Chap 13. Transaction Processing Concepts and Theory

DATABASE(CSI3105-01)

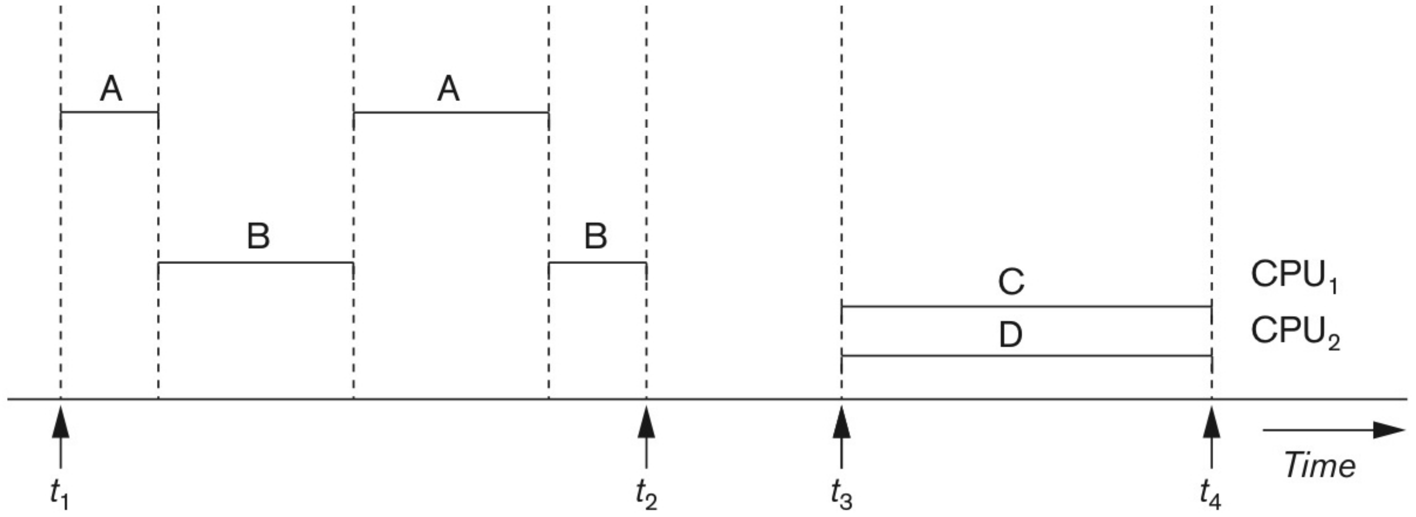
Donghyun Kang

Modified from Wonsuk Lee's Lecture notes

Images from Fundamentals of Database Systems 7^{th}

Single-user vs Multi-user Systems

- Single-user DBMS : at most one user at a time can use DB system
- Multi-user DBMS : many users can use DB system concurrently
- single CPU : multi-programming OS
  -> interleaved execution (**interleaved concurrency**)
- multiple CPU : parallel processing(simultaneous concurrency)



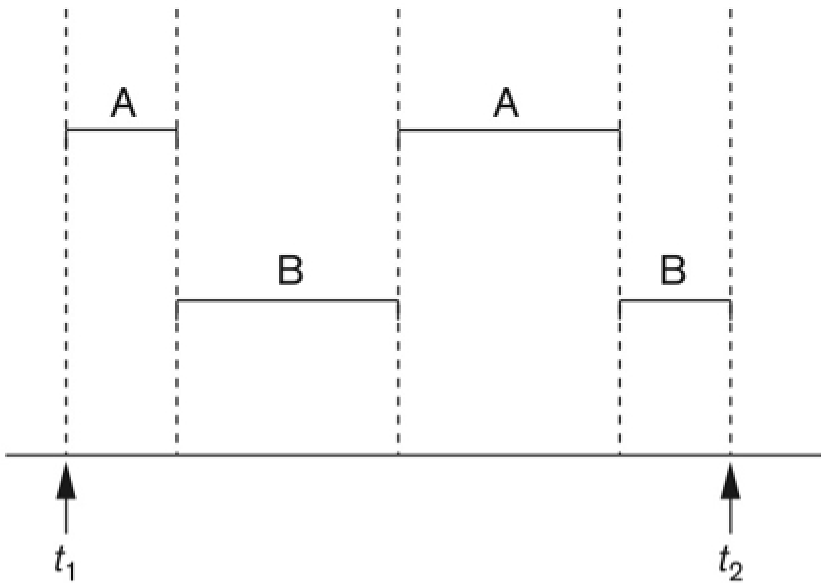
Transactions

- Transaction :
  - execution of a program that forms a logical unit of database processing.
  - one or more atomic read/write operations
    - **read_item(X).**
      Reads a database item named X into a program variable.
      write_item(X).
      Writes the value of program variable into the database item named X

