# TRANSACTIONS MANAGEMENT

Credits to Dr. Ahmad Abusnaina

# 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 J O D 500 from A's account to B's account.

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

*Transactions Management*

# TRANSACTION

o **A's Account**

Open_Account(A)

Old_Balance = A.balance

New_Balance = Old_Balance - 500

A.balance = New_Balance

Close_Account(A)

o **B's Account**

Open_Account(B)

Old_Balance = B.balance

New_Balance = Old_Balance + 500

B.balance = New_Balance

Close_Account(B)

*Transactions Management*

3

# 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.

**4**

# 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.

5

# 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.

*Transactions Management*

Uploaded By: Jibreel Bornat

6

# 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.

*Transactions Management*

# 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.

*Transactions Management*

Uploaded By: Jibreel Bornat

# 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.

*Transactions Management*

Uploaded By: Jibreel Bornat

○ 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.

*Transactions Management*

Uploaded By: Jibreel Bornat

- 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.

11

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

- 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

*Transactions Management*

12

# STATES OF TRANSACTION

*Transactions Management*

13
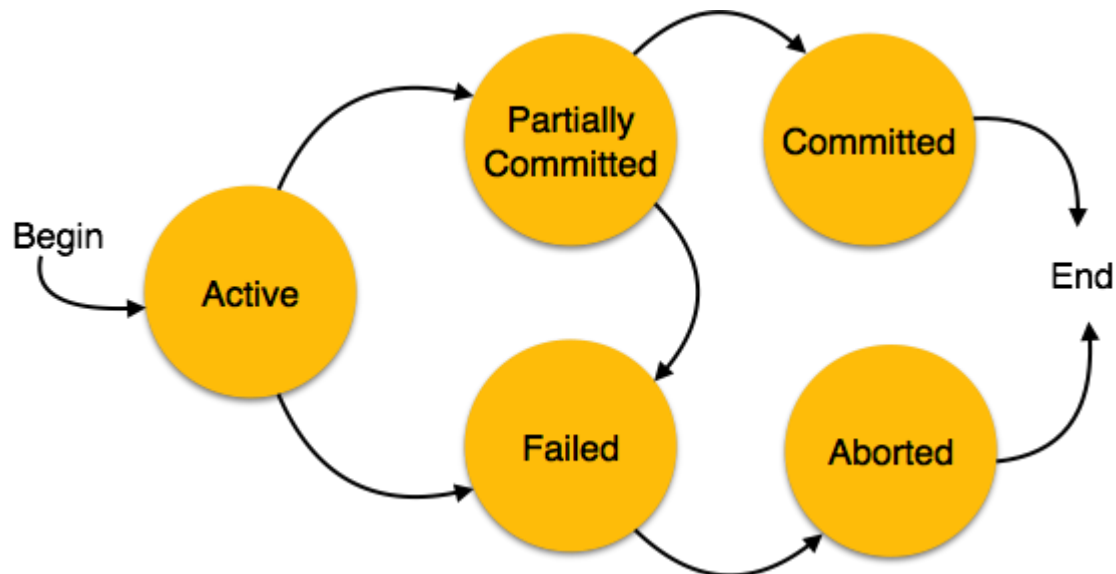
# TRANSACTIONS AND SCHEDULES

- 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

14

- **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

*Transactions Management*

- 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

*Transactions Management*

16

| 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?

18

*Transactions Management*

# 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.

*Transactions Management*

# 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)

*Transactions Management*  Uploaded By: Jibreel Bornat

# 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)

*Transactions Management*

21

# Serializability

- A schedule is **serializable** if it is equivalent to <span style="color:red">complete serial</span> 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 <u>conflicting</u> operations are ordered in the same way

- <u>conflicting</u> those read and write operations to the same element,

22

# SERIALIZABILITY

| Serializable | Non-Serializable |
|---|---|

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

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

23

*Transactions Management*

Uploaded By: Jibreel Bornat

# 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

24

# 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

# 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!
- It is called **Unrepeatable read**

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

26

*Transactions Management*

# Overwriting Uncommitted Data WW Conflict

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

| **T1** | **T2** |
|--------|--------|
| W(A) | |
| | W(A) |
| | W(B) |
| | Commit |
| W(B) | |
| Commit | |

*Transactions Management*

Uploaded By: Jibreel Bornat

- 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 | |

*Transactions Management*

28

**Which schedule is serial, non serial, serializable?**

(a)

| $T_1$ | $T_2$ |
|---|---|
| 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); |

