
# Transactions

# Lecture will cover following points

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Transaction Definition in SQL
- Testing for Serializability.

# What is a Transaction?

- A logical unit of work on a database
  - An entire program
  - A portion of a program
  - A single command
- The entire series of steps necessary to accomplish a logical unit of work
- Successful transactions change the database from one CONSISTENT STATE to another

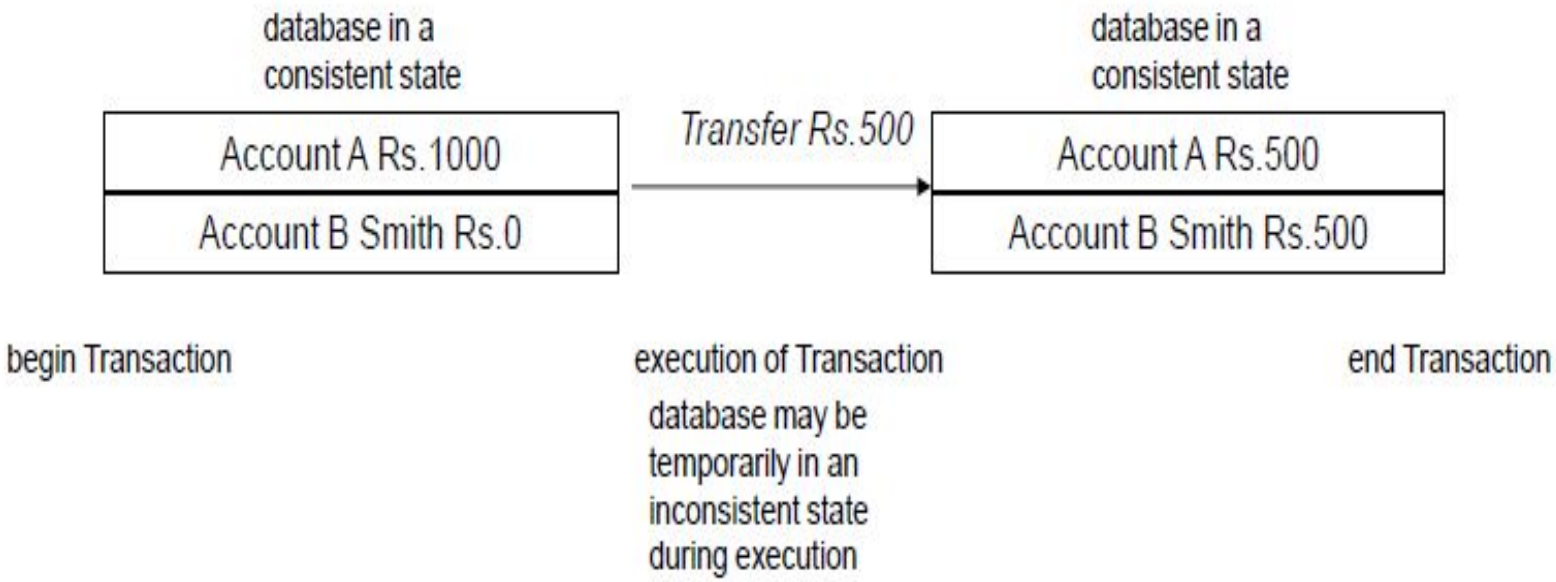
(One where all data integrity constraints are satisfied)

- All database access operations between **Begin Transaction and End Transaction statements are considered one logical transaction.**
- If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction.**
- **Basic database access operations:**
  - **read_item(X): reads a database item X into program variable.**
  - **Write_item(X): Writes the value of program variable X into the database item X.**

# Example of a Transaction

- Updating a Record
  - Locate the Record on Disk
  - Bring record into Buffer
  - Update Data in the Buffer
  - Writing Data Back to Disk

# What is a transaction



# ACID Properties

- To preserve integrity of data, the database system must ensure:
  - **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
  - **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
  - **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
    - That is, for every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that either $T_j$ finished execution before $T_i$ started, or $T_j$ started execution after $T_i$ finished.
  - **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

# Example of Fund Transfer

- Transaction to transfer $50 from account *A* to account *B*:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- Consistency requirement – the sum of *A* and *B* is unchanged by the execution of the transaction.
- Atomicity requirement — if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

  Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database

# Example of Fund Transfer (Cont.)

- Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.