    - **read-only transaction**
    - **read-write transaction**

Transactions

- Two transactions T1 and T2 are requested at the same time
  - correct result of interleaved execution
    => either T1 -> T2 or T2 ->T1



Transactions

- **Sample transaction**

```
1                 EXEC SQL WHENEVER SQLERROR GOTO UNDO:
2                 EXEC SQL SET TRANSACTION
                           READ WRITE
                           DIAGNOSTIC SIZE 5
                           ISOLATION LEVEL SERIALIZABLE;
3                 EXEC SQL INSERT INTO EMPLOYEE (Fname,Lname,Ssn,Dno,Salary)
                           VALUES('robert','smith','991004321',2,35000);
4                 EXEC SQL UPDATE EMPLOYEE
                           SET salary = salary *1.1 WHERE dno=2;
5                 EXEC SQL COMMIT;
6                 GOTO THE_END:
7  UNDO:          EXEC SQL ROLLBACK;
8  THE_END:       ....;
```

Concurrency Control

- Why Concurrency Control?

  [Example] Airline reservation system

  - T1 : cancels N reservations in X flight &
    reserves N seats in Y flight
  - T2 : reserves M seats in X flight

initial value of
X= 80,
N=5,
M=4

(a)

T_1

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

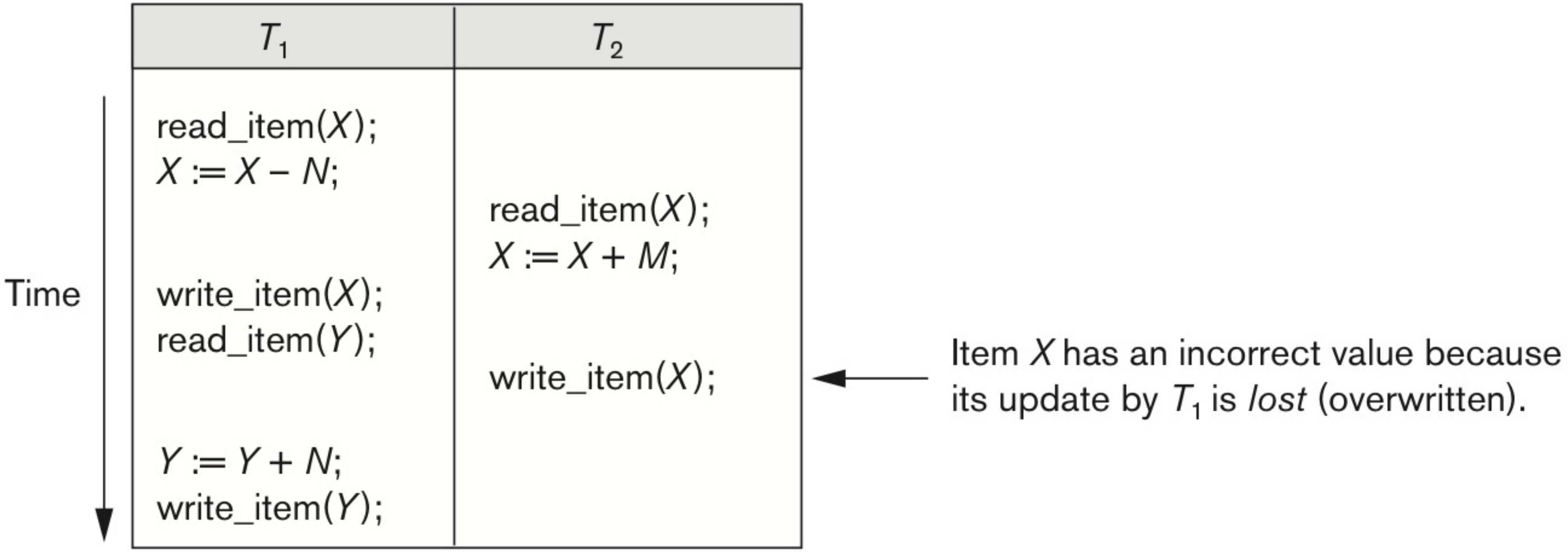
(b)