Time

Schedule A

(b)

| $T_1$ | $T_2$ |
|---|---|
| | read_item(X); |
| | $X := X + M$; |
| | write_item(X); |
| read_item(X); | |
| $X := X - N$; | |
| write_item(X); | |
| read_item(Y); | |
| $Y := Y + N$; | |
| write_item(Y); | |

Time

Schedule B

(c)

| $T_1$ | $T_2$ |
|---|---|
| read_item(X); | |
| $X := X - N$; | |
| | read_item(X); |
| | $X := X + M$; |
| write_item(X); | |
| read_item(Y); | |
| | write_item(X); |
| $Y := Y + N$; | |
| write_item(Y); | |

Time

Schedule C

| $T_1$ | $T_2$ |
|---|---|
| read_item(X); | |
| $X := X - N$; | |
| write_item(X); | |
| | read_item(X); |
| | $X := X + M$; |
| | write_item(X); |
| read_item(Y); | |
| $Y := Y + N$; | |
| write_item(Y); | |

Time

Schedule D

29
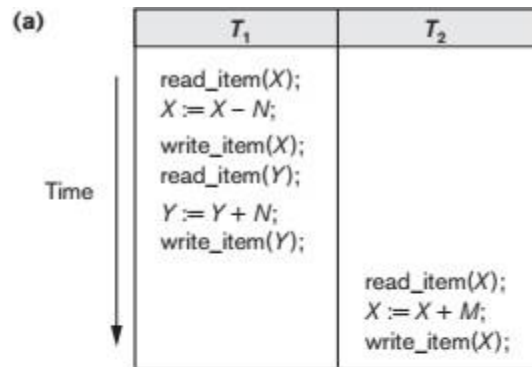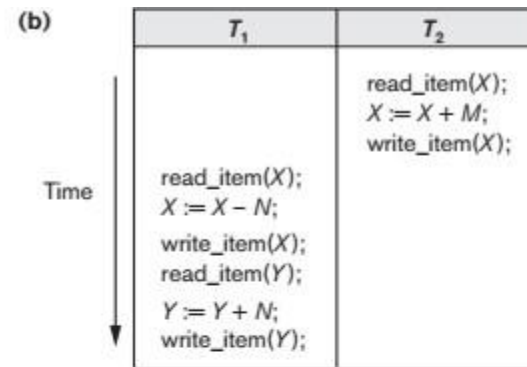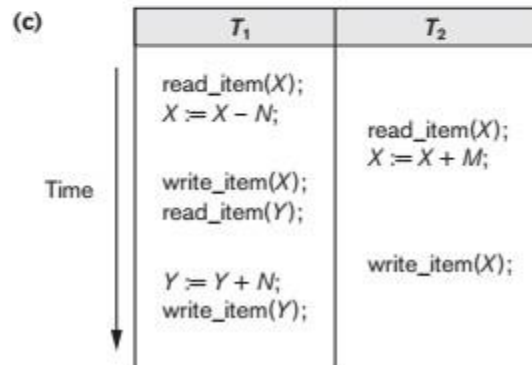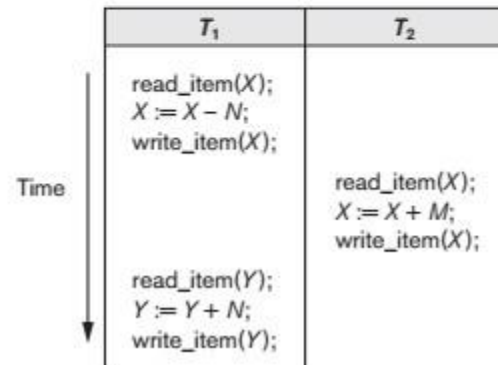
Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability.

(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).