# Example of Fund Transfer (Cont.)

- **Isolation requirement—if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum *A + B will be less than it should be).***

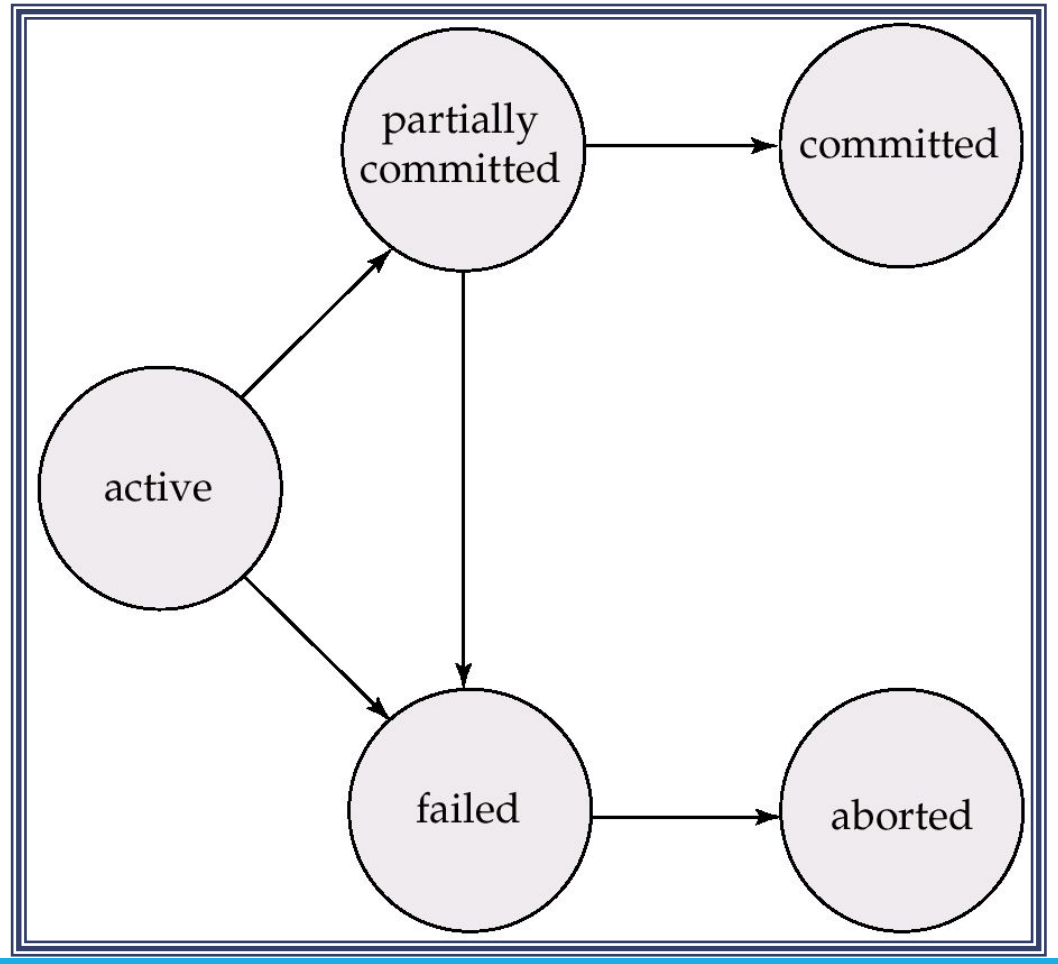
| ***T1*** | ***T2*** |
|---|---|
| 1.**read(*A*)** | |
| 2.*A:= A –50* | |
| 3.**write(*A*)** | |
| | ***read(A), read(B), print(A+B)*** |
| 4.**read(*B*)** | |
| 5.*B:= B + 50* | |
| 6.**write(*B*** | |

  - Isolation can be ensured trivially by running transactions **serially**
    –that is, one after the other.
  - However, executing multiple transactions concurrently has significant benefits, as we will see later.

# Transaction State

- **Active,** the initial state; the transaction stays in this state while it is executing
- **Partially committed,** after the final statement has been executed.
- **Failed,** after the discovery that normal execution can no longer proceed.
- **Aborted,** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction – only if no internal logical error
  - kill the transaction
- **Committed,** after *successful completion*.

