جامعة بيرزيت
BIRZEIT UNIVERSITY

Transactions Management

Dr. Ahmad Abusnaina
Birzeit University
Department of Computer Sciences

TRANSACTION

- A transaction can be defined as a group of tasks. A single task is the minimum processing work which cannot be divided further.

- Let's take an example of a simple transaction. Suppose a bank employee transfers JOD 500 from A's account to B's account.

- This very simple and small transaction involves several low-level tasks.

Transaction

- **A's Account**

  Open_Account(A)

  Old_Balance = A.balance

  New_Balance = Old_Balance - 500

  A.balance = New_Balance

  Close_Account(A)

- **B's Account**

  Open_Account(B)

  Old_Balance = B.balance

  New_Balance = Old_Balance + 500

  B.balance = New_Balance

  Close_Account(B)

TRANSACTION

- A transaction is the DBMS's abstract view of a user program (or activity):

  - A sequence of reads and writes of database objects.
  - Unit of work that must commit or abort as an atomic unit.
  - A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
  - Transaction Manager controls the execution of transactions.

ACID Properties

- A transaction is a very small unit of a program and it may contain several low-level tasks.
- A transaction in a database system must maintain four properties:
  - **A**tomicity,
  - **C**onsistency,
  - **I**solation,
  - **D**urability
- Commonly known as **ACID** properties.

ATOMIC

- The execution of each transaction has to be **atomic**
- Either all actions are carried out (happen) or none happen.
- There must be no state in a database where a transaction is left partially completed.
- The user should not worry about the effect of incomplete transactions.

CONSISTENCY

- The database must remain in a consistent state after any transaction.
- No transaction should have any adverse effect on the data residing in the database.
- If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

ISOLATION

- In a database system where more than one transaction are being executed simultaneously and in parallel,
- Every transaction is an independent entity. One transaction should not affect any other transaction running at the same time.
- all the transactions will be carried out and executed as if it is the only transaction in the system.

DURABILITY

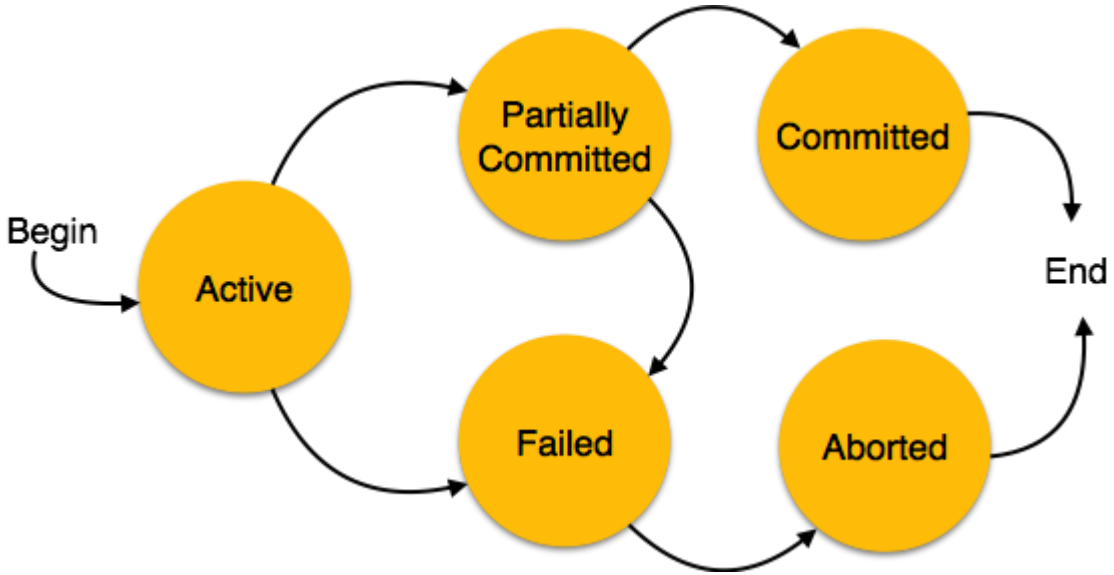
- Once the DBMS informs the user that the transaction has been successfully completed:
  - Its effects should be permanent even if the system crashes before changes are reflected to disk.
- If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

- A transaction ends in one of three cases:
  - Commit after completing all actions
  - Abort after executing some actions
  - System crash while the transaction is in progress.

