M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

# Course Name: Database Systems

# Course Code: CS52

# Credits: 3:1:0

# UNIT 5

# Term: Oct 2021 – Feb 2022

---

Faculty:
Dr. Sini Anna Alex

RAMAIAH
Institute of Technology

# Chapter 17

## INTRODUCTION TO TRANSACTION PROCESSING CONCEPTS AND THEORY

RAMAIAH
Institute of Technology

# Introduction to Transaction Processing

**Single-User System**:

- At most one user at a time can use the system.

**Multiuser System**:

- Many users can access the system concurrently.

**Concurrency**

- **Interleaved processing**:
  - Concurrent execution of processes is interleaved in a single CPU.
- **Parallel processing**:
  - Processes are concurrently executed in multiple CPUs.

RAMAIAH
Institute of Technology

# Introduction to Transaction Processing

A **Transaction**:

- Logical unit of database processing that includes one or more access operations (read - retrieval, write - insert or update, delete).

A transaction (set of operations) may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.

**Transaction boundaries**:

- Begin and End transaction.

An **application program** may contain several transactions separated by the Begin and End transaction boundaries.

RAMAIAH
Institute of Technology

# Introduction to Transaction Processing

SIMPLE MODEL OF A DATABASE (for purposes of discussing transactions):

**A database** is a collection of named data items

**Granularity** of data - a field, a record , or a whole disk block (Concepts are independent of granularity)

Basic operations are **read** and **write**

- **read_item(X)**: Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
- **write_item(X)**: Writes the value of program variable X into the database item named X.

RAMAIAH
Institute of Technology

# Introduction to Transaction Processing

READ AND WRITE OPERATIONS:

Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.

read_item(X) command includes the following steps:

- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
- Copy item X from the buffer to the program variable named X.

RAMAIAH
Institute of Technology

# Introduction to Transaction Processing

READ AND WRITE OPERATIONS (contd.):

**write_item(X)** command includes the following steps:

- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
- Copy item X from the program variable named X into its correct location in the buffer.
- Store the updated block from the buffer back to disk (either immediately or at some later point in time).

RAMAIAH Institute of Technology

# Two sample transactions

Two sample transactions:

- (a) Transaction T1
- (b) Transaction T2

(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);
```

RAMAIAH
Institute of Technology

# Introduction to Transaction Processing

## Why Concurrency Control is needed:

**The Lost Update Problem**

- This occurs when two transactions that access **the same database items** have their operations interleaved in a way that makes the value of some database item incorrect.

**The Temporary Update (or Dirty Read) Problem**

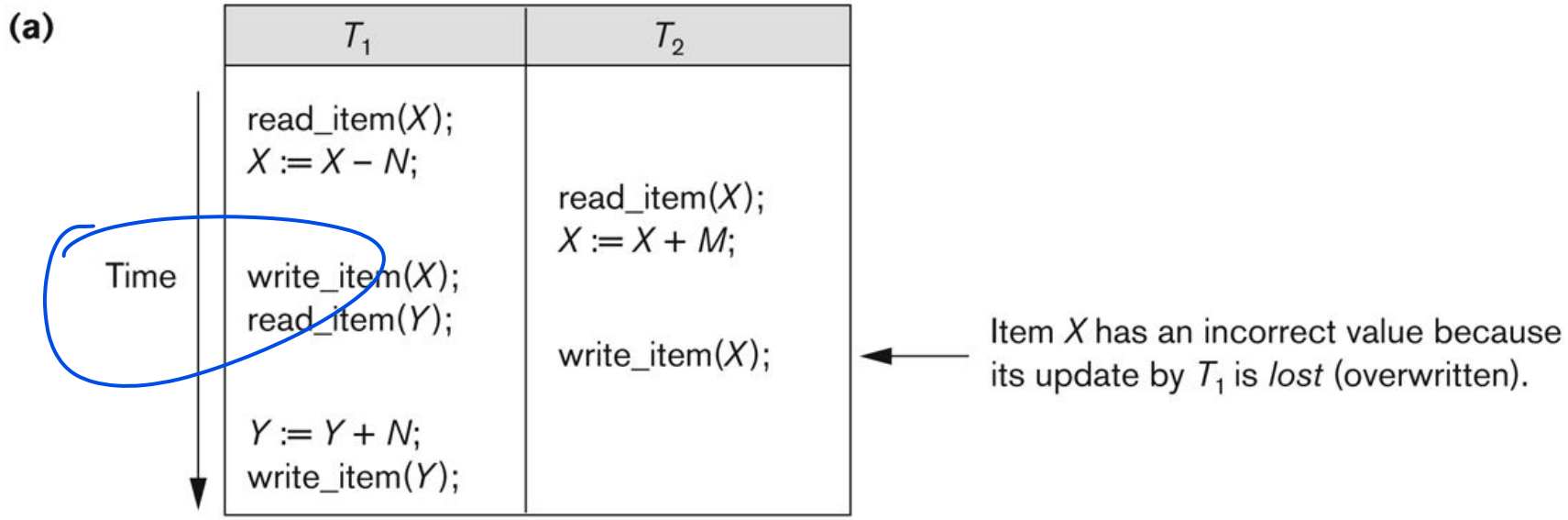
- This occurs when one **transaction updates a database item** and then the transaction fails for some reason.
- The updated item is accessed by another transaction before it is changed back to its original value.

**The Incorrect Summary Problem**

- If one transaction is **calculating an aggregate summary function** on a number of records while **other transactions are updating some of these records**, the aggregate function may calculate some values before they are updated and others after they are updated.

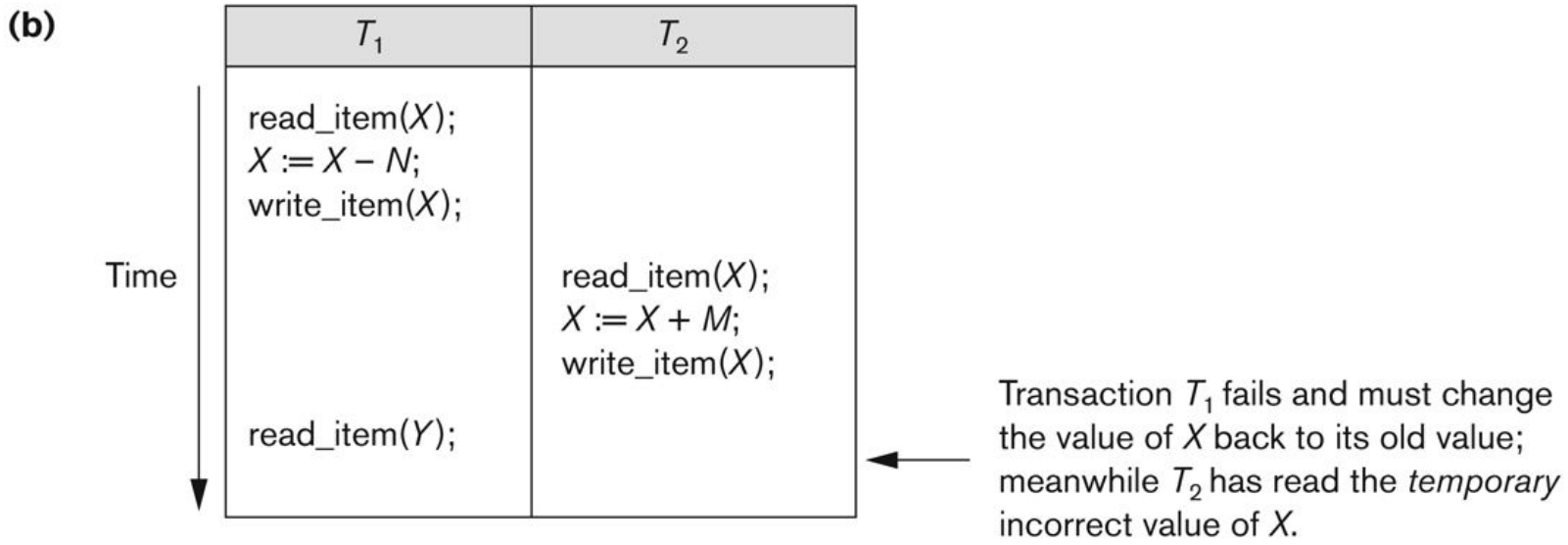
# Concurrent execution is uncontrolled: (a) The lost update problem.

**Figure 17.3**
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



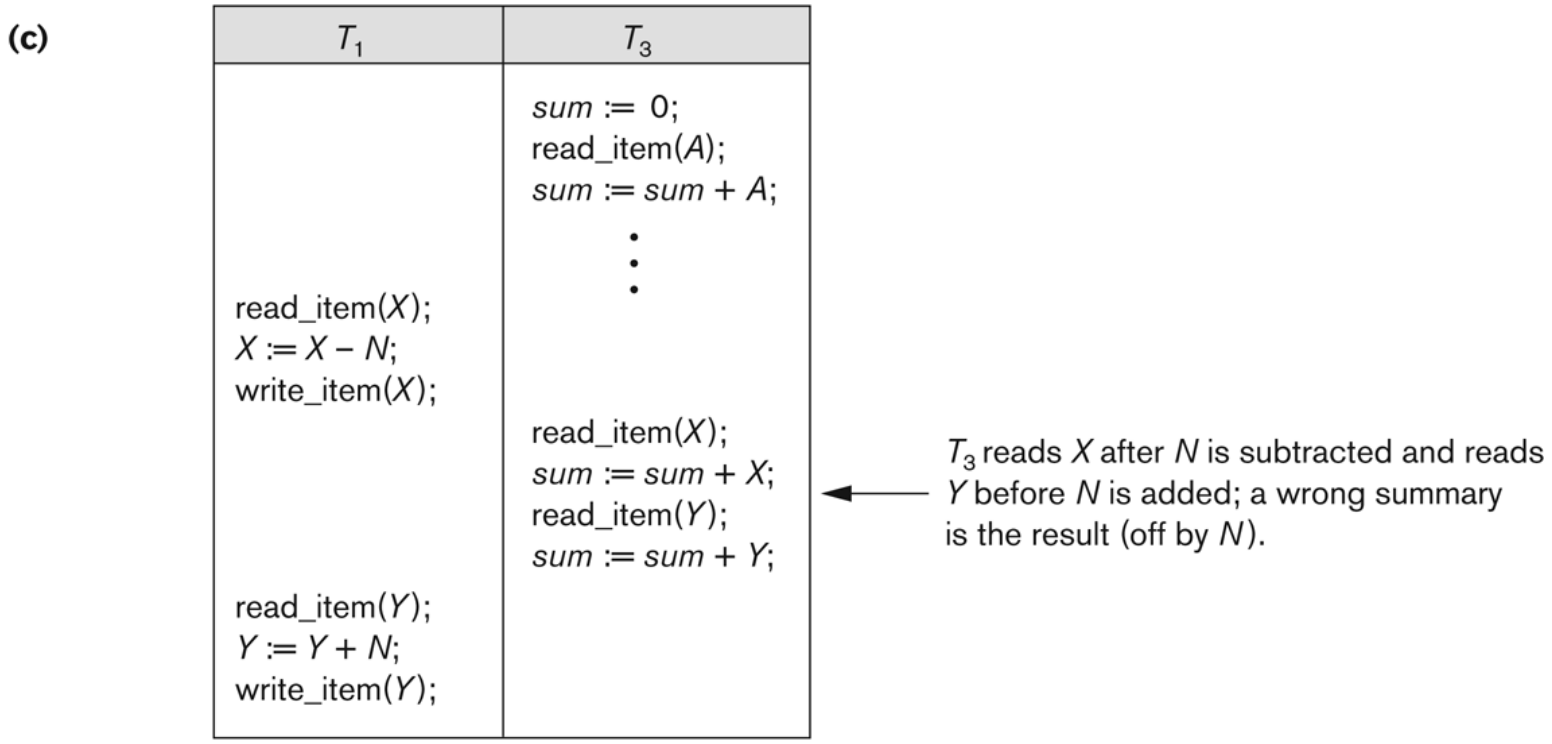
# Concurrent execution is uncontrolled: (b) The temporary update problem.

**Figure 17.3**
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



# Concurrent execution is uncontrolled: (c) The incorrect summary problem.

**Figure 17.3**
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



RAMAIAH
Institute of Technology

# Introduction to Transaction Processing

Why **recovery** is needed:

(What causes a Transaction to fail)

1. A computer failure (system crash):

   A **hardware or software error** occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. A transaction or system error:

   Some operation in the transaction may cause it to fail, such as **integer overflow or division by zero**. Transaction failure may also occur because of **erroneous parameter values or because of a logical programming error**. In addition, the **user may interrupt** the transaction during its execution.

RAMAIAH
Institute of Technology

# Introduction to Transaction Processing

Why **recovery** is needed (Contd.):

(What causes a Transaction to fail)

3. Local errors or exception conditions detected by the transaction:

   Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as **insufficient account balance in a banking database**, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

   A programmed abort in the transaction causes it to fail.

4. Concurrency control enforcement:

   The concurrency control method may decide to abort the transaction, to be restarted later, because it **violates serializability** or because several transactions are in a state of **deadlock** .

RAMAIAH
Institute of Technology

# Introduction to Transaction Processing

Why **recovery** is needed (contd.):

(What causes a Transaction to fail)

5. Disk failure:

   Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes:

   This refers to an endless list of problems that includes **power or air-conditioning failure, fire, theft**, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

RAMAIAH
Institute of Technology

# Transaction and System Concepts

A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.

- For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

**Transaction states**:

- Active state
- Partially committed state
- Committed state
- Failed state
- Terminated State

RAMAIAH
Institute of Technology

# State transition diagram illustrating the states for transaction execution

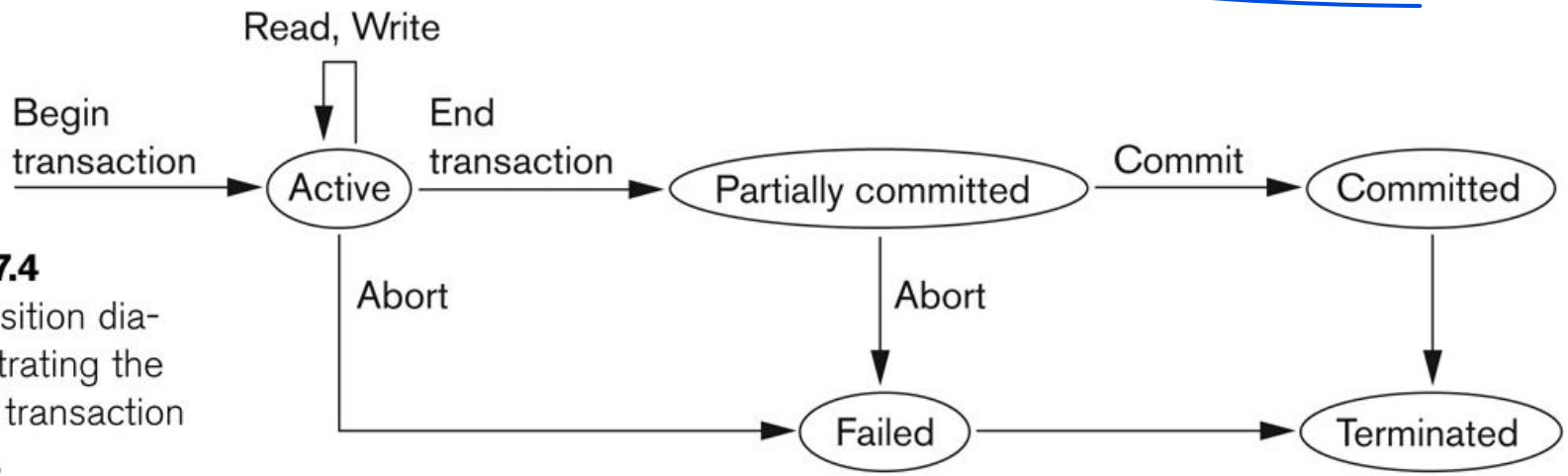



**Figure 17.4**
State transition diagram illustrating the states for transaction execution.

RAMAIAH
Institute of Technology

# Transaction and System Concepts

Recovery manager keeps track of the following operations:

- **begin_transaction**: This marks the beginning of transaction execution.
- **read** or **write**: These specify read or write operations on the database items that are executed as part of a transaction.
- **end_transaction**: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
  - At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

RAMAIAH
Institute of Technology

# Transaction and System Concepts

Recovery manager keeps track of the following operations (contd…):

- **commit_transaction**: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
- **rollback** (or **abort**): This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

Recovery techniques use the following operators:

- **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
- **redo**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

RAMAIAH
Institute of Technology

# Transaction and System Concepts

## The System Log

- **Log** or **Journal**: The log keeps **track of all transaction operations that affect the values of database items.**
  - This information may be needed to permit recovery from transaction failures.
  - The **log is kept on disk**, so it is **not affected by any type of failure except for disk or catastrophic failure**.
  - In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

RAMAIAH
Institute of Technology

# Transaction and System Concepts

The System Log (cont):

- T in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:

- **Types of log record**:
  - **[start_transaction,T]**: Records that transaction T has started execution.
  - **[write_item,T,X,old_value,new_value]:** Records that transaction T has changed the value of database item X from old_value to new_value.
  - **[read_item,T,X]:** Records that transaction T has read the value of database item X.
  - **[commit,T]:** Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
  - **[abort,T]:** Records that transaction T has been aborted.

RAMAIAH
Institute of Technology

# Transaction and System Concepts

Recovery using log records:

If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques described in Chapter 19.

1. Because the **log contains a record of every write operation that changes the value of some database item**, it is possible to **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.
2. We can also **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

RAMAIAH
Institute of Technology

# Transaction and System Concepts

## Commit Point of a Transaction:

### Definition a Commit Point:

- A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log.
- Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database.
- The transaction then writes an entry [commit,T] into the log.

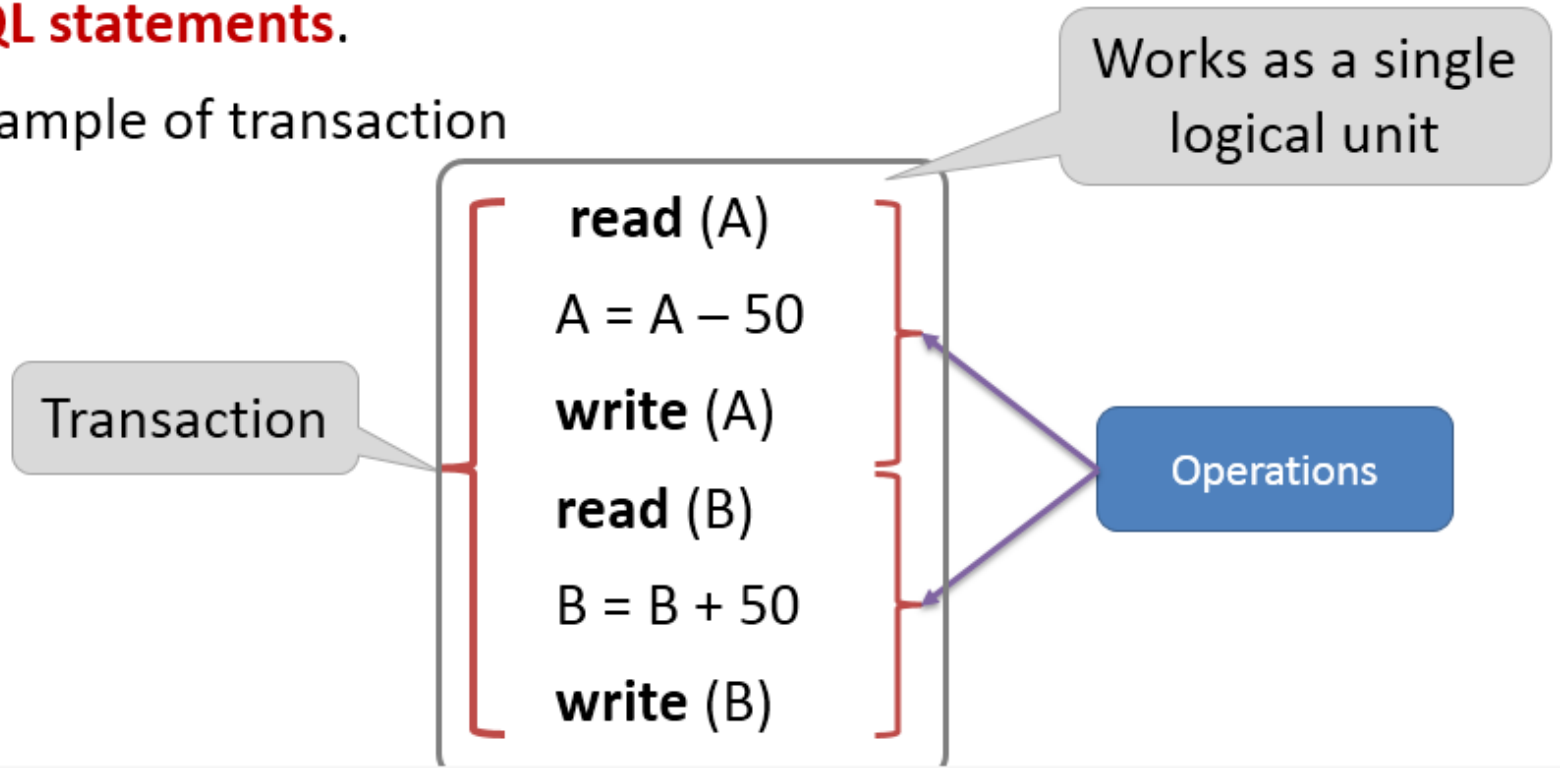
### Roll Back of transactions:

- Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

RAMAIAH