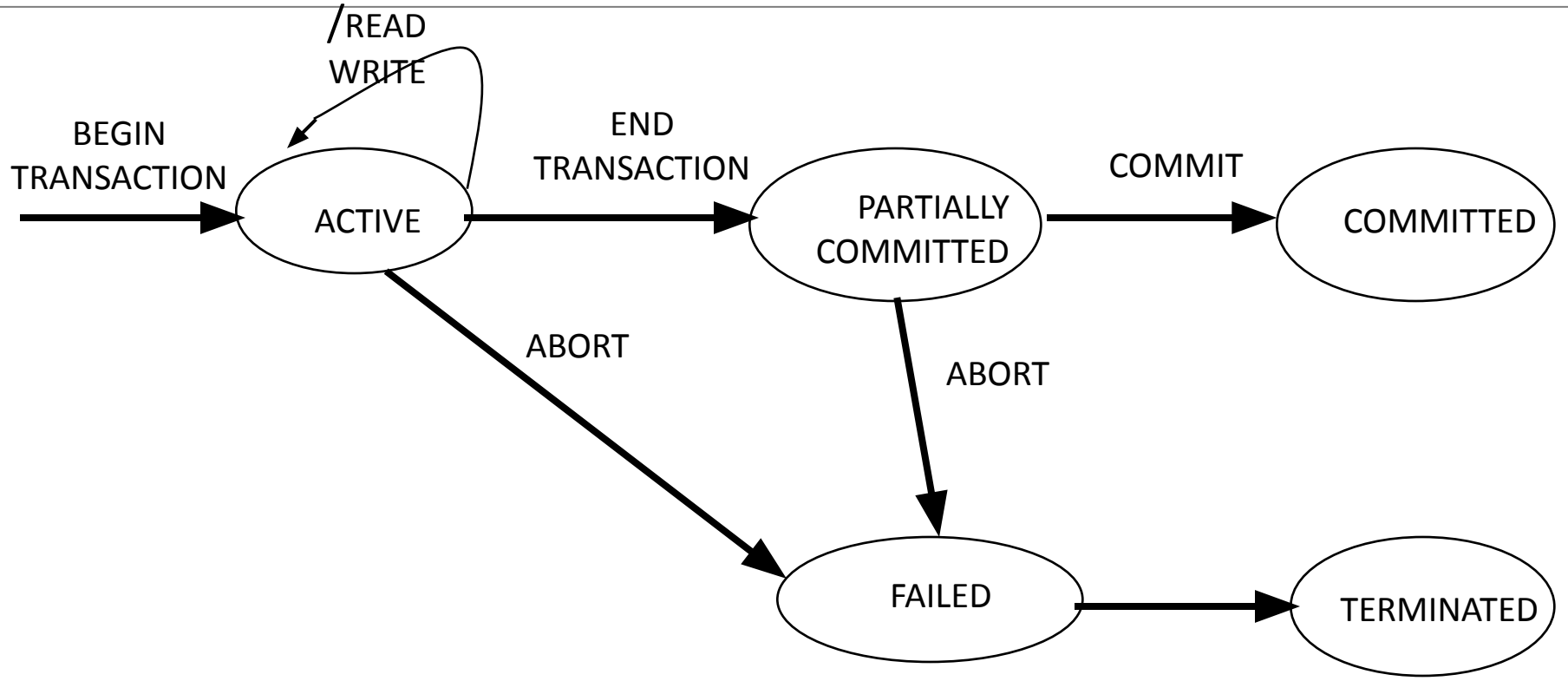
# Transaction State (Cont.)



# The System Log

- The system maintains a log to keep track of all transaction operations that affect the values of database items.
- This log may be needed to recover from failures.
- Types of log records :
  - **[start_transaction,T] :** indicates that transaction T has started execution.
  - **[write_item,T,X,old_value,new_value] :** indicates that transaction T has changed the value of database item X from old_value to new_value. (new_value may not be recorded)
  - **[read_item,T,X]:** indicates that transaction T has read the value of database item X.
  - (read_item may not be recorded)
  - **[commit,T]:** transaction T has recorded permanently .
  - **[abort,T]:** indicates that transaction T has been aborted.

# Transaction states and additional operations




State transition diagram illustrating the states for transaction execution

# Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.
- Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- *Concurrency control schemes* – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

# Schedules

- *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

# Example Schedules

Let $T_1$ transfer \$50 from $A$ to $B$, and $T_2$ transfer 10% of the balance from $A$ to $B$. The following is a serial schedule (Schedule 1 in the text of the Korth book), in which $T_1$ is followed by $T_2$.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| write ($A$) | |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | read($A$) |
| | $temp := A * 0.1$ |
| | $A := A - temp$ |
| | write($A$) |
| | read($B$) |
| | $B := B + temp$ |
| | write($B$) |

# Example Schedule (Cont.)

Let $T_1$ and $T_2$ be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to the previous Schedule.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| write($A$) | |
| | read($A$) |
| | $temp := A * 0.1$ |
| | $A := A - temp$ |
| | write($A$) |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | read($B$) |
| | $B := B + temp$ |
| | write($B$) |

In both Schedule the sum A + B is preserved.

# Example Schedules (Cont.)

The following concurrent schedule does not preserve the value of the sum *A* + *B*.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| | read($A$) |
| | $temp := A * 0.1$ |
| | $A := A - temp$ |
| | write($A$) |
| | read($B$) |
| write($A$) | |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | $B := B + temp$ |
| | write($B$) |

# Serializability

- Basic Assumption – Each transaction preserves database consistency.

  Thus serial execution of a set of transactions preserves database consistency.

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. conflict serializability
  2. view serializability

- We ignore operations other than **read** and **write** instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions.