- DBMS ensures the above states by logging all actions:
  - Undo the actions of aborted/failed transactions
  - Redo actions of committed transactions not yet propagated to disk when system crashes.

- The DBMS must find a way to clean up partial transactions
- The DBMS uses **Log**
- Keeping in it all the changes made to the database
- It is also used for durability

States of Transaction



TRANSACTIONS AND SCHEDULES

- A transaction is seen by DBMS as a series of **actions**…..**read** and **write**

- **R(O)** : transaction reading an object from DB

- **W(O)** : transaction writing an object to DB
- **Abort** : action of a transaction aborting
- **Commit** : action of transaction committing

- **A schedule:** is a list of actions of reading, writing, aborting or committing from a set of transactions with the same order as when the transactions are in the origin transaction

- Vertical axis's is the time

- A **complete schedule** include all actions of all transactions appearing in it

- **Serial schedule** no interleaving (no concurrent execution) of actions from different transactions.
- Refer to figure 16.1

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

Figure A Schedule Involving Two Transactions

Concurrent execution of transactions

- Concurrent Execution: The DBMS interleaves the actions of different transactions to improve the performance

- But not all interleaves should be allowed, why?

MOTIVATION FOR CONCURRENT EXECUTION

- First: while one transaction is waiting for page reading from disk, the CPU can process another transaction . **Fast**
- Second: interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly.
  - In serial execution, the short transaction will have to wait. Short could stuck behind long transaction.
- **Concurrent Execution**
  - **increase System throughput:** number of transactions completed in a given amount of time.
  - **decrease Response Time:** average time taken to complete a transaction.

SERIALIZABILITY AND SCHEDULES

- Given a set of transactions, a schedule is a sequence of interleaved actions from all transactions

- Example: Given the following transactions, how to schedule them?
- T1: Read(A), Write(A), Read(B), Write(B)
- T2: Read(A), Write(A), Read(B), Write(B)

SERIAL SCHEDULE

- A possible serial schedule is:

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

SERIALIZABILITY

- A schedule is **serializable** if it is equivalent to complete serial schedule.
- **A serializable schedule:** over a set of a committed transaction has the effect of the database to be the same as some other complete serial schedule.
- Informally, equivalent, means that all conflicting operations are ordered in the same way
- conflicting those read and write operations to the same element,

SERIALIZABILITY

Serializable

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

Non-Serializable

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

Anomalies due to interleaved execution

- Two actions on the same data object may **conflict** if at least one of them is a **write**
  - WR (dirty read)
  - RW (unrepeatable read)
  - WW (overwriting uncommitted data)
  - Unrecoverable schedule

Reading uncommitted Data
WR Conflict

- A transaction T2 could read a database object A that has been modified by T1, but not yet committed

- We call it ***Dirty Read***

T1	T2
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

Figure 18.2 Reading Uncommitted Da

UNREPEATABLE READS
RW CONFLICT

- T2 change the value of an object A that has been read by T1 **while T1 is still in progress**

- If T1 tries to read A another time it will be different!

T1	T2
R(A)	
	R(A)
	W(A)
	Commit
R(A)	
Commit	

Overwriting Uncommitted Data

WW Conflict

- T2 could overwrites object A which has already been modified by T1, while T1 is still in progress.