# What is transaction?

- A transaction is a **sequence of operations performed as a single logical unit of work**.
- A transaction is a **logical unit of work that contains one or more SQL statements**.
- Example of transaction



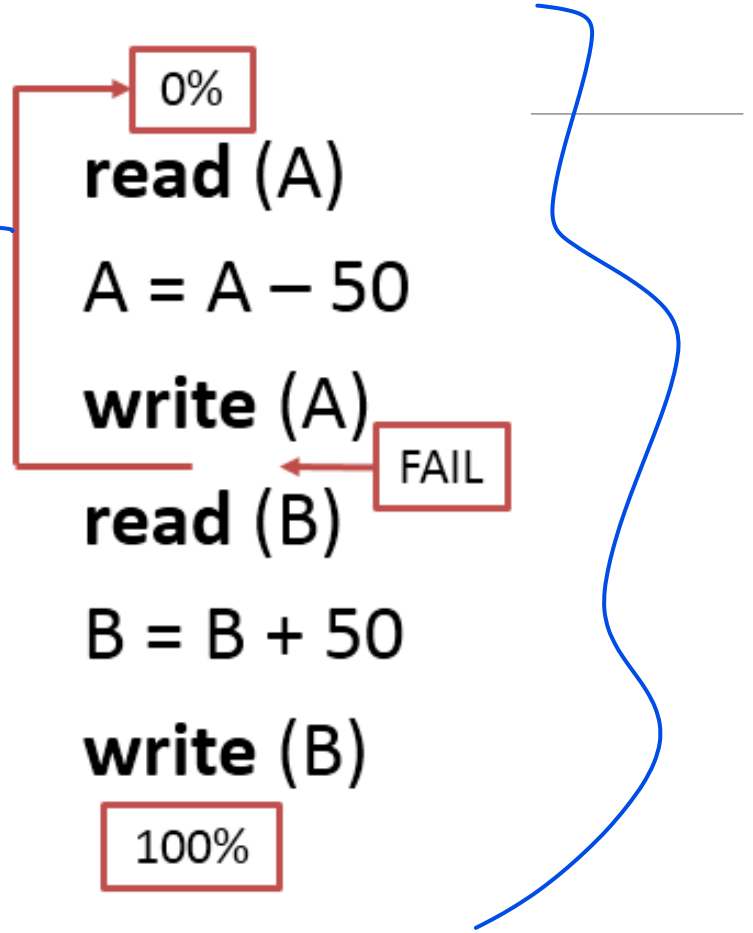
RAMAIAH
Institute of Technology

# ACID properties of transaction

- Atomicity (Either transaction execute 0% or 100%)
- Consistency (database must remain in a consistent state after any transaction)
- Isolation (Intermediate transaction results must be hidden from other concurrently executed transactions)
- Durability (Once a transaction completed successfully, the changes it has made into the database should be permanent)

# ACID properties of transaction

- **Atomicity**
  - This property states that a **transaction must be treated as an atomic unit**, that is, **either all of its operations are executed or none**.
  - **Either transaction execute 0% or 100%**.
  - For example, consider a transaction to transfer Rs. 50 from account A to account B.
  - In this transaction, if Rs. 50 is deducted from account A then it must be added to account B.



# ACID properties of transaction

- **Consistency**
  - The **database must remain in a consistent state after any transaction**.
  - 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.
  - In our example, total of A and B must remain same before and after the execution of transaction.

A=500, B=500

A+B=1000

**read** (A)

A = A – 50

**write** (A)

**read** (B)

B = B + 50

**write** (B)

A=450, B=550

A+B=1000

# ACID properties of transaction

- **Isolation**
  - **Changes occurring in a particular transaction will not be visible to any other transaction until it has been committed.**
  - **Intermediate transaction results must be hidden from other concurrently executed transactions.**
  - In our example once our transaction starts from first step (step 1) its result should not be access by any other transaction until last step (step 6) is completed.

**read** (A)

A = A – 50

**write** (A)

**read** (B)

B = B + 50

**write** (B)

# ACID properties of transaction

- **Durability**
  - After a transaction completes successfully, **the changes it has made to the database persist (permanent)**, even if there are system failures.
  - Once our transaction completed up to last step (step 6) its result must be stored permanently. It should not be removed if system fails.

A=500, B=500

**read** (A)

A = A – 50

**write** (A)

**read** (B)

B = B + 50

**write** (B)

A=450, B=550

# What is schedule?

- A schedule is a **process of grouping the transactions** into one and **executing them in a predefined order**.
- A schedule is the **chronological (sequential) order** in which **instructions are executed** in a system.
- A schedule is required in a database because **when some transactions execute in parallel, they may affect the result of the transaction**.
- Means if one transaction is updating the values which the other transaction is accessing, then the order of these two transactions will change the result of another transaction.
- Hence a schedule is created to execute the transactions.

RAMAIAH
Institute of Technology

# Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

**Recoverable schedule**:

- One where no transaction needs to be rolled back.
- A schedule S is recoverable **if no transaction T in S commits** until all transactions T' that have written an item that T reads have committed.

**Cascadeless schedule**:

- One where **every transaction reads only the items** that are **written by committed transactions.**

RAMAIAH
Institute of Technology

# Characterizing Schedules based on Recoverability

Schedules classified on recoverability (contd.):

**Schedules requiring cascaded rollback**:

- A schedule in which **uncommitted transactions that read an item from a failed transaction** must be rolled back.

**Strict Schedules**:

- A schedule in which a **transaction can neither read or write an item X until the last transaction that wrote X has committed**.

RAMAIAH Institute of Technology

# Characterizing Schedules Based on Recoverability

A shorthand notation for describing a schedule uses the symbols ***b*, *r*, *w*, *e*, *c*, and *a*** for the operations begin_transaction, read_item, write_item, end_transaction, commit, and abort, respectively.

$S_a$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;

Same as $S_a$ *except 2 commit operations*

$S_b$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$;$c_2$; $w_1(Y)$;$c_1$

$S_b$ *is recoverable , even though it suffers from lost update problem*

$S_c$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $c_2$;$a_1$;

Is not recoverable because T2 reads item X from T1, and the T2 commits before T1 commits.

RAMAIAH

# Serial schedule

- A serial schedule is one in which **no transaction starts until a running transaction has ended**.
- Transactions are executed one after the other.
- This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

# Example of serial schedule

| T1 | T2 |
|---|---|
| Read (A) | |
| Temp = A * 0.1 | |
| A = A - temp | |
| Write (A) | |
| Read (B) | |
| B = B + temp | |
| Write (B) | |
| Commit | |
| | Read (A) |
| | A = A - 50 |
| | Write (A) |
| | Read (B) |
| | B = B + 50 |
| | Write (B) |
| | Commit |

RAMAIAH
Institute of Technology

# Interleaved schedule

- Schedule that **interleave the execution of different transactions**.
- Means **second transaction is started before the first one could end** and **execution can switch between the transactions back and forth**.

RAMAIAH Institute of Technology

# Example of interleaved schedule

| T1 | T2 |
|---|---|
| | Read (A) |
| | Temp = A * 0.1 |
| | A = A - temp |
| Read (B) | Write (A) |
| B = B + temp | |
| Write (B) | |
| Commit | |
| | Read (A) |
| | A = A - 50 |
| | Write (A) |
| Read (B) | |
| B = B + 50 | |
| Write (B) | |
| Commit | |

# Serializability

- A schedule is serializable **if it is equivalent to a serial schedule**.
- In serial schedules, **only one transaction is allowed to execute at a time** i.e. no concurrency is allowed.
- Whereas in serializable schedules, **multiple transactions can execute simultaneously** i.e. concurrency is allowed.
- Types (forms) of serializability
  1. Conflict serializability
  2. View serializability

RAMAIAH Institute of Technology

# Conflicting instructions

- Let $I_i$ and $I_j$ be two instructions of transactions $T_i$ and $T_j$ respectively.

1. $I_i$ = read(Q), $I_j$ = read(Q)

   $I_i$ and $I_j$ don't conflict

| T1 | T2 |
|---|---|
| read (Q) | |
| | read (Q) |

| T1 | T2 |
|---|---|
| | read (Q) |
| read (Q) | |

2. $I_i$ = read(Q), $I_j$ = write(Q)

   $I_i$ and $I_j$ conflict

| T1 | T2 |
|---|---|
| read (Q) | |
| | write(Q) |

| T1 | T2 |
|---|---|
| | write(Q) |
| read (Q) | |

3. $I_i$ = write(Q), $I_j$ = read(Q)