# Conflict Serializability

- Instructions $I_i$ and $I_j$ of transactions $T_i$ and $T_j$ respectively, **conflict** if and only if there exists some item $Q$ accessed by both $I_i$ and $I_j$, and at least one of these instructions wrote $Q$.
  1. $I_i$ = **read**($Q$), $I_j$ = **read**($Q$). $I_i$ and $I_j$ don't conflict.
  2. $I_i$ = **read**($Q$), $I_j$ = **write**($Q$). They conflict.
  3. $I_i$ = **write**($Q$), $I_j$ = **read**($Q$). They conflict
  4. $I_i$ = **write**($Q$), $I_j$ = **write**($Q$). They conflict
- Intuitively, a conflict between $I_i$ and $I_j$ forces a (logical) temporal order between them. If $I_i$ and $I_j$ are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

# CONCURRENT EXECUTIONS – CONFLICTING ACTIONS

- What are the Anomalies Associated with Interleaved Execution?
  Let T1 and T2 be two *consistency preserving transactions.*

- Now, if the actions of T1 and T2 are interleaved, then two action on the same data object conflict , if at least one of them is a write.

  Possible conflicts are:
  **1. WRITE-READ (WR) CONFLICT**
  **2. READ-WRITE (RW) CONFLICT**
  **3. WRITE-WRITE (WW) CONFLICT**

- WR CONFLICT [READING UNCOMMITTED DATA ]
  When a transaction T2 could read a database object that has been modified by another transaction T1, which has not yet committed. Such a read is called dirty read.

- RW CONFLICTS [UNREPEATABLE READS]
  When a transaction T2 changes the value of an object A that has been already read by a transaction T1, while T1 is still in progress. Problem:

  If T1 tries to read the value of A again, it will get a different result, even though it has not modified A in the meantime. It is called an unrepeatable read.

- WW CONFLICTS [OVERWRITING UNCOMMITTED DATA]
  When a transaction T2 could overwrite the value of an object A, which has already been modified by a transaction T1, while T1 is still in progress. Even if T2 does not read the value of A written by T1, a potential problem exists .

# CONCURRENT EXECUTIONS – CONFLICTING ACTIONS – DIRTY READ

- **Example:** Consider two transactions *T1 and T2,* each of which, run alone, preserves database consistency :
  Consider fund transfer from account A to B using following serial execution of T1 and T2:
  (The task is to transfer $100 from A to B and then increment the funds in A and B by 6%)
  1. T1 transfers $100 from A to B
  2. T2 increments both A and B by 6 percent.
  The database is consistent because the above schedule is serial.
  **[Dirty Read] :**
  Now, Suppose that their actions are interleaved as below:
  1. T1 deducts $100 from account A,
  2. T2 reads the current values of accounts A and B and adds 6 percent interest
  3. The account transfer program credits $100 to account B
  The above schedule will not leave the database in a consistent state.

# CONCURRENT EXECUTIONS – CONFLICTING ACTIONS – EXAMPLE

WW CONFLICTS (OVERWRITING UNCOMMITTED DATA)
Suppose that Harry and Larry are two employees.
**Consistent criterion :** Their salaries must be kept equal.
Consider the following action of T1 and T2 each of which is consistency preserving:
**Actions of T1 :**
1. set Harry's salary to $1000.
2. set Larry's salary to $1000
**Actions of T2 :**
1. set Harry's salary to $2000.
2. set Larry's salary to $2000
Any serial order of T1 and T2 , satisfies the **Consistent criterion .**
Notice that neither transaction reads a salary value before writing it, such a write is called a blind write

## WW CONFLICTS (OVERWRITING UNCOMMITTED DATA)

Now, consider the following interleaving of the actions of T1 and T2:
1. T1 sets Harry's salary to $1,000
2. T2 sets Larry's salary to $2,000
3. T1 sets Larry's salary to $1,000
4. T2 sets Harry's salary to $2,000

It violates the desired consistency criterion that the two salaries must be equal.
The result is not identical to the result of either of the two possible serial executions, and the interleaved schedule is therefore not serializable.

What is a serializable schedule ?

A schedule S whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule. i.e.

When an equivalent Complete Serial Schedule S' is executed against a consistent database, the result is same.

**Example:** In following given two schedules, *the database is consistent in both i.e. sum A+B is preserved*

| T1 | T2 |
|---|---|
| R(A)<br>W(A) | |
| | R(A)<br>W(A) |
| R(B)<br>W(B) | |
| | R(B)<br>W(B) |

A SERIALIZABLE SCHEDULE

| T1 | T2 |
|---|---|
| R(A)<br>W(A)<br>R(B)<br>W(B) | |
| | R(A)<br>W(A)<br>R(B)<br>W(B) |

AN EQUIVALENT SERIAL SCHEDULE

# Conflict Serializability (Cont.)

If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.

We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Result equivalent Two schedules are called ***result equivalent*** if they produce the same final state of the database

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| | write($A$) |
| read($B$) | |
| write($B$) | |
| | read($B$) |
| | write($B$) |

Schedule A

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| read($B$) | |
| | write($A$) |
| write($B$) | |
| | read($B$) |
| | write($B$) |

Conflict Equivalent of schedule A

# Conflict Serializability (Cont.)

- S1 below can be transformed into S2, a serial schedule where $T_2$ follows $T_1$, by series of swaps of non-conflicting instructions. Therefore Schedule 1 is conflict serializable.