*Transactions Management*

(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 $T_3$ |
|---|---|---|---|
| Time | | read_item(Z);<br>read_item(Y);<br>write_item(Y); | |
| | | | read_item(Y);<br>read_item(Z); |
| | read_item(X);<br>write_item(X); | | |
| | | | write_item(Y);<br>write_item(Z); |
| | | read_item(X); | |
| | read_item(Y);<br>write_item(Y); | write_item(X); | |

Schedule E
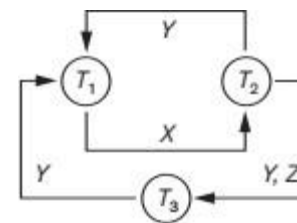


Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

31

(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($Z$); |
| 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



Equivalent serial schedules

$T_3 \to T_1 \to T_2$

STUDENTS-HUB.com

*Transactions Management*

Uploaded By: Jibreel Bornat

# 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

*Transactions Management*

Uploaded By: Jibreel Bornat

# 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.

34

*Transactions Management*

## 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 | |

*Transactions Management*

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 |

## Left schedule

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

## Right schedule

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

*Transactions Management*

Uploaded By: Jibreel Bornat

# 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.

38

*Transactions Management*

# 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.

39

# 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

40

# Wait-for-graph deadlock detection
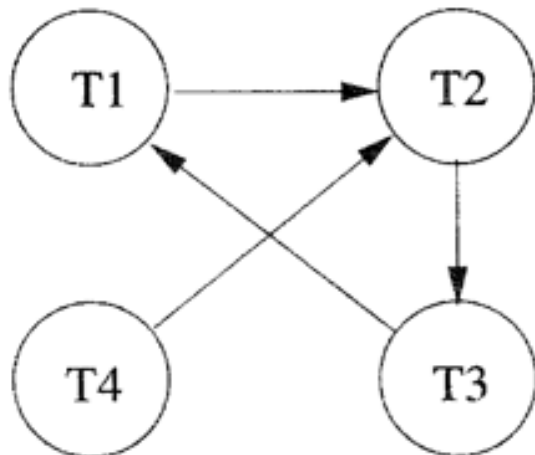
Key Elements of a Wait-for-Graph:

*   Nodes: Each node represents a transaction currently active in the database.

*   Directed Edges: An edge from node Ti to node Tj indicates that transaction Ti is waiting for a lock currently held by transaction Tj.

*   Cycles: The presence of a cycle in the graph signifies a deadlock, as each transaction in the cycle is indefinitely waiting for another transaction to release a lock.
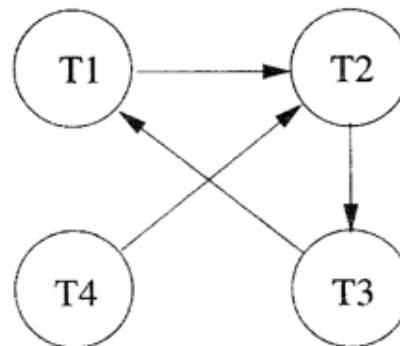
# Construction of the Wait-for-Graph

1. Create a node for each active transaction in the system.
   - This will represent transactions like T1, T2, etc.
2. Draw a directed edge from transaction Ti to transaction Tj if:
   - Transaction Ti has requested a data lock
   - The requested data is currently locked by Tj
   - Ti has to wait for Tj to release the lock
3. Repeat step 2 as lock requests come in from transactions over time. This incrementally builds the graph.
4. Whenever a new lock request adds an edge from Ti to Tj:
   - Check if graph now has a cycle from Ti → ... → Tj → Ti
   - If yes, a deadlock is detected
5. To check for cycles, use a cycle detection algorithm like depth first search on directed graph.

| T1 | T2 | T3 | T4 |
|------|------|------|------|
| S(A) |      |      |      |
| R(A) |      |      |      |
|      | X(B) |      |      |
|      | W(B) |      |      |
| S(B) |      |      |      |
|      |      | S(C) |      |
|      |      | R(C) |      |
|      | X(C) |      |      |
|      |      |      | X(B) |
|      |      | X(A) |      |

Uploaded By: Jibreel Bornat

# Deadlock Handling

- **Aborting Transactions**:
  - The DBMS typically selects one or more transactions from the cycle to be aborted (rolled back).

- **Victim Selection**:
  - The choice of which transaction(s) to abort might be based on factors like priority, age, or number of resources held.

- **Releasing Locks**:
  - Once a transaction is aborted, its locks are released, allowing other transactions to proceed.

Uploaded By: Jibreel Bornat

# Thrashing

Thrashing occurs when a database system becomes overloaded with too many transactions attempting to access and modify data simultaneously. This excessive contention for resources leads to significant performance degradation.

**Thrashing Threshold**: When around 30% of active transactions are blocked, thrashing typically begins.

**Throughput Decline**: Beyond this threshold, adding more transactions counterintuitively reduces overall throughput, as the system becomes overwhelmed with managing conflicts and resource contention.