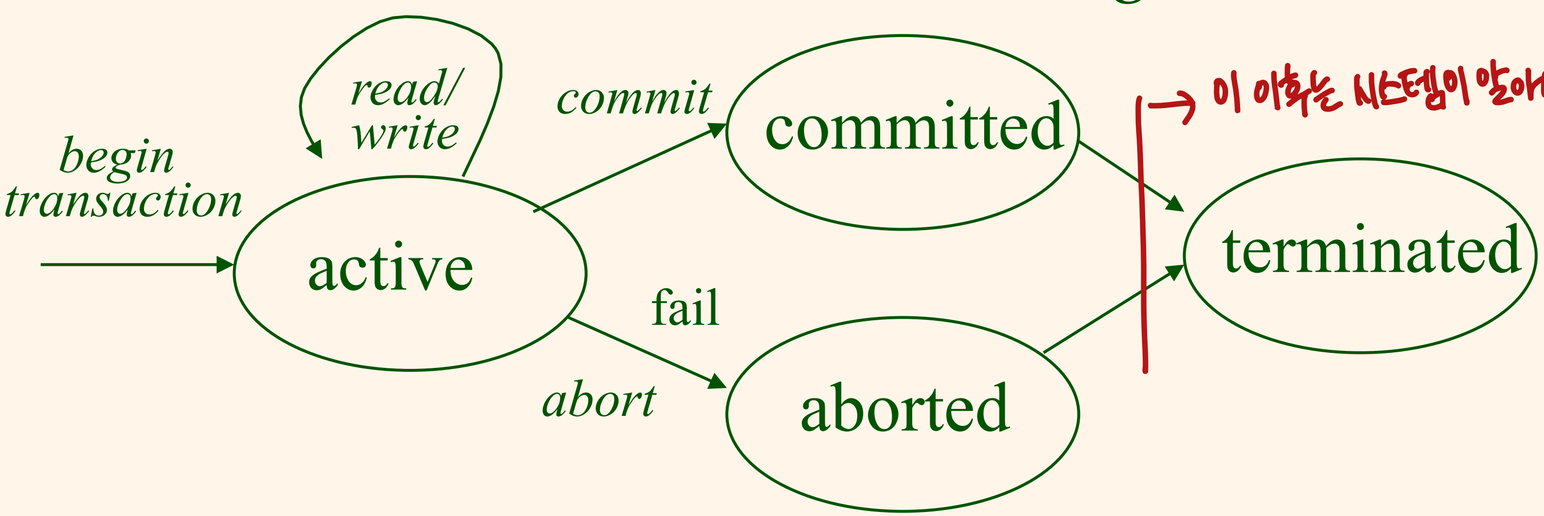
dirty bit
abort : 중단되면 처음으로

- Failures can happen in many forms:
  - Computer failure (system crash) : e.g. *hardware, network failure*
  - Transaction or system error : e.g. *divide by zero*
  - Local errors or exception conditions raised : e.g. *low balance*
  - Concurrency control enforcement *:* e.g. not *serializable*
  - Disk failure
  - Physical problems
- We need a recovery system in DBMS : DBMS *logs* all actions so that it *may undo* the actions of non-committed transactions.

Transaction States and Transition Diagram

- Transaction States
  - Active: a transaction is begun. The transaction can issue *Read* and *Write* operations.
  - Committed: a transaction issued *Commit* operation.
  - Aborted: a transaction failed or issued *Abort* operation. un-do, rollback?
  - Terminated: a transaction is done and leaves the system.
- Transaction States Transition Diagram

→ 이 이후는 시스템이 알아서



객관식?

ACID Properties of Transaction

all or nth

- **Atomicity** : a transaction is an atomic unit of processing.

transition (반의어)

- **Consistency**: a transaction preserves the consistency of the database. +) 수행중에는? → 논외.
- **Isolation** : the execution of a transaction should not be interfered by any other transaction. cf) Isolation level
- **Durability** : changes made by any committed transaction must be persistent in the database.

Schedule of Transactions and Serializability

- Explanation by example.