S1

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

S2

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

Example of a schedule that is not conflict serializable:

| $T_3$ | $T_4$ |
|---|---|
| **read**($Q$) | |
| | **write**($Q$) |
| **write**($Q$) | |

We are unable to swap instructions in the above schedule to obtain either the serial schedule < $T_3$, $T_4$ >, or the serial schedule < $T_4$, $T_3$ >.

# Testing for Serializability

Consider some schedule of a set of transactions $T_1$, $T_2$, ..., $T_n$

**Precedence graph** — a direct graph where the vertices are the transactions (names).

To draw one:

1) Draw a node for each transaction in the schedule

2) For each pair of following ordered conflict operations in S, create a directional arc in the same order

| Ti | Tj | |
|---|---|---|
| W(X) | R(X) | create arc Ti -> Tj |
| R(X) | W(X) | create arc Ti -> Tj |
| W(X) | W(X) | create arc Ti -> Tj |

We may label the arc by the item that was accessed.

y

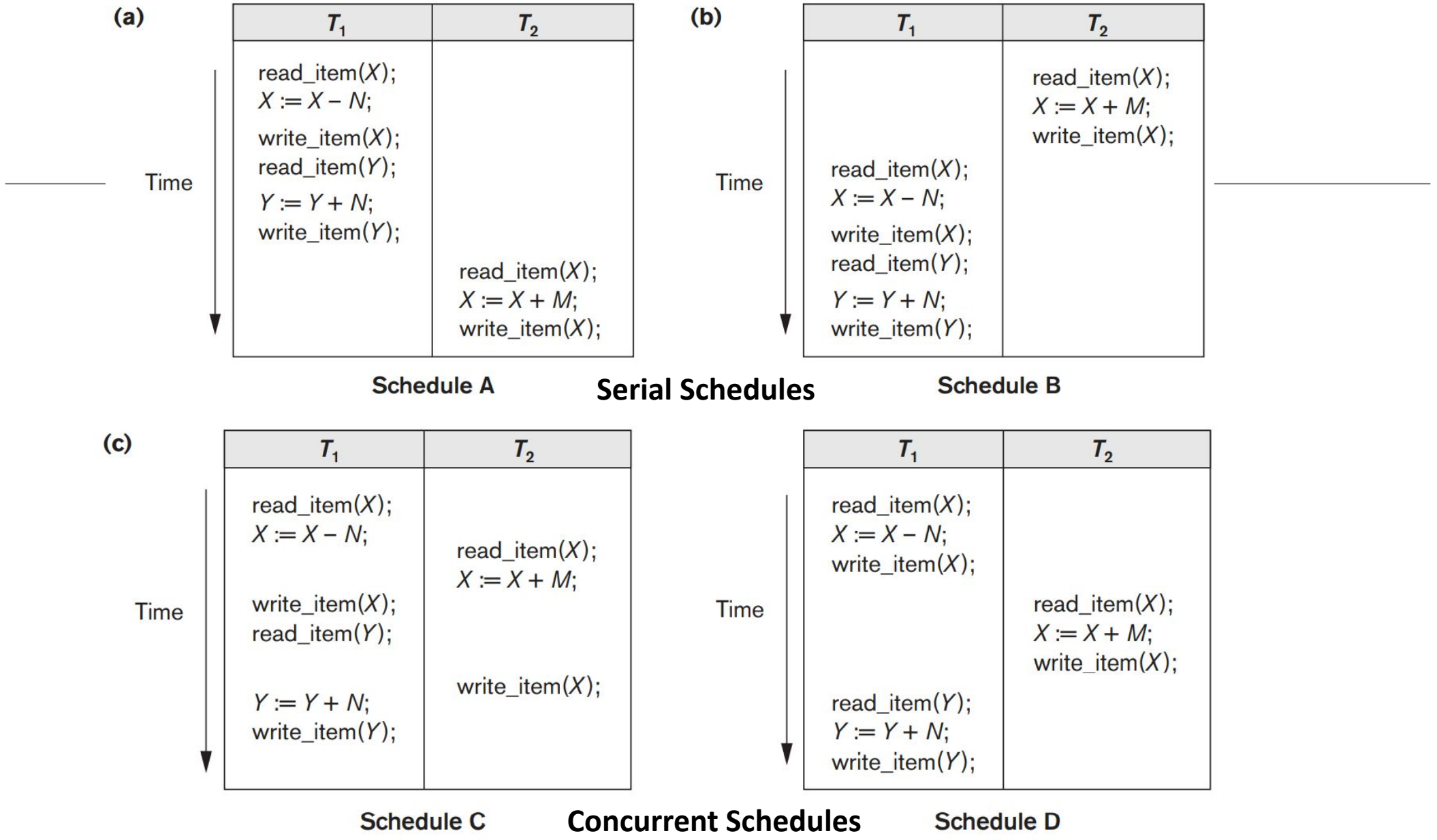
# Test for Conflict Serializability

A schedule is conflict serializable if and only if its precedence graph is acyclic.