   $I_i$ and $I_j$ conflict

| T1 | T2 |
|---|---|
| write(Q) | |
| | read (Q) |

| T1 | T2 |
|---|---|
| | read (Q) |
| write(Q) | |

4. $I_i$ = write(Q), $I_j$ = write(Q)

   $I_i$ and $I_j$ conflict

| T1 | T2 |
|---|---|
| write(Q) | |
| | write(Q) |

| T1 | T2 |
|---|---|
| | write(Q) |
| write(Q) | |

# Conflicting instructions

- Instructions **$I_i$ and $I_j$ conflict** if and only if there exists some item **Q accessed by both Ii and Ij**, and **at least one of these instructions wrote Q**.
- If **both the transactions access different data item** then they are **not conflict**.

# Conflict serializability

- If a given **schedule can be converted into a serial schedule by swapping its non-conflicting operations**, then it is called as a conflict serializable schedule.

# Conflict serializability

- Example of a schedule that is not conflict serializable:

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

- We are **unable to swap instructions** in the above schedule to obtain either the serial schedule < T1, T2 >, or the serial schedule < T2, T1 >.

# View Serializability

- View equivalence of two schedules
  - 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
  - Read operations said to see the same view in both schedules
- View serializable schedule
  - View equivalent to a serial schedule

RAMAIAH
Institute of Technology

# View Serializability

- Conflict serializability similar to view serializability if constrained write assumption (no blind writes) applies
- Unconstrained write assumption
  - Value written by an operation can be independent of its old value
- Debit-credit transactions
  - Less-stringent conditions than conflict serializability or view serializability

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

# Characterizing Schedules Based on Serializability (cont'd.)

Testing for serializability of a schedule

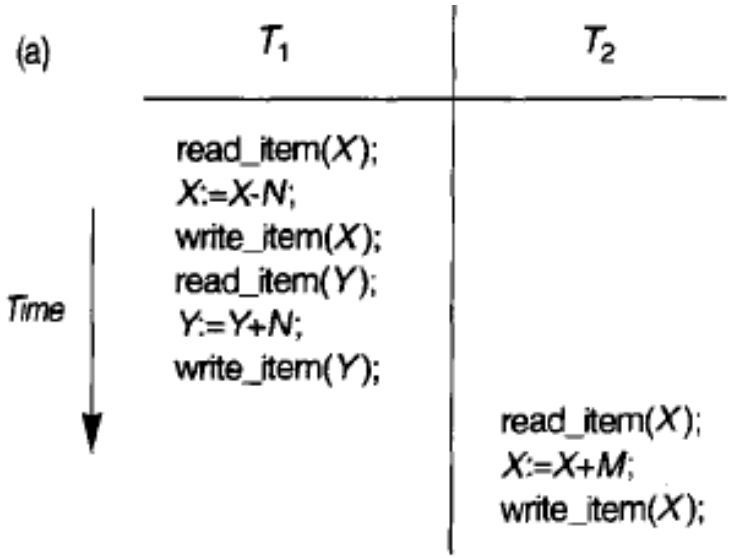
1. For each transaction $T_i$ participating in schedule $S$, create a node labeled $T_i$ in the precedence graph.
2. For each case in $S$ where $T_j$ executes a read_item($X$) after $T_i$ executes a write_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in $S$ where $T_j$ executes a write_item($X$) after $T_i$ executes a read_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in $S$ where $T_j$ executes a write_item($X$) after $T_i$ executes a write_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule $S$ is serializable if and only if the precedence graph has no cycles.

RAMAIAH Institute of Technology

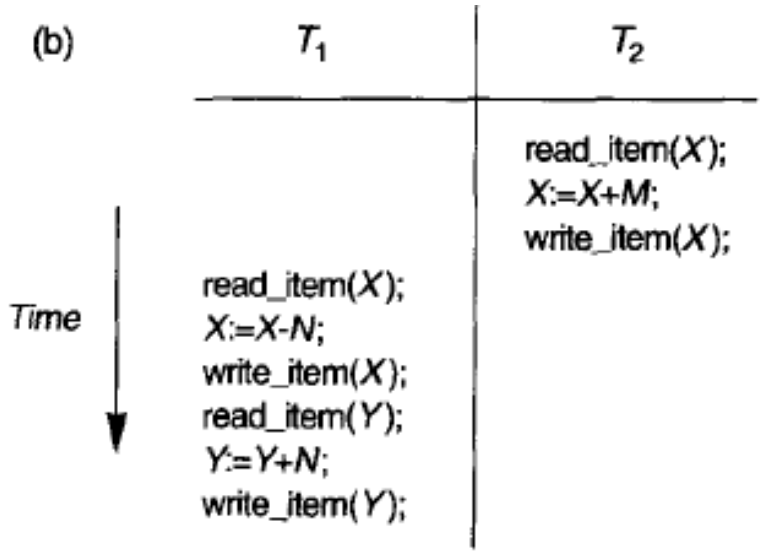
# Constructing the Precedence Graphs

Constructing the precedence graphs for schedules A and D to test for conflict serializability.

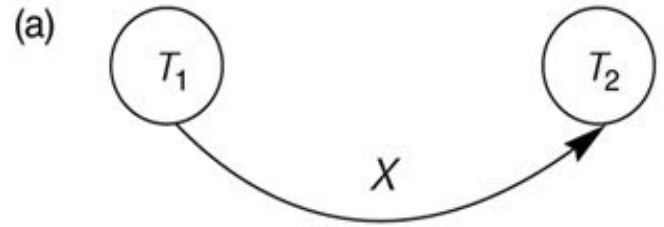
- (a) Precedence graph for serial schedule A.
- (b) Precedence graph for serial schedule B.

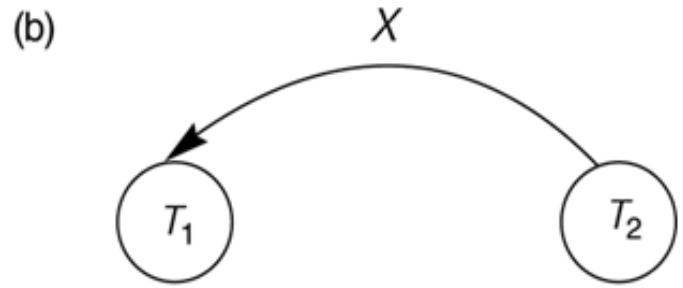



Schedule A




Schedule B



Constructing the precedence graphs for schedules A and D to test for conflict serializability.
(c) Precedence graph for schedule C (not serializable).
(d) Precedence graph for schedule D (serializable, equivalent to schedule A).

(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($Y$); | |

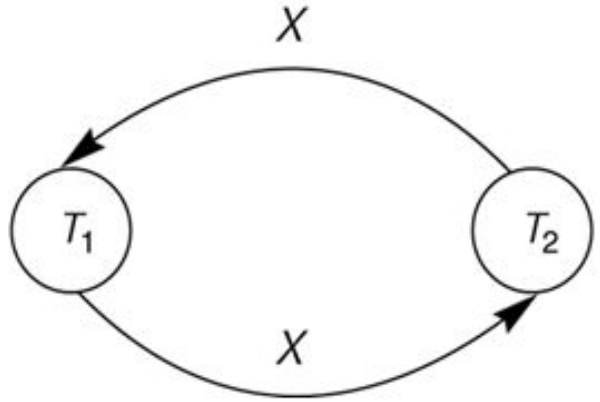
Time

Schedule $C$

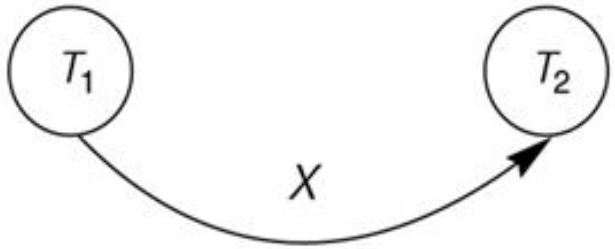
| $T_1$ | $T_2$ |
|---|---|
| read_item($X$); | |
| $X$:=$X$-$N$; | |
| write_item($X$); | |
| | read_item($X$); |
| | $X$:=$X$+$M$; |
| | write_item($X$); |
| read_item($Y$); | |
| $Y$:=$Y$+$N$; | |
| write_item($Y$); | |

Schedule $D$

(c)




(d)



RAMAIAH
Institute of Technology

# Another example of serializability Testing

**Figure 17.8**
Another example of serializability testing. (a) The read and write operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule E. (c) Schedule *F*.

**(a)**

| Transaction $T_1$ |
|---|
| read_item($X$); |
| write_item($X$); |
| read_item($Y$); |
| write_item($Y$); |

| Transaction $T_2$ |
|---|
| read_item($Z$); |
| read_item($Y$); |
| write_item($Y$); |
| read_item($X$); |
| write_item($X$); |

| Transaction $T_3$ |
|---|
| read_item($Y$); |
| read_item($Z$); |
| write_item($Y$); |
| write_item($Z$); |

RAMAIAH
Institute of Technology

# Another Example of Serializability Testing

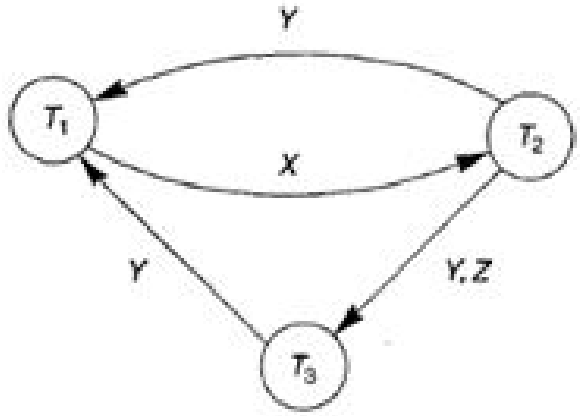
**Figure 17.8**
Another example of serializability testing. (a) The read and write operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule E. (c) Schedule F.

**(b)**

Time ↓

| 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($X$); | |

**Schedule E**




Equivalent serial schedules

None

Reason

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

RAMAIAH Institute of Technology

# Another Example of Serializability Testing

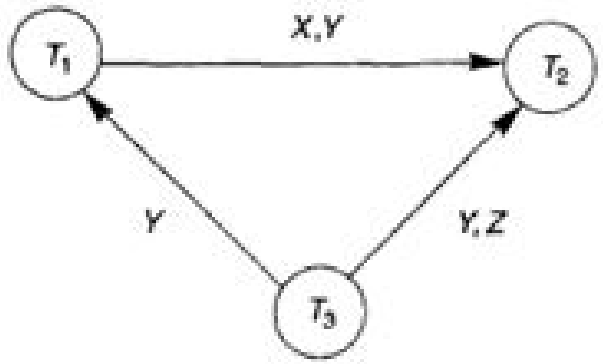
**Figure 17.8**
Another example of serializability testing. (a) The read and write operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule E. (c) Schedule *F*.

(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

$T_3 \rightarrow T_1 \rightarrow T_2$

RAMAIAH Institute of Technology

# Isolation levels in Transaction Processing

A transaction isolation level is defined by the following phenomena –

**Dirty Read –** A Dirty read is the situation when a transaction reads a data that has not yet been committed. Reading a value that was written by a transaction which failed.

$S_b$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$;$c_2$; $w_1(Y)$;$c_1$

**Non Repeatable read –** Non Repeatable read occurs when a transaction reads same row twice, and get a different value each time. Allowing another transaction to write a new value between multiple reads of one transaction.

- A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.

**Phantom Read –** Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different.

Based on these phenomena, The SQL standard defines four isolation levels :

RAMAIAH
Institute of Technology

# Isolation levels in Transaction Processing

**Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.

**Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.

RAMAIAH
Institute of Technology

# Isolation levels in Transaction Processing

**Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.

**Serializable –** This is the Highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

RAMAIAH
Institute of Technology

# Transaction Support in SQL

Possible violation of serializabilty:

| Isolation level | Type of Violation: Dirty read | nonrepeatable read | phantom |
|---|---|---|---|
| READ UNCOMMITTED | yes | yes | yes |
| READ COMMITTED | no | yes | yes |
| REPEATABLE READ | no | no | yes |
| SERIALIZABLE | no | no | no |

RAMAIAH
Institute of Technology

# Which of the following Schedules is conflict serializable? For each serializable schedule , determine equivalent serial schedules

a). r1(X); r3(X); w1(X); r2(X); w3(X);