T_2

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

lost update problem

- two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect



Temporary update problem

- One transcation updates a DB item and fail for some reason
  - The updated item is accessed by another transaction before it is changed back to its original value

Time

T_1	T_2

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

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

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

The Incorrect Summary Problem

- one transaction is calculating an aggregate summary function on a number of database items
  - while other transactions are updating some of these items

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

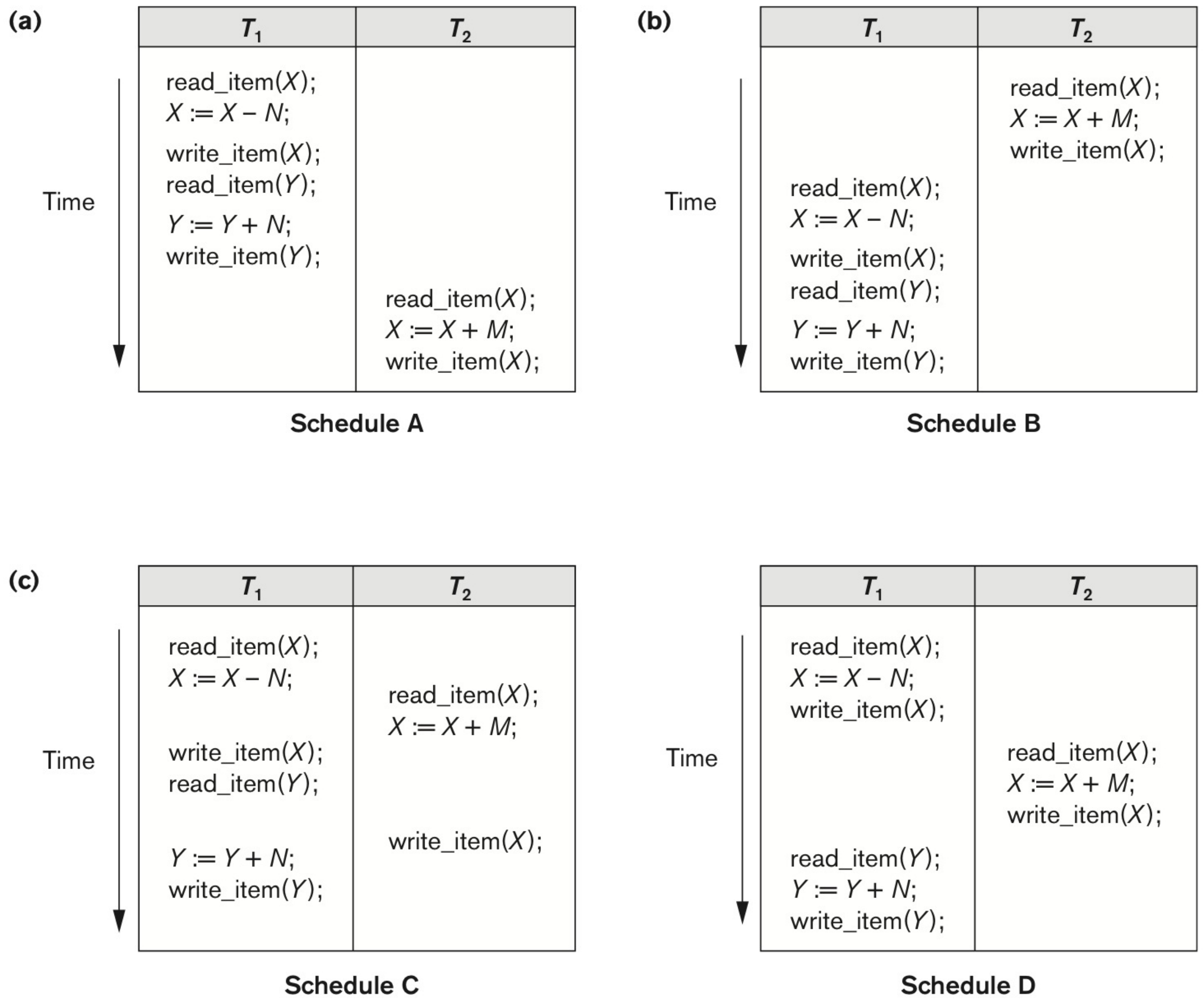
The Unrepeatable Read Problem

- a transaction T1 reads the same item twice
    - the item is changed by another transaction T2 between the two reads.
    - Hence, T1 receives different values for its two reads of the same item.