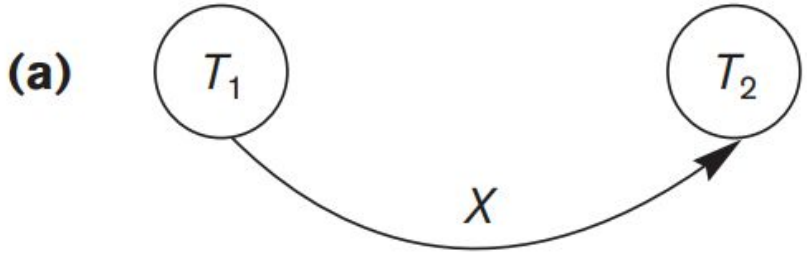
Cycle-detection algorithms exist which take order $n^2$ time, where $n$ is the number of vertices in the graph. (Better algorithms take order $n + e$ where $e$ is the number of edges.)

If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph. This is a linear order consistent with the partial order of the graph.
For example, a serializability order for Schedule A would be
$T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$ .

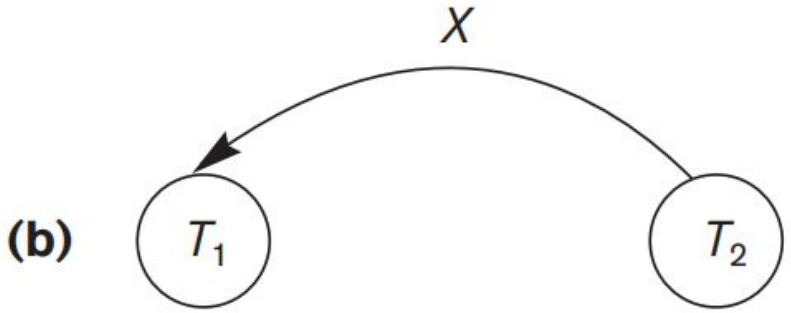
**Topological sorting** for Directed Acyclic Graph (DAG) is a linear **ordering** of vertices such that for every directed edge uv, vertex u comes before v in the **ordering**.



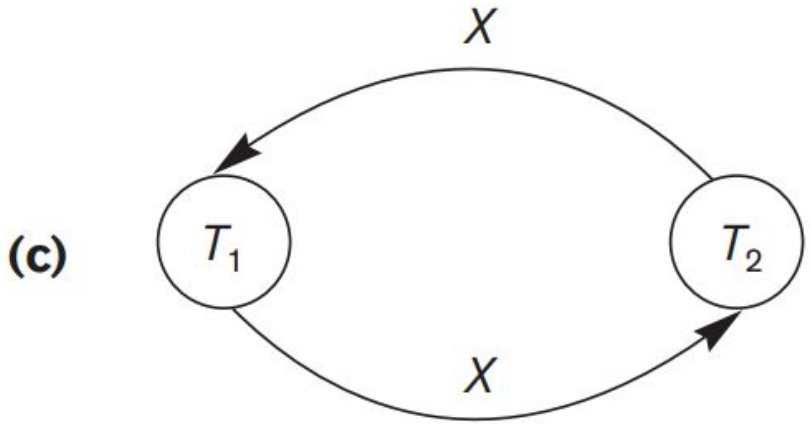

Reading Suggestion & Image Source: Elmasri and Navathe, "Fundamentals of Database Systems", Chapter 20



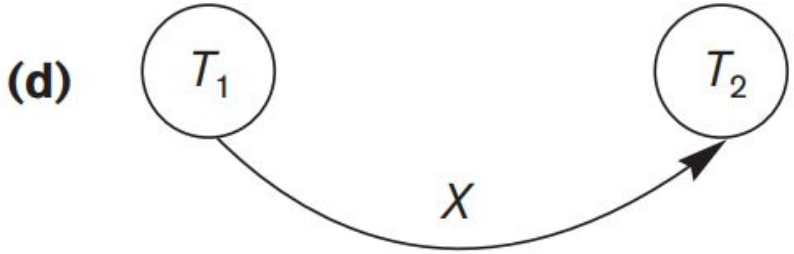

(a) Precedence graph for serial schedule A.




(b) Precedence graph for serial schedule B.




(c) Precedence graph for schedule C (not serializable).

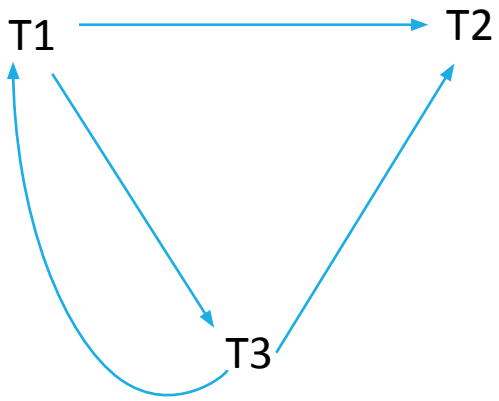



(d) Precedence graph for schedule D (serializable, equivalent to schedule A).