b). r1(X); r3(X); w3(X); w1(X); r2(X);

c). r3(X); r2(X); w3(X); r1(X); w1(X);

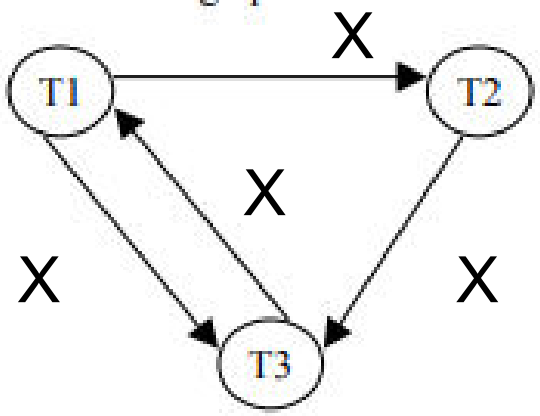
d). r3(X); r2(X); r1(X); w3(X); w1(X);

RAMAIAH
Institute of Technology

# Solution

a). *r1(X); r3(X); w1(X); r2(X); w3(X);*

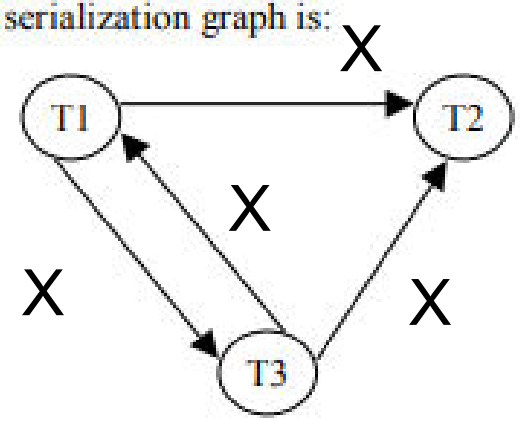
The serialization graph is:




Not serializable.

b). *r1(X); r3(X); w3(X); w1(X); r2(X);*

The serialization graph is:



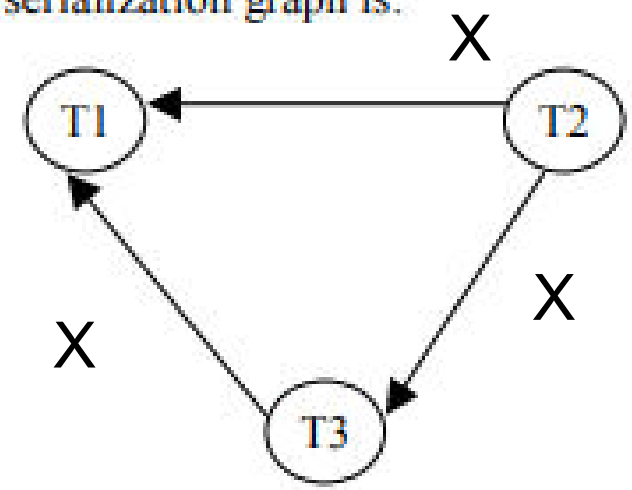

Not serializable.

RAMAIAH
Institute of Technology

# Solution

c). *r3(X); r2(X); w3(X); r1(X); w1(X);*

The serialization graph is:



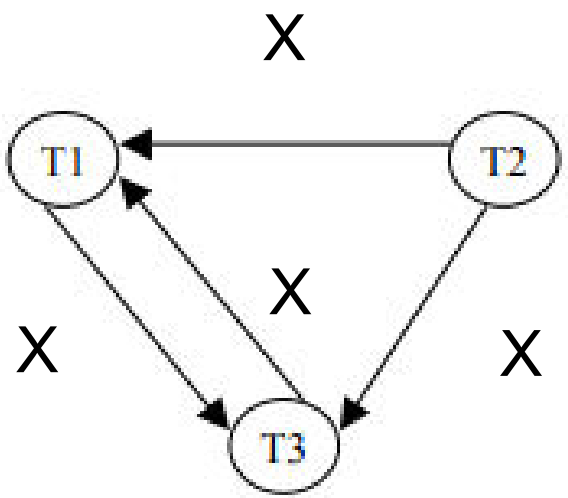

Serializable.

The equivalent serial schedule is: *r2(X); r3(X); w3(X); r1(X); w1(X);*

# Solution

d). *r3(X); r2(X); r1(X); w3(X); w1(X);*

The serialization graph is:




Not serializable.

RAMAIAH
Institute of Technology

# Thank you

# CI32/CY32 -DBS

# UNIT 5

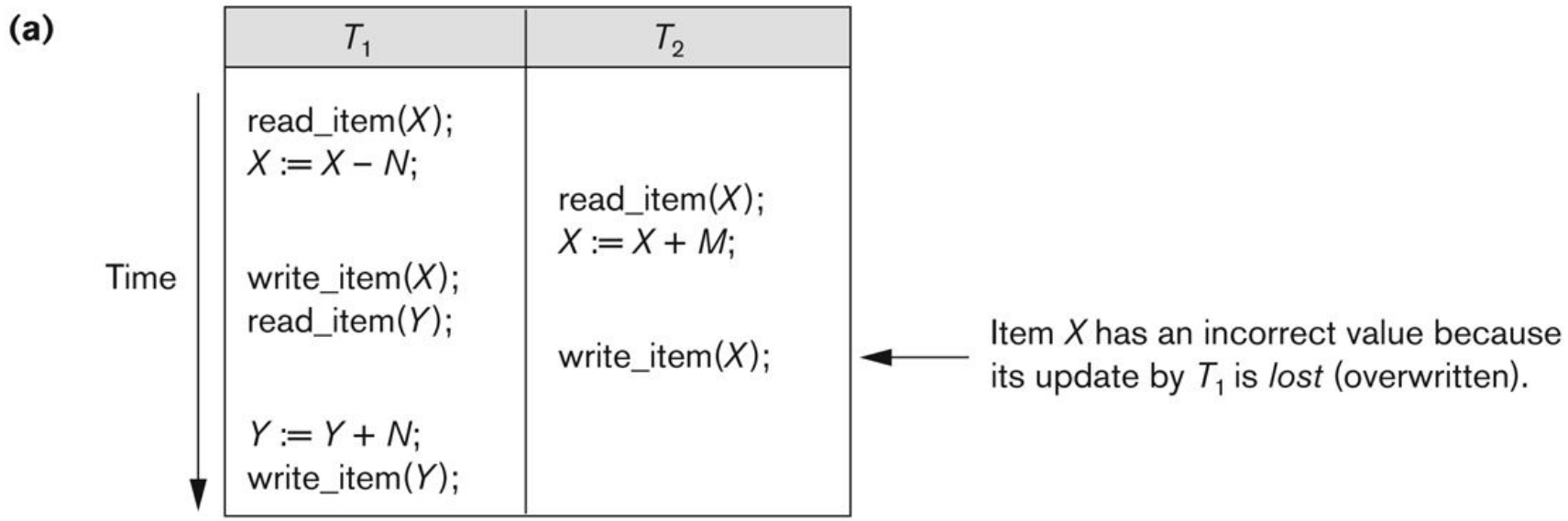
# Introduction to Transaction Processing

Why Concurrency Control is needed:

- **The Lost Update Problem**
  - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
- **The Temporary Update (or Dirty Read) Problem**
  - This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4).
  - The updated item is accessed by another transaction before it is changed back to its original value.
- **The Incorrect Summary Problem**
  - If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

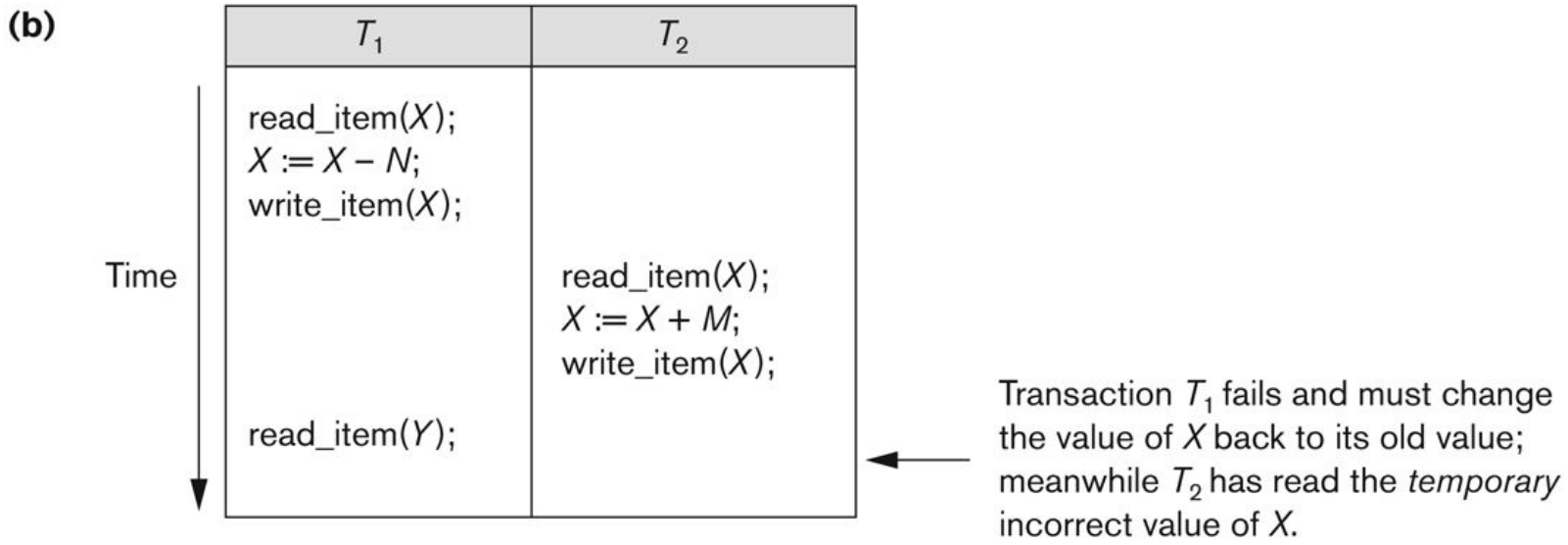
# Concurrent execution is uncontrolled: (a) The lost update problem.

**Figure 17.3**
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



# Concurrent execution is uncontrolled: (b) The temporary update problem.

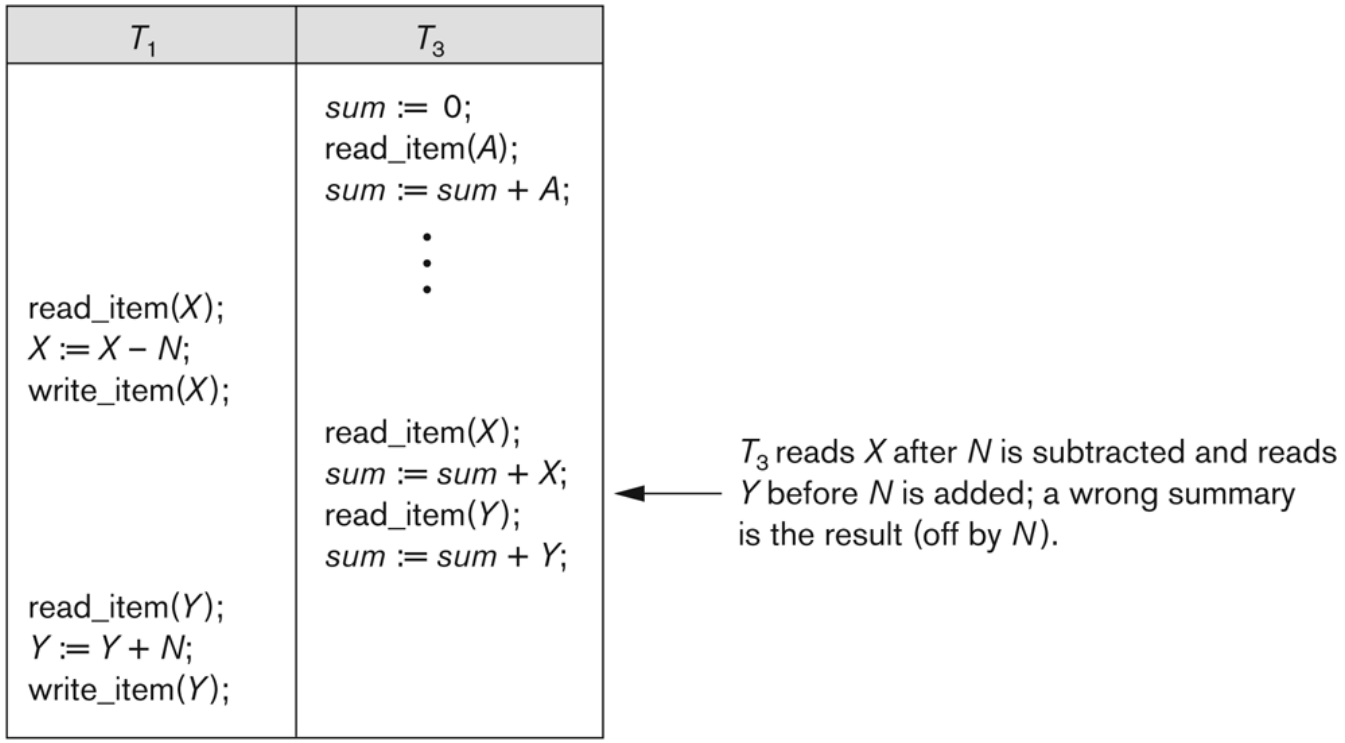
**Figure 17.3**
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



# Concurrent execution is uncontrolled: (c) The incorrect summary problem.

**Figure 17.3**
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

**(c)**



# Introduction to Transaction Processing

Why **recovery** is needed:

(What causes a Transaction to fail)

1. A computer failure (system crash):

   A **hardware or software error** occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. A transaction or system error:

   Some operation in the transaction may cause it to fail, such as **integer overflow or division by zero**. Transaction failure may also occur because of **erroneous parameter values or because of a logical programming error**. In addition, the **user may interrupt** the transaction during its execution.

# Introduction to Transaction Processing

Why **recovery** is needed (Contd.):

(What causes a Transaction to fail)

3. Local errors or exception conditions detected by the transaction:

   Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as **insufficient account balance in a banking database**, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

   A programmed abort in the transaction causes it to fail.

4. Concurrency control enforcement:

   The concurrency control method may decide to abort the transaction, to be restarted later, because it **violates serializability** or because several transactions are in a state of **deadlock** .

# Introduction to Transaction Processing

Why **recovery** is needed (contd.):

(What causes a Transaction to fail)

5. Disk failure:

   Some disk blocks may lose their data because of a **read or write malfunction** or because of **a disk read/write head crash**. This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes:

   This refers to an endless list of problems that includes **power or air-conditioning failure, fire, theft**, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

# Transaction and System Concepts

- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.
  - For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
- **Transaction states**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

# Transaction and System Concepts

- Recovery manager keeps track of the following operations:
  - **begin_transaction**: This marks the beginning of transaction execution.
  - **read** or **write**: These specify read or write operations on the database items that are executed as part of a transaction.
  - **end_transaction**: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
    - At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

# Transaction and System Concepts

- Recovery manager keeps track of the following operations (cont):
  - **commit_transaction**: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
  - **rollback** (or **abort**): This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

# State transition diagram illustrating the states for transaction execution

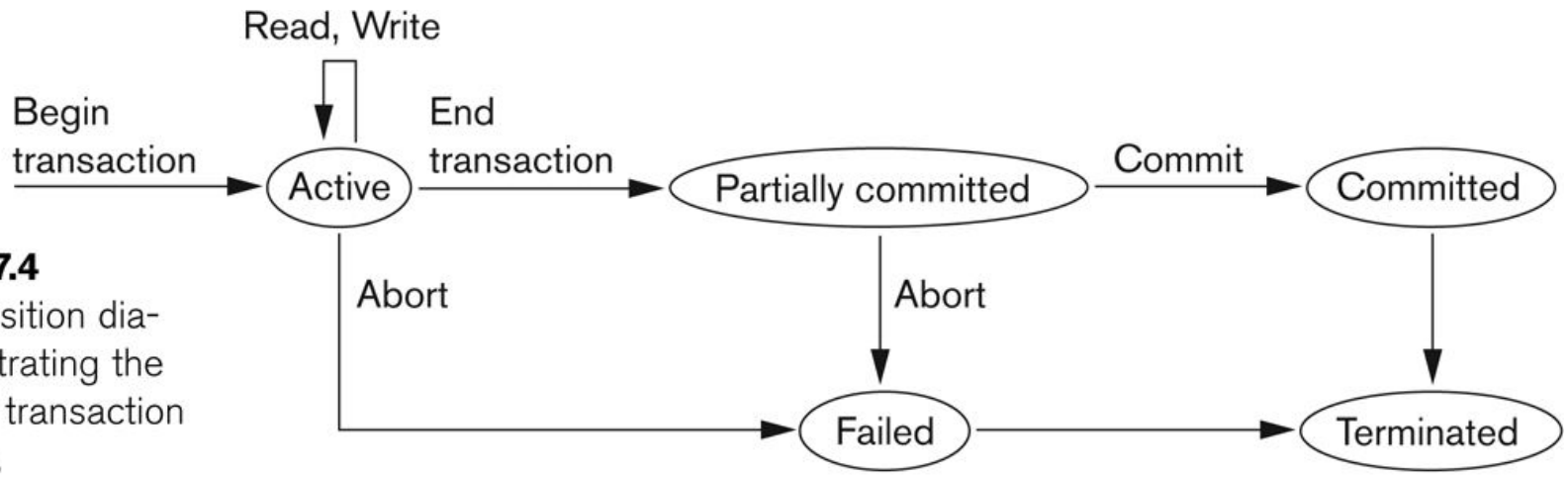



**Figure 17.4**
State transition diagram illustrating the states for transaction execution.

# Transaction and System Concepts

- The System Log
  - **Log** or **Journal**: The log keeps **track of all transaction operations that affect the values of database items.**
    - This information may be needed to permit recovery from transaction failures.
    - The **log is kept on disk**, so it is **not affected by any type of failure except for disk or catastrophic failure**.
    - In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

# Transaction and System Concepts

- The System Log (cont):
  - T in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:
  - Types of log record:
    - [start_transaction,T]: Records that transaction T has started execution.
    - [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
    - [read_item,T,X]: Records that transaction T has read the value of database item X.
    - [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
    - [abort,T]: Records that transaction T has been aborted.

# Characterizing Schedules based on Recoverability

- **Transaction schedule or history**:
  - When transactions are executing **concurrently in an interleaved fashion, the order of execution of operations** from the various transactions forms what is known as a transaction schedule (or history).
- A **schedule** (or **history**) S of n transactions T1, T2, …, Tn:
  - It is an **ordering of the operations** of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.
  - Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

# Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

- **Recoverable schedule**:
  - One where no transaction needs to be rolled back.
  - A schedule S is recoverable **if no transaction T in S commits** until all transactions T' that have written an item that T reads have committed.
- **Cascadeless schedule**:
  - One where **every transaction reads only the items** that are **written by committed transactions.**