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.05*A,	B=1.05*B	END

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 5% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted simultaneously. However, the net effect *had better* be equivalent to these two transactions running serially in some order.

Schedule of Transactions and Serializability (Cont'd)

- What can you expect, assuming that you do not have any idea of interleaved executions and that A = B = 100 before you execute those transactions ?

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.05*A,	B=1.05*B	END

Schedule of Transactions and Serializable Schedule (Cont'd)

- You would expect:

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.05*A,	B=1.05*B	END

- If T1 is processed and then T2 is processed, then A = 210, B = 0.

Serial schedule ①

T1:	BEGIN	A=A+100,	B=B-100	END				
T2:					BEGIN	A=1.05*A,	B=1.05*B	END

- If T2 is processed and then T1 is processed, then A = 205, B = 5.

Serial schedule ②

T1:					BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.05*A,	B=1.05*B	END				

Schedule of Transactions and Serializable Schedule (Cont'd)

- Consider a possible interleaving (*schedule A*):

serializable(O)

T1:	A=A+100,		B=B-100	
T2:		A=1.05*A,		B=1.05*B

- This is OK. Why ?
- But what about the following schedule B ?

serializable(X) : 위의 두 serial schedule 중 어떤 것과도 결과가 같지 않음

T1:	A=A+100,		B=B-100
T2:		A=1.05*A, B=1.05*B	

Schedule of Transactions and Serializable Schedule (Cont'd)

serial vs. serializable
↳ 하나가 끝나야 다른 것 시작
interleave으로 실행한 결과가
serial schedule과 같으면
serializable schedule

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.

  serial ↔ concurrent

  - T1 and T2
  - T2 and T1

- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.
  - In the previous slides, Schedule A is serializable schedule while Schedule B is not serializable schedule.

11/28 10분지각 T.T

Conflict Serializable Schedules

- Two operations in a schedule are said to be conflict if:
  - they belong to different transactions → 같은 transaction에 속해 있으면 conflict X
  - they access the same item
  - at least one of the operations is W(X).

Schedule1:			
T1:	R(X),		R(Y) …..commit
T2:		W(X)	commit

Schedule1 (by conventional notation)			
	R1(X), ⟷	W2(X)	R1(Y),C2,C1

순서만 중요
간격은 안중요함

commit

Conflict Serializable Schedules (Cont'd)

R1(X) R1(Y) W2(X)
R1(X) W2(X) R1(Y) → conflict equivalent

conflict 측면에서 equivalent

- Two schedules are conflict equivalent if:
  - Involve the same operations of the same transactions → 순서는 달라도 내용은 같아야함
  - Every pair of conflicting operations is ordered the same way → conflict operation의 순서가 같아야함
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule.

serial schedule과
conflict equivalent하다

Example

<serial execution>
serial하게 진행하는 법 : 2가지
T1→T2 or T2→T1
① ②

serial schedule과 conflict 측면에서 같아야 함

- A schedule that is not conflict serializable:

≠ parallel
interleave (동시수행)

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

T1→T2
T2→T1

옮기면 ① 과 같아짐!
→ conflict serializable

①
T1→T2

T1:	R(A), W(A), R(B), W(B)	
T2:		R(A), W(A), R(B), W(B)

②
T2→T1

T1:		R(A), W(A), R(B), W(B)
T2:	R(A), W(A), R(B), W(B)	

⇒ 순서가 같은게 없음!!
conflict serializable이 될 수 없음

(빨강끼리, 파랑끼리)
3쌍의 operation이 A에 대해 conflict
순서는 T1→T2

(빨강끼리, 파랑끼리)
3쌍의 operation이 B에 대해 conflict
순서는 T2→T1

transaction → node
conflict → edge
operation
→ dependancy graph

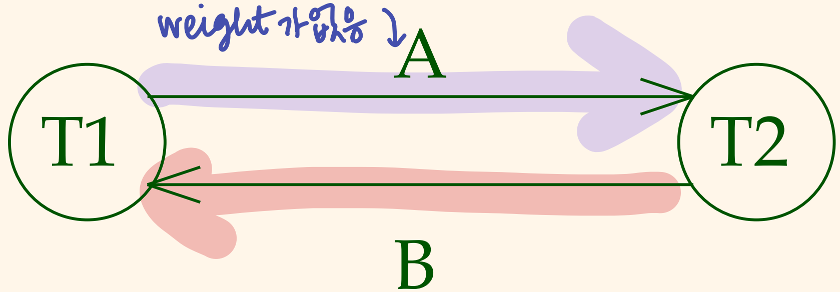
Example (Cont'd)