T1	T2
Read_item(X)	
	X:=X*1.1 Write_item(X)
Read_item(X)	

Schedule & serializable

- A **schedule** S of n transactions is an ordering of the operations of the transaction
- Schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule
- Schedule S is **Serializable** if it is *equivalent* to some serial schedule of the same n transactions.



Schedule & serializable

- Two schedules are Equivalent
  - Result equivalent
    - the same final state of the database.
  - View equivalent
    - as long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results
  - **Conflict equivalent**
    - if the relative order of any two *conflicting operations* is the same in both schedules.
    - two operations in a schedule are *conflict* if they belong to different transactions, access the same database item, andeither both are write_item operations or one is a write_item and the other a read_item

TWO Phase Locking (2PL)

- Two Phase Locking

  multiple-mode locks (read & write locks) + protocols to guarantee serializability

  - Transaction T follows 2PL if all locking operations precede the first unlock operation in T

  (i) expanding phase : acquire new locks
  (no unlock op)

  (ii) shrinking phase : unlock all locks (no lock op)

TWO Phase Locking (2PL)

(a)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

TWO Phase Locking (2PL)

(c)

Time ↓

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);	

Result of schedule S:
X=50, Y=50
(nonserializable)

Transactions that do not obey 2PL.

Interleaved execution of T1 and T2.

TWO Phase Locking (2PL)

T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y)	unlock(X)
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

수정사항

write lock(X)

X = X+X

unlock(X)

Transactions that follow 2PL

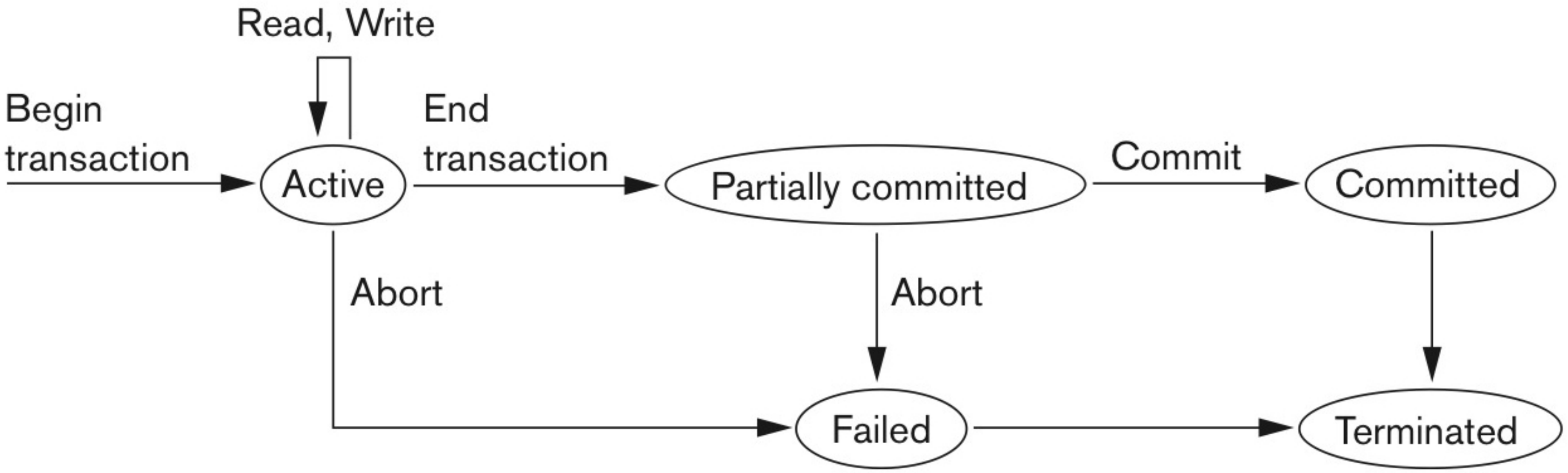
Recovery

- Why Recovery?
  - transaction should be atomic
    - all(**committed**) or nothing(**aborted**)