T1	T2
W(A)	
	W(A)
	W(B)
	Commit
W(B)	
Commit	

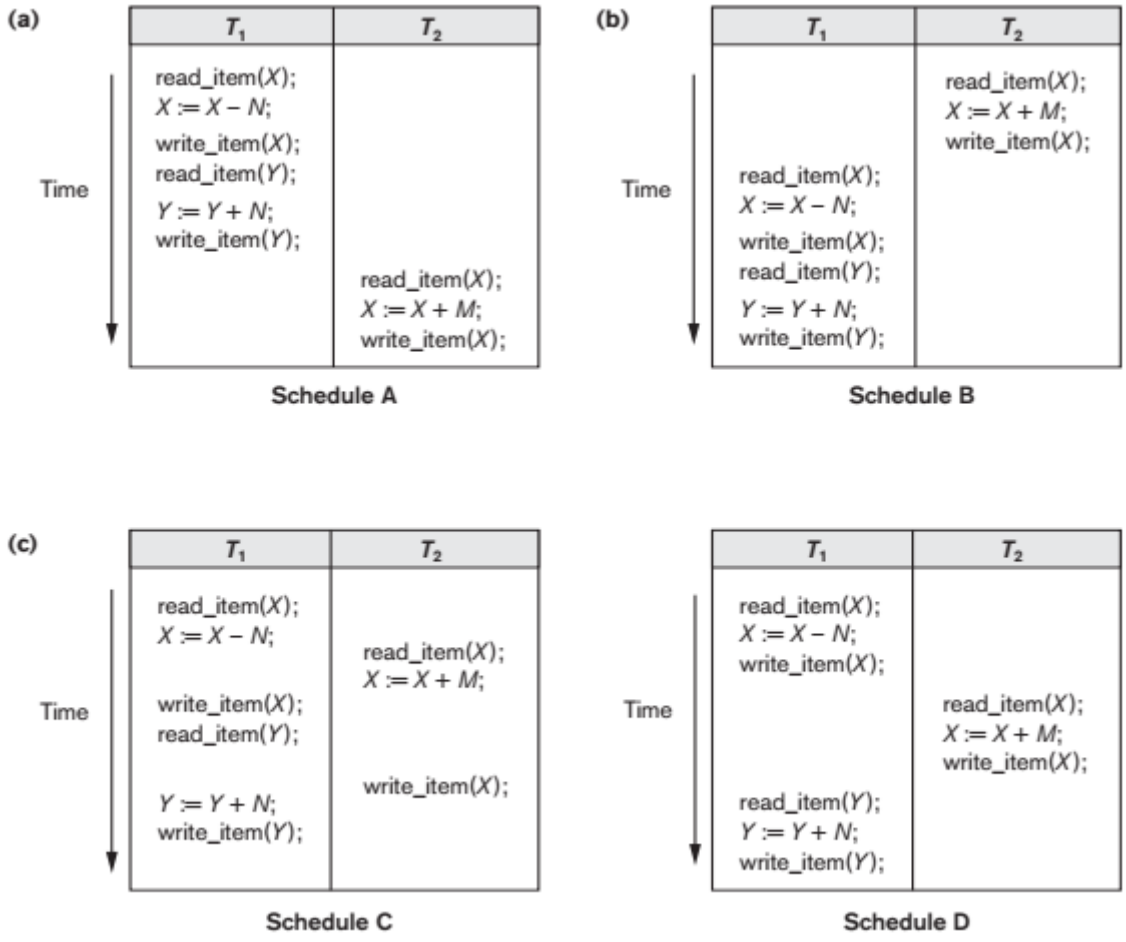
UNRECOVERABLE SCHEDULE

- In case of aborted transaction, the system make rollback (undo), so the effect of T2 will be lost.

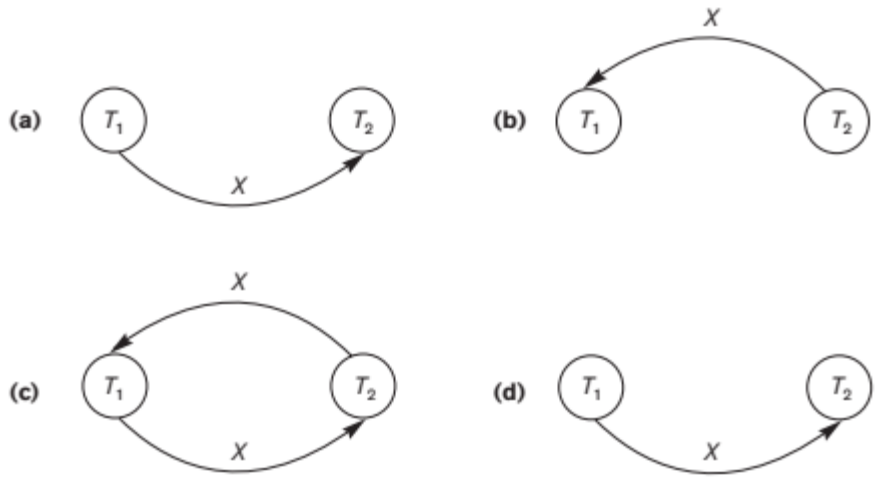
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	Commit
R(L)	
Abort	

TESTING SERIALIZABILITY

Which schedule is serial, non serial, serializable?