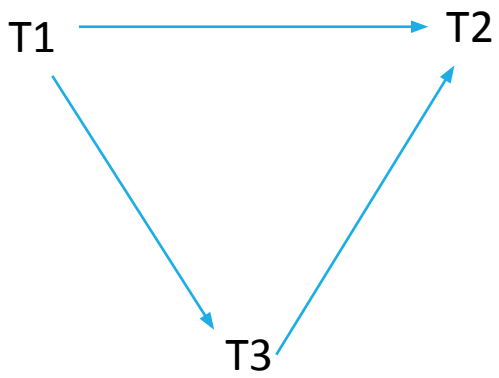
# Example 1

| T1 | T2 | T3 |
|---|---|---|
| R(X) | | |
| | | R(Z) |
| | | W(Z) |
| | R(Y) | |
| R(Y) | | |
| | W(Y) | |
| | | W(X) |
| | W(Z) | |
| W(X) | | |

Here in this graph  cycle is there, theerfore schedule in not conflict serializable



| T1 | T2 | T3 |
|---|---|---|
| R(X) | | |
| | R(Y) | |
| | | R(Y) |
| | W(Y) | |
| W(X) | | |
| | | |
| | | W(X) |
| | R(X) | |
| | W(X) | |




As there is no cycle, this schedule in conflict serializable.

**Order of serializability** □ T1 -> T3 -> T2

# Practice Problems

Consider the following four schedules due to three transactions (indicated by the subscript) using read and write on a data item x, denoted by r(x) and w(x) respectively. Which one of them is conflict serializable.

i. r1(x); r2(x); w1(x); r3(x); w2(x)

Ii. R2(x); r1(x); w2(x); r3(x); w1(x)

Iii. R3(x); r2(x); r1(x); w2(x); w1(x)

Iv. R2(x); w2(x); r3(x); r1(x); w1(x)

Ans: iv

Consider the following schedule S of transactions T1, T2, T3, T4:

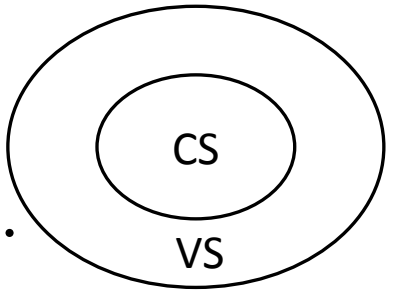
| T1 | T2 | T3 | T4 |
|---|---|---|---|
| | Read(x) | | |
| | | Write(x) | |
| | | C | |
| Write (x) | | | |
| C | | | |
| | Write(y) | | |
| | Read(z) | | |
| | C | | |
| | | | Read(x) |
| | | | Read(y) |
| | | | C |

Check whether S is conflict serializable

Ans: S is conflict-serializable

# View Serializability

- A weaker form of conflict serializability (A schedule which is View serializable may/may not conflict serializable but conflict serializable schedule definitely be View serializable).
- Very difficult to check for view serializable.
- View serializable schedule must have one blind write.



# View Serializability (Cont…)

- Two schedules S and S' are said to be view equivalent if following three conditions are met:
  - Initial read of all data items should be perform by the same transaction in both the schedules.
  - Final write of all data items should be perform by the same transactions in both the schedules.
  - Intermediate read of data items should be perform by the same transactions in both the schedules
- Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation.
- Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.
- A schedule S is said to be view serializable if it is view equivalent to a serial schedule.

# Example 1

| T1 | T2 |
|---|---|
| Read(A) | |
| Write(A) | |
| | Read(A) |
| Read(B) | |
| | Write(A) |
| Write(B) | |
| | Read(B) |
| | Write(B) |

S

| T1 | T2 |
|---|---|
| Read(A) | |
| Write(A) | |
| Read(B)<br>Write(B) | |
| | Read(A) |
| | Write(A) |
| | Read(B) |
| | Write(B) |

S’

| Condition | S | S’ |
|---|---|---|
| Initial read of A | T1 | T1 |
| Initial read of B | T1 | T1 |
| Final write of A | T2 | T2 |
| Final write of B | T2 | T2 |
| Intermediate read of A | T2 | T2 |
| Intermediate read of B | T2 | T2 |

**S and S’ are View equivalent, and S’ is serializable. Hence S is view serializable**

# Example 2

S

| T1 | T2 | T3 |
|---|---|---|
| R(a) | | |
| | W(a) | |
| W(a) | | |
| | | W(a) |

S’

| T1 | T2 | T3 |
|---|---|---|
| R(a) | | |
| W(a) | | |
| | W(a) | |
| | | W(a) |

To check whether S is view serializable or not, we need to consider the possibilities of serial scedules.
3 transactions are there, so total 3! Serial schedules are possible.
123, 231, 321, 213, 312, 132, Now we need to check with whether S is view equivalent with any of the possible serial schedules.

| Condition | S | S’ |
|---|---|---|
| Initial read of a | T1 | T1 |
| Final write of a | T3 | T3 |
| Intermediate read of A | NA | NA |

**S and S’ are View equivalent, and S’ is serializable. Hence S is view serializable**

# Recoverability

Need to address the effect of transaction failures on concurrently running transactions.