  - Types of Failures
    - computer failure(**system crash**)
    - transaction or system error
    - Local errors or exception conditions detected by the transaction
    - Concurrency control enforcement.
    - Disk failure.
    - Physical problems and catastrophes

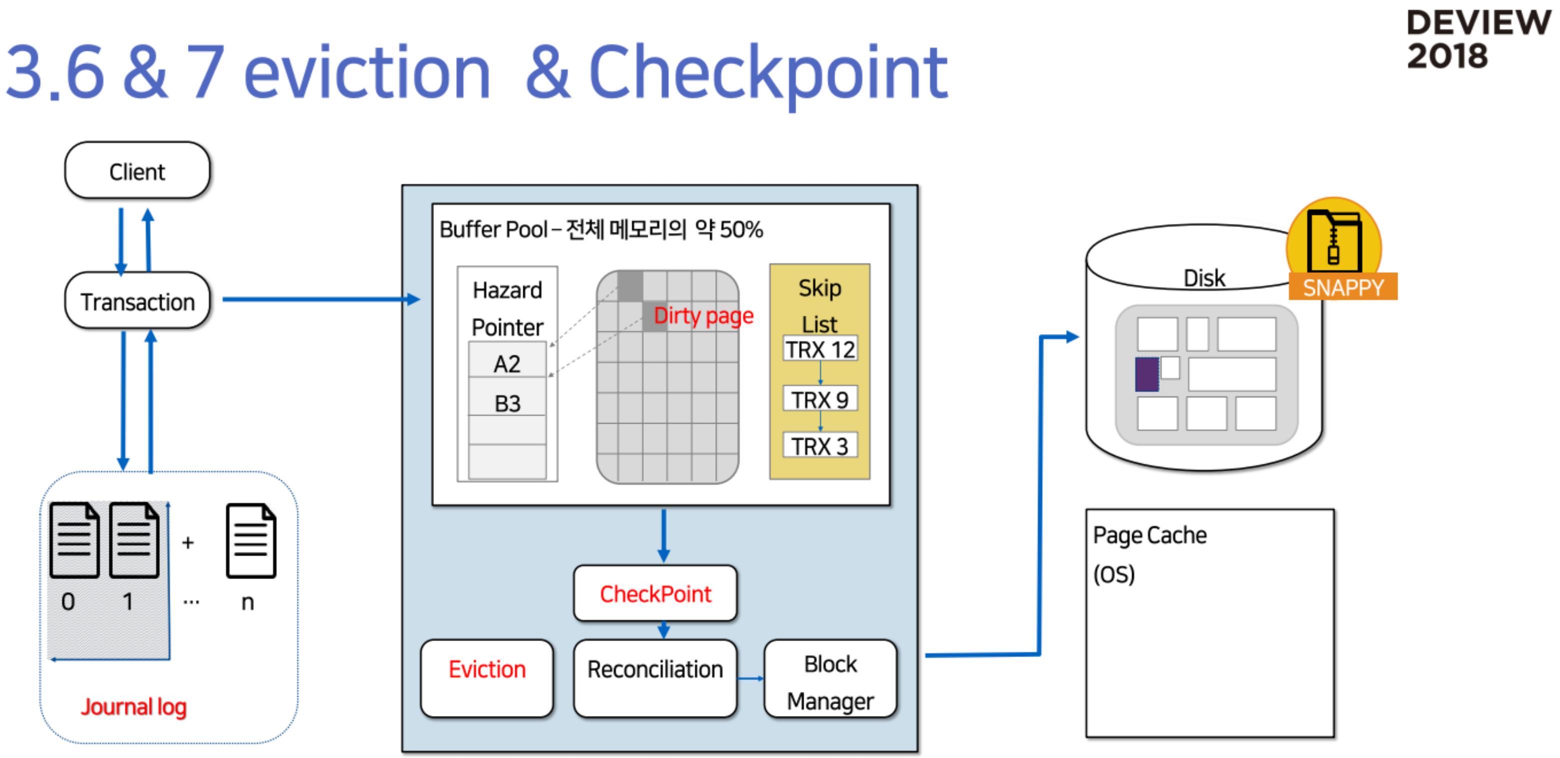
Recovery

- Committed transaction
  - a transaction which is executed successfully.
  - recovery manager only cares for committed transactions
- commit point
  - commit record [commit, transaction T] into the log.