# Characterizing Schedules based on Recoverability

Schedules classified on recoverability (contd.):

- **Schedules requiring cascaded rollback**:
  - A schedule in which **uncommitted transactions that read an item from a failed transaction** must be rolled back.
- **Strict Schedules**:
  - A schedule in which a **transaction can neither read or write an item X until the last transaction that wrote X has committed**.

# Characterizing Schedules based on Serializability

- Serial schedule:
  - A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
    - Otherwise, the schedule is called nonserial schedule.
- Serializable schedule:
  - A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

# Characterizing Schedules based on Serializability

- Result equivalent:
  - Two schedules are called result equivalent if they produce the same final state of the database.
- Conflict equivalent:
  - Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- Conflict serializable:
  - A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S’.

# Characterizing Schedules based on Recoverability

- **Transaction schedule or history**:
  - When transactions are executing **concurrently in an interleaved fashion, the order of execution of operations** from the various transactions forms what is known as a transaction schedule (or history).
- A **schedule** (or **history**) S of n transactions T1, T2, …, Tn:
  - It is an **ordering of the operations** of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.
  - Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

# Isolation levels in Transaction Processing

- A transaction isolation level is defined by the following phenomena –
- **Dirty Read –** A Dirty read is the situation when a **transaction reads a data that has not yet been committed.**
- **Non Repeatable read –** Non Repeatable read occurs when a **transaction reads same row twice, and get a different value each time.**
- **Phantom Read –** Phantom Read occurs **when two same queries are executed, but the rows retrieved by the two, are different.**

Based on these phenomena, The SQL standard defines four isolation levels :

# Isolation levels in Transaction Processing

- **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.

- **Read Committed –** This isolation level guarantees that **any data read is committed at the moment it is read**. Thus it does not allows dirty read. The transaction **holds a read or write lock on the current row**, and thus prevent other transactions from reading, updating or deleting it.

# Isolation levels in Transaction Processing

- **Repeatable Read –** This is the most restrictive isolation level. The transaction holds **read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes**. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.

- **Serializable – This is the Highest isolation level. A *serializable* execution is guaranteed to be serializable**. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

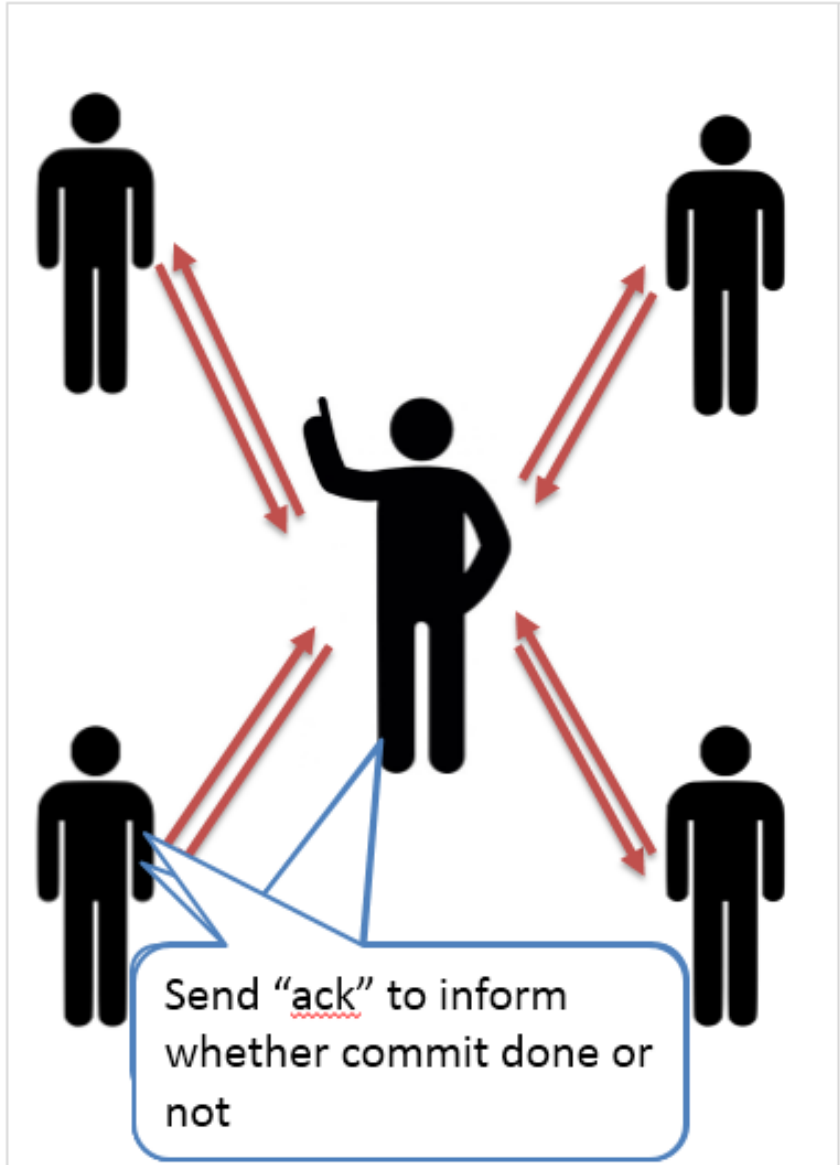
# Two phase commit protocol

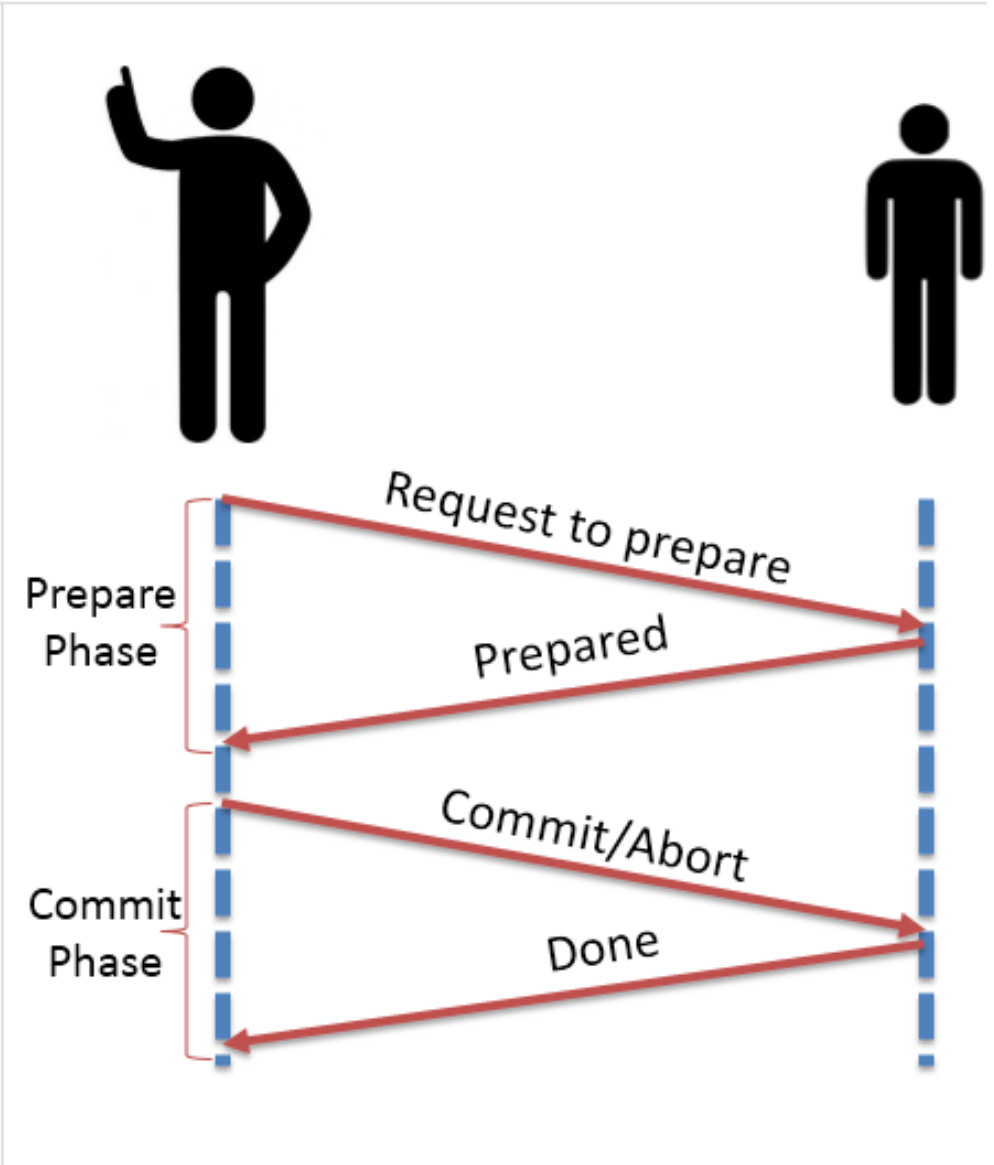
- Two phase commit protocol **ensures that all participants perform the same action** (either to commit or to rollback a transaction).
- It is designed to ensure that either all the databases are updated or none of them, so that the databases remain synchronized.
- In two phase commit protocol there is one node which is act as a coordinator or controlling site and all other participating node are known as cohorts or participant or slave.
- Coordinator (controlling site) – the component that coordinates with all the participants.
- Cohorts (Participants/Slaves) – each individual node except coordinator are participant.

# Two phase commit protocol

- As the name suggests, the two phase commit protocol involves two phases.
  1. Commit request phase OR Prepare phase
  2. Commit/Abort phase

# Two phase commit protocol



# Two phase commit protocol

1. Commit Request Phase (Obtaining Decision)
   - **After each slave has locally completed its transaction**, it **sends a "DONE" message** to the controlling site.
   - When the **controlling site has received "DONE" message** from all slaves, it **sends a "Prepare" (prepare to commit) message** to the slaves.
   - The **slaves vote** on whether they still **want to commit or not**.
   - **If a slave wants to commit**, it **sends a "Ready" message**.
   - A **slave that does not want to commit sends a "Not Ready" message**.
   - This may happen when the slave has conflicting concurrent transactions or there is a timeout.

# Two phase commit protocol

2. Commit Phase (Performing Decision)

   1) After the **controlling site has received "Ready" message** from all the slaves:

      - The controlling site **sends a "Global Commit" message** to the slaves.
      - The **slaves commit** the transaction and **send a "Commit ACK"** message to the controlling site.
      - When the **controlling site receives "Commit ACK"** message **from all the slaves**, it considers the **transaction as committed**.

# Two phase commit protocol

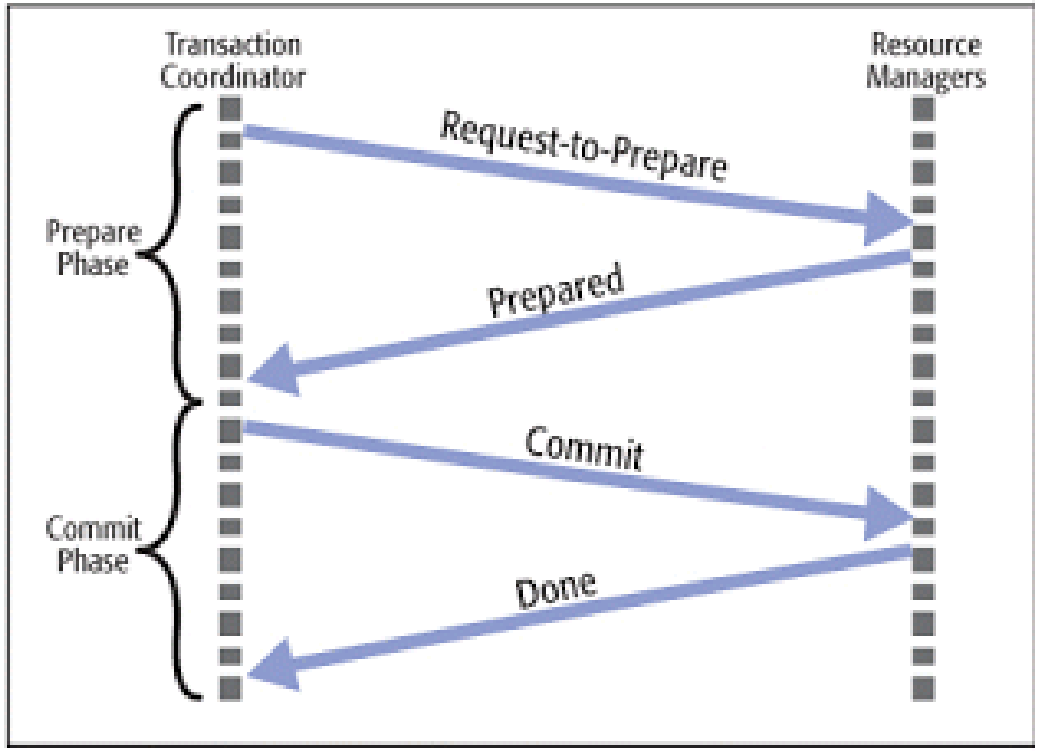
2. Commit Phase (Performing Decision)

   2) After the **controlling site has received the first "Not Ready" message** from any slave:

      - The **controlling site sends a "Global Abort" message** to the slaves.
      - The **slaves abort** the transaction and **send a "Abort ACK"** message to the controlling site.
      - When the **controlling site receives "Abort ACK"** message **from all the slaves**, it considers the **transaction as aborted**.

# Two phase commit protocol

- Atomic commitment protocol
- Distributed algorithm that coordinates all process and participate in a distributed atomic transaction, whether to commit or abort T



# Limitation

- Block Problem:

  - Blocking reduces the availability of system since blocked transactions keep all the resources until they receive final command from coordinator.

# Database recovery

- There are many situations in which a **transaction may not reach a commit or abort point**.
  - **Operating system crash**
  - **DBMS crash**
  - **System might lose power (power failure)**
  - **Disk may fail** or other **hardware may fail (disk/hardware failure)**
  - **Human error**
- In any of above situations, data in the database may become inconsistent or lost.

# Database recovery

- For example, if a transaction has completed 30 out of 40 write instructions to the database when the DBMS crashes, then the database may be in an inconsistent state as only part of the transaction's work was completed.
- Database recovery is the **process of restoring the database and the data to a consistent state**.
- This may include **restoring lost data up to the point of the event** (e.g. system crash).