**Recoverable schedule** — if a transaction $T_j$ reads a data items previously written by a transaction $T_i$, the commit operation of $T_i$ appears before the commit operation of $T_j$.

The following schedule is not recoverable if *Tj* commits immediately after the write

| Ti | Tj |
|---|---|
| read(A) | |
| write(A) | |
| | read(A) |
| | write(A)<br>commit |
| commit | |

This is a serial schedule and conflict serializable also. Then also the schedule is irrecoverable and can go in inconsistent state after this.

If *Ti* should abort, *Tj* would have read (and possibly shown to the user) an inconsistent database state. Hence database must ensure that schedules are recoverable.

# Recoverable Schedule

- If any problem occur before the commit operation, we need to rollback the operation. For this the schedules should be recoverable.
- Schedules are recoverable iff it don't have Dirty Read.
- Dirty Read: read the uncommitted value of the data item by the other transaction.
- If schedule has dirty read but if the transaction are committing in the order same as the order of dirty read, then the schedule will be recoverable.

# Example (Which schedule are recoverable)

S1

| T1 | T2 |
|---|---|
| Read(A) | |
| A=A+10 | |
| Write(A) | |
| | Read(A) |
| | A=A-5 |
| | Write(A) |
| Commit | |
| | Commit |

S2

| T1 | T2 |
|---|---|
| Read(A) | |
| A=A+10 | |
| Write(A) | |
| | Read(A) |
| | A=A-5 |
| | Write(A) |
| | Commit |
| Commit | |

S3

| T1 | T2 |
|---|---|
| Read(A) | |
| A=A+10 | |
| Write(A) | |
| | Read(B) |
| | B=B-5 |
| | Write(B) |
| | Commit |
| Commit | |

- **S1 is recoverable.**
- **S2 is not recoverable.**
- **S3 is recoverable.**

# Recoverability (Cont.)

**Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

| Ti -> Tj -> Tk | Ti -> Tj -> Tk |
|---|---|
| c | c |
| c | c |
| c | c |
| Not Recoverable | Recoverable |

| Ti | Tj | Tk |
|---|---|---|
| read(A) | | |
| write(A) | | |
| | read(A) | |
| | write(A) | |
| | | read(A) |

If *Ti* fails, *Tj* and *Tk* must also be rolled back.

Can lead to the undoing of a significant amount of work

# Cascadeless Schedule

- If the systems does not contain cascading rollbacks.

| T1 | T2 | T3 |
|---|---|---|
| Read(A) | | |
| Write(A) | | |
| | Read(A) | |
| | Write(A) | |
| | | Read(A) |
| | | Write(A) |
| C | | |
| | C | |
| | | C |

- If T1 fails before commit then it will rollback, and then T2 and T3 will also rollback. This is cascading schedules.
- So if dirty read is there then schedule definitely will be cascade schedule.
- This is optional property.
- If the schedule doesn't contain dirty read, it will be cascadeless inspite of having blind write.

# Example

| T1 | T2 | T3 |
|---|---|---|
| Read(A) | | |
| Write(A)<br>C | | |
| | | |
| | Write(A)<br>C | |
| | | Read(B) |
| | | Write(B) |
| | | C |

Cascadeless schedule, dirty read is not there.

# Recoverability (Cont.)

**Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions $T_i$ and $T_j$ such that $T_j$ reads a data item previously written by $T_i$, the commit operation of $T_i$ appears before the read operation of $T_j$.

| Ti | Tj | Tk |
|---|---|---|
| read(A) | | |
| write(A)<br>commit | | |
| | read(A) | |
| | write(A)<br>commit | |
| | | read(A)<br>commit |

Every cascadeless schedule is also recoverable

It is desirable to restrict the schedules to those that are cascadeless

# Strict Schedule

- Other transaction can perform read/write of the data item only after the commit called as strict schedule.

| T1 | T2 | T3 |
|---|---|---|
| Read(A) | | |
| Write(A) | | |
| C | | |
| | Read(A) | |
| | Write(A) | |
| | C | |
| | | Read(A) |
| | | Write(A) |
| | | C |

Consider the following schedule S of transactions T1, T2, T3, T4:

| T1 | T2 | T3 | T4 |
|---|---|---|---|
| | Read(x) | | |
| | | Write(x) | |
| | | C | |
| Write (x) | | | |
| C | | | |
| | Write(y) | | |
| | Read(z) | | |
| | C | | |
| | | | Read(x) |
| | | | Read(y) |
| | | | C |

Check whether S is conflict serializable and recoverable or not.

Ans: S is both conflict-serializable and recoverable

# Thankyou