https://tv.naver.com/v/4578359
https://www.slideshare.net/deview/124-data-platform-mongodb

Recovery



- **checkpoint** : [checkpoint, list of active transactions] log
  - the system writes out to the database on disk all DBMS buffers that have been modified.
  - all operations in a transaction have been executed successfully and recorded in the log file [commit, T]

Recovery

- Recovery:
  - system fail => return to consistent DB state

- **if** catastrophic failure,
  - tape dump & log file
  - restructuring old DB state to current state by redoing committed transaction

  else if inconsistent DB state
  - log file
  - undo & redo (-> consistent DB state)

Recovery

(i) deferred update

: after commit point, all updates by a transaction are recorded persistently

=> change the content of DB in disk after commit point

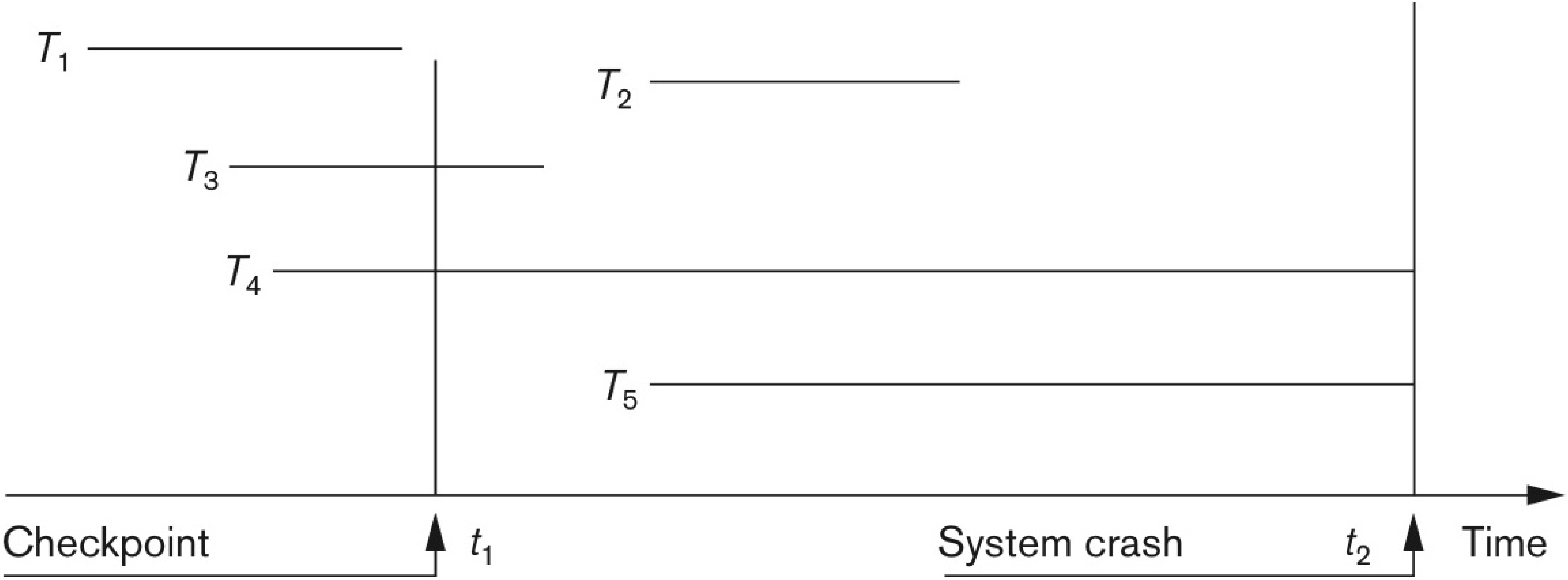
=> No-Undo/Redo algorithm

(ii) immediate update

: update DB in disk before commit point

=> Undo/Redo algorithm

Recovery



Properties of Transactions (ACID)

- Desirable Properties of Transactions

(i) Atomicity : transaction = atomic unit of processing (recovery manager)

(ii) Consistency perservation : (programmer)
consistent DB state -> another consistent DB state

(iii) Isolation : all updates in a transaction should not be visible to other transactions until it is committed (concurrency control)

(iv) Durability (permanency) : (recovery manager)
- Once a transaction is commited (changes DB), the changes should not be lost because of subsequent failure