CONSTRUCTING THE PRECEDENCE GRAPHS FOR SCHEDULES



Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability.

(a) Precedence graph for serial schedule A.
(b) Precedence graph for serial schedule B.
(c) Precedence graph for schedule C (not serializable).
(d) Precedence graph for schedule D (serializable, equivalent to schedule A).

(a)

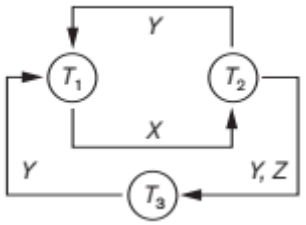
Transaction T_1	Transaction T_2	Transaction T_3
read_item(X);	read_item(Z);	read_item(Y);
write_item(X);	read_item(Y);	read_item(Z);
read_item(Y);	write_item(Y);	write_item(Y);
write_item(Y);	read_item(X);	write_item(Z);
	write_item(X);	

(b)

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

Time

Schedule E



Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

(a)

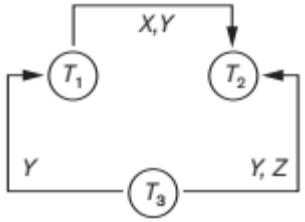
Transaction T_1	Transaction T_2	Transaction T_3
read_item(X);	read_item(Z);	read_item(Y);
write_item(X);	read_item(Y);	read_item(Z);
read_item(Y);	write_item(Y);	write_item(Y);
write_item(Y);	read_item(X);	write_item(Z);
	write_item(X);	

(c)

Time

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



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

LOCK-BASED CONCURRENCY CONTROL

- A locking protocol is a set of rules to be followed by each transaction to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in serial order.
- To avoid previous conflicts

Strict Two-phase Locking (Strict 2PL) Protocol

- Each transaction must obtain an S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
- Lock rules:
  - If a Tx holds an X lock on an object, no other Tx can acquire a lock (S or X) on that object;
  - If a Tx holds an S lock, no other Tx can get an X lock on that object.

CONCURRENT

T1	T2
R(A)	
	R(A)
	W(B)
	Commit
R(C)	
W(C)	
Commit	

AFTER APPLYING STRICT 2PL

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
	X(B)
	W(B)
	Commit
S(C)	
R(C)	
X(C)	
W(C)	
Commit	

SCHEDULING WITH 2PL

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

Figure 18.2 Reading Uncommitted Data

- **T1** **T2**
- S(A)
- R(A)
- X(A)
- W(A)
- S(B)
- R(B)
- X(B)
- W(B)
- COMMIT
- S(A)
- R(A)
- X(A)
- W(A)
- S(B)
- R(B)
- X(B)
- W(B)
- Commit

SCHEDULING WITH 2PL (EXAMPLE2)

T1	T2
R(A)	
	R(C)
	W(C)
R(B)	
W(B)	
	W(B)
	COMMIT
COMMIT	

T1	T2
S(A)	
R(A)	
	X(C)
	R(C)
	W(C)
X(B)	
R(B)	
W(B)	
COMMIT	
	X(B)
	W(B)
	COMMIT

Dead Lock

- In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.
- For example, assume a set of transactions {T_0, T_1, T_2, ...,T_n}.
  - T_0 needs an object X to complete its task.
  - object X is held by T_1, and T_1 is waiting for a object Y,
  - Y is held by T_2. T_2 is waiting for object Z,
  - Z is held by T_0.
- Thus, all the processes wait for each other to release resources.
- In this situation, none of the processes can finish their task.
- This situation is known as a deadlock.

Dead Lock

- Deadlocks are not healthy for a system.
- In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.
- Deadlock Detection: many, one of them is timeout mechanism.

- Deadlock Prevention: If DBMS finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

Dead Lock

T1	T2
X(A)	
R(A)	
W(A)	
	X(C)
	R(C)
	W(C)
X(C)	
R(C)	
W(C)	
COMMIT	
	X(A)
	W(A)
	COMMIT