- A schedule that is not conflict serializable:

node 2개

T1: R(A), W(A), R(B), W(B)
T2: R(A), W(A), R(B), W(B)

횟수가 중요하지 않으므로
weight가 없음



"cycle이 있으면 conflict serializable이 아님!"

Dependency graph

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Dependency Graph

transaction 개수가 많으면
conflict serializable한지 찾기 힘드니까

- *Dependency graph*: One node per transaction; edge from *Ti* to *Tj* if an operation of Ti precedes and conflicts with any operations of Tj.
- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic.

cycle이 없다

필요충분조건

(동치)

conflict serializable하게 유지하는 법

1) (X) 좋지 않음

요청해도 시켜주면 안됨.

2) ⇒ 뒤로 미뤄서 스케줄링!
기다리라고 해야함.

서로필요한 resource
이에따라 매커니즘도 필요

→ access control mechanism

aquire = get

OS 내용 (scheduling)
누구를 줄것인가
(policy)
FIFO?. → queue

Lock-Based Concurrency Control

3가지를 지키자

cf) protocol : 규약. 협약
≒ 법

★ read lock = shared lock
write lock = exclusive lock
못얻으면 기다려라

- *Two-phase Locking (2PL) Protocol*:
  1. – Each transaction must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing. lock을 얻을때까지 기다린다
     ↳ 독점!
  2. – A transaction can not request additional locks once it releases any lock. (growing phase and shrinking phase)
     하나의 lock을 갖고있으면 놓기전까지 또 *
  3. – If a transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object. Or no transaction can not get X lock on an object if other transaction holds a lock (S or X) on that object.

SS → O
SX
XS → X
XX
호환안됨

* R.W 둘다 하지 않을것이라고 DBMS가 아는 시점
: commit 또는 rollback
(transaction이 끝날때)

ex) T1이 A에 대한 X를 가지고있으면 T2는 A에 대한 어떤 lock도 가질수 없음

- 2PL allows only conflict serializable schedules.

11/28

Lock-Based Concurrency Control (Con'd)

transaction 끝날때 끝났다고 직접 말함! → strict 2PL
DBMS가 마지막 오퍼레이션이 뭔지 모르니까 일반 2PL X
→ 꽉 잡고 놓지 않으면 concurrency level ↓
→ 2PL에 비해 strict 2PL이 낮음

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each transaction must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes.
  - If a transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object. Or no transaction can not get X lock on an object if other transaction holds a lock (S or X) on that object.
- Strict 2PL allows only conflict serializable schedules.

프로그램 중단

Aborting a Transaction

- If a transaction *Ti* is aborted, all its actions have to be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
- Most systems avoid such 종속 *cascading aborts* by releasing a transaction's locks only at commit time.
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits.

T1: R(X), X=X-N, W(X)		*..abort.*
T2:	R(X), X=X+M,	*this must be aborted too.*

T1 작업후에 이루어져야함 (종속)
⇒ T1 abort시 같이 undo

cf) deadlock detection / recovery

Aborting a Transaction (Cont'd)

- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active transactions at the time of the crash are aborted when the system comes back up.

T1: R(X), X=X-N, W(X) …. *fails then abort.*

Log: T1 updates X from 100 to 50.

Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency!!!!!!.
  - DBMS does it for you !!!!!
  - For example, DBMS automatically inserts lock/unlock requests and schedules actions of different transactions in such a way as to ensure that the resulting execution is equivalent to executing the transactions one after the other in some order. +recovery, system reboot, … 자동화
- Recovery is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
- Users need not worry about recovery!!!!!!.
  - DBMS does it for you !!!!!
  - For example, DBMS maintains the log.

마지막 수업 날 질의응답

11/30