RAMAIAH
Institute of Technology

# Characterizing Schedules based on Recoverability

**Transaction schedule or history**:

- When transactions are executing **concurrently in an interleaved fashion, the order of execution of operations** from the various transactions forms what is known as a transaction schedule (or history).

A **schedule** (or **history**) S of n transactions T1, T2, …, Tn:

- It is an **ordering of the operations** of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.
- Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

RAMAIAH
Institute of Technology

# Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

**Recoverable schedule**:

- One where no transaction needs to be rolled back.
- A schedule S is recoverable **if no transaction T in S commits** until all transactions T' that have written an item that T reads have committed.

**Cascadeless schedule**:

- One where **every transaction reads only the items** that are **written by committed transactions.**

RAMAIAH Institute of Technology

# Characterizing Schedules based on Recoverability

## Schedules classified on recoverability (contd.):

**Schedules requiring cascaded rollback**:

- A schedule in which **uncommitted transactions that read an item from a failed transaction** must be rolled back.

**Strict Schedules**:

- A schedule in which a **transaction can neither read or write an item X until the last transaction that wrote X has committed**.

# Characterizing Schedules based on Serializability

Serial schedule:

- A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
  - Otherwise, the schedule is called nonserial schedule.

Serializable schedule:

- A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

RAMAIAH Institute of Technology

# Characterizing Schedules based on Serializability

Result equivalent:

- Two schedules are called result equivalent if they produce the same final state of the database.

Conflict equivalent:

- Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

Conflict serializable:

- A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.

# Characterizing Schedules based on Recoverability

**Transaction schedule or history**:

- When transactions are executing **concurrently in an interleaved fashion, the order of execution of operations** from the various transactions forms what is known as a transaction schedule (or history).

A **schedule** (or **history**) S of n transactions T1, T2, …, Tn:

- It is an **ordering of the operations** of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.
- Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

RAMAIAH Institute of Technology

# Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

**Recoverable schedule**:

- One where no transaction needs to be rolled back.
- A schedule S is recoverable **if no transaction T in S commits** until all transactions T' that have written an item that T reads have committed.

**Cascadeless schedule**:

- One where **every transaction reads only the items** that are **written by committed transactions.**

RAMAIAH Institute of Technology

# Characterizing Schedules based on Recoverability

Schedules classified on recoverability (contd.):

**Schedules requiring cascaded rollback**:

- A schedule in which **uncommitted transactions that read an item from a failed transaction** must be rolled back.

**Strict Schedules**:

- A schedule in which a **transaction can neither read or write an item X until the last transaction that wrote X has committed**.

RAMAIAH Institute of Technology

# Characterizing Schedules based on Serializability

Serial schedule:

- A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
  - Otherwise, the schedule is called nonserial schedule.

Serializable schedule:

- A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

RAMAIAH Institute of Technology

# Characterizing Schedules based on Serializability

Result equivalent:

- Two schedules are called result equivalent if they produce the same final state of the database.

Conflict equivalent:

- Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

Conflict serializable:

- A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S’.

RAMAIAH Institute of Technology

# Isolation levels in Transaction Processing

A transaction isolation level is defined by the following phenomena –

**Dirty Read –** A Dirty read is the situation when a transaction reads a data that has not yet been committed.

**Non Repeatable read –** Non Repeatable read occurs when a transaction reads same row twice, and get a different value each time.

**Phantom Read –** Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different.

Based on these phenomena, The SQL standard defines four isolation levels :

RAMAIAH
Institute of Technology

# Isolation levels in Transaction Processing

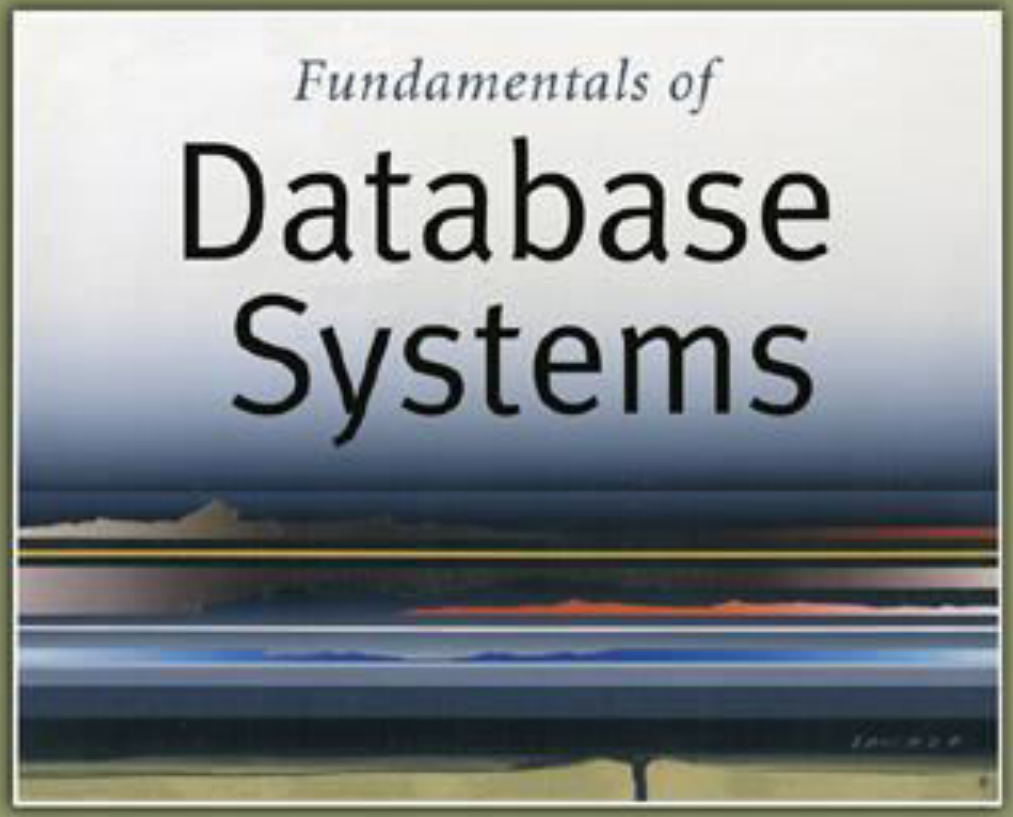
**Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.

**Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.

# Isolation levels in Transaction Processing

**Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.

**Serializable –** This is the Highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

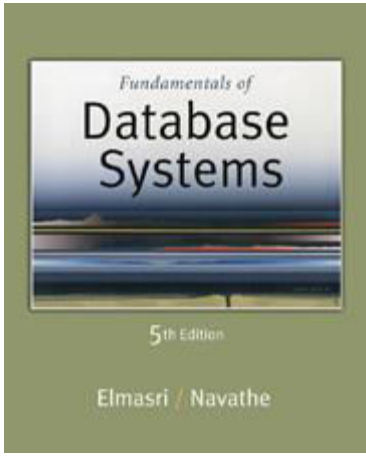



5th Edition

Elmasri / Navathe

# Chapter 18

## Concurrency Control Techniques



# Chapter 18 Outline

- Databases Concurrency Control
  1. Purpose of Concurrency Control
  2. Two-Phase locking
  3. Limitations of CCMs
  4. Index Locking
  5. Lock Compatibility Matrix
  6. Lock Granularity

# Database Concurrency Control

- 1 Purpose of Concurrency Control
  - To enforce Isolation (through mutual exclusion) among conflicting transactions.
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts.

- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

# Database Concurrency Control

## Two-Phase Locking Techniques

- Locking is an operation which secures
  - (a) permission to Read
  - (b) permission to Write a data item for a transaction.
- Example:
  - Lock (X). Data item X is locked in behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
  - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- Two locks modes:
  - (a) shared (read)  (b) exclusive (write).
- Shared mode:  shared lock (X)
  - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- Exclusive mode: Write lock (X)
  - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.
- Conflict matrix

| | Read | Write |
|---|---|---|
| Read | Y | N |
| Write | N | N |

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- Lock Manager:
  - Managing locks on data items.
- Lock table:
  - Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

| Transaction ID | Data item id | lock mode | Ptr to next data item |
|---|---|---|---|
| T1 | X1 | Read | Next |

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the lock operation:

```
B: if LOCK (X) = 0 (*item is unlocked*)
   then LOCK (X) ← 1 (*lock the item*)
   else begin
     wait (until lock (X) = 0) and
     the lock manager wakes up the transaction);
   goto B
   end;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the unlock operation:

LOCK (X) ← 0 (*unlock the item*)
if any transactions are waiting then
  wake up one of the waiting the transactions;

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the read operation:

```
B: if LOCK (X) = "unlocked" then
  begin LOCK (X) ← "read-locked";
      no_of_reads (X) ← 1;
  end
  else if LOCK (X) ← "read-locked" then
           no_of_reads (X) ← no_of_reads (X) +1
        else begin wait (until LOCK (X) = "unlocked" and
           the lock manager wakes up the transaction);
           go to B
         end;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the write lock operation:

```
B: if LOCK (X) = "unlocked" then
  begin LOCK (X) ← "read-locked";
      no_of_reads (X) ← 1;
  end
  else if LOCK (X) ← "read-locked" then
           no_of_reads (X) ← no_of_reads (X) +1
        else begin wait (until LOCK (X) = "unlocked" and
           the lock manager wakes up the transaction);
           go to B
         end;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the unlock operation:

```
if LOCK (X) = "write-locked" then
  begin LOCK (X) ← "unlocked";
        wakes up one of the transactions, if any
  end
  else if LOCK (X) ← "read-locked" then
      begin
           no_of_reads (X) ← no_of_reads (X) -1
           if  no_of_reads (X) = 0 then
           begin
         LOCK (X) = "unlocked";
        wake up one of the transactions, if any
           end
      end;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- Lock conversion
  - Lock upgrade: existing read lock to write lock

    if Ti has a read-lock (X) and Tj has no read-lock (X) ($i \neq j$) then
     convert read-lock (X) to write-lock (X)
    else
     force Ti to wait until Tj unlocks X

  - Lock downgrade: existing write lock to read lock

    Ti has a write-lock (X)  (*no transaction can have any lock on X*)
    convert write-lock (X) to read-lock (X)

# Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

- Two Phases:
  - (a) Locking (Growing)
  - (b) Unlocking (Shrinking).
- **Locking (Growing) Phase:**
  - A transaction applies locks (read or write) on desired data items one at a time.
- **Unlocking (Shrinking) Phase:**
  - A transaction unlocks its locked data items one at a time.
- **Requirement:**
  - For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

# Database Concurrency Control

## Two-Phase Locking Techniques: The algorithm

| T1 | T2 | Result |
|---|---|---|
| read_lock (Y); | read_lock (X); | Initial values: X=20; Y=30 |
| read_item (Y); | read_item (X); | Result of serial execution |
| unlock (Y); | unlock (X); | T1 followed by T2 |
| write_lock (X); | Write_lock (Y); | X=50, Y=80. |
| read_item (X); | read_item (Y); | Result of serial execution |
| X:=X+Y; | Y:=X+Y; | T2 followed by T1 |
| write_item (X); | write_item (Y); | X=70, Y=50 |
| unlock (X); | unlock (Y); | |

# Database Concurrency Control

## Two-Phase Locking Techniques: The algorithm

| T1 | T2 |
|---|---|
| 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); | |

Time

**Result**

X=50; Y=50
Nonserializable because it.
violated two-phase policy.

# Database Concurrency Control

## Two-Phase Locking Techniques: The algorithm

| T'1 | T'2 | |
|---|---|---|
| read_lock (Y); | read_lock (X); | T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with. |
| 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); | |

# Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

- Two-phase policy generates two locking algorithms
  - (a) **Basic**
  - (b) **Conservative**
- **Conservative**:
  - Prevents deadlock by locking all desired data items before transaction begins execution.
- **Basic**:
  - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Strict**:
  - A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

# Database Concurrency Control

## Dealing with Deadlock and Starvation

- **Deadlock**

| T'1 | T'2 | |
|---|---|---|
| read_lock (Y); | | T1 and T2 did follow two-phase policy but they are deadlock |
| read_item (Y); | | |
| | read_lock (X); | |
| | read_item (Y); | |
| write_lock (X); | | |
| (waits for X) | | |
| | write_lock (Y); | |
| | (waits for Y) | |

- Deadlock (T'1 and T'2)

# Database Concurrency Control

## Dealing with Deadlock and Starvation

- **Deadlock detection and resolution**
  - In this approach, deadlocks are allowed to happen. The scheduler maintains **a wait-for-graph for detecting cycle.** If a cycle exists, then one transaction involved in the cycle is **selected (victim) and rolled-back**.
  - A wait-for-graph is created using the **lock table**. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle.

# Database Concurrency Control

## Validation (Optimistic) Concurrency Control Schemes

- In this technique only at the time of commit, serializability is checked and transactions are aborted in case of non-serializable schedules.
- Three phases:
  1. **Read phase**
  2. **Validation phase**
  3. **Write phase**

1. **Read phase**:
   - A transaction can read values of committed data items.

# Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

2. **Validation phase**: Serializability is checked before transactions write their updates to the database.
   - This phase for Ti checks that, for each transaction Tj that is either committed or is in its validation phase, one of the following conditions holds:
     - Tj completes its write phase before Ti starts its read phase.
     - Ti starts its write phase after Tj completes its write phase
     - Both the read_set and write_set of Ti have no items in common with the write_set of Tj, and Tj completes its read phase.

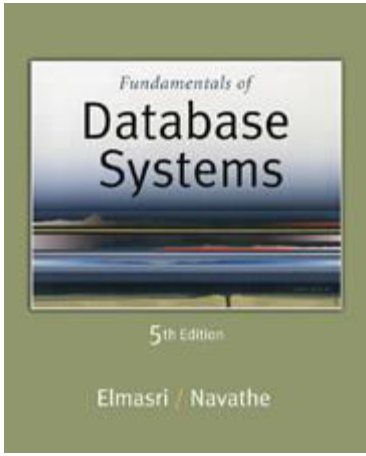
# Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

3. **Write phase**: On a successful validation transactions' updates are applied to the database; otherwise, transactions are restarted.

# Chapter 17

## Introduction to Transaction Processing Concepts and Theory



# Chapter Outline

1 Introduction to Transaction Processing

2 Transaction and System Concepts

3 Desirable Properties of Transactions

4 Characterizing Schedules based on Recoverability

5 Characterizing Schedules based on Serializability

6 Transaction Support in SQL

# 1 Introduction to Transaction Processing (1)

- **Single-User System**:
  - At most one user at a time can use the system.
- **Multiuser System**:
  - Many users can access the system concurrently.
- **Concurrency**
  - **Interleaved processing**:
    - Concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing**:
    - Processes are concurrently executed in multiple CPUs.

# Introduction to Transaction Processing (2)

- A **Transaction**:
  - Logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).
- A transaction (set of operations) may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.
- **Transaction boundaries**:
  - Begin and End transaction.
- An **application program** may contain several transactions separated by the Begin and End transaction boundaries.

# Introduction to Transaction Processing (3)

SIMPLE MODEL OF A DATABASE (for purposes of discussing transactions):

- **A database** is a collection of named data items
- **Granularity** of data - a field, a record , or a whole disk block (Concepts are independent of granularity)
- Basic operations are **read** and **write**
  - **read_item(X)**: Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
  - **write_item(X)**: Writes the value of program variable X into the database item named X.

# Introduction to Transaction Processing (4)

READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.
- read_item(X) command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the buffer to the program variable named X.

# Introduction to Transaction Processing (5)

READ AND WRITE OPERATIONS (contd.):

- **write_item(X)** command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the program variable named X into its correct location in the buffer.
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time).

# Two sample transactions

- FIGURE 17.2 Two sample transactions:
  - (a) Transaction T1
  - (b) Transaction T2

(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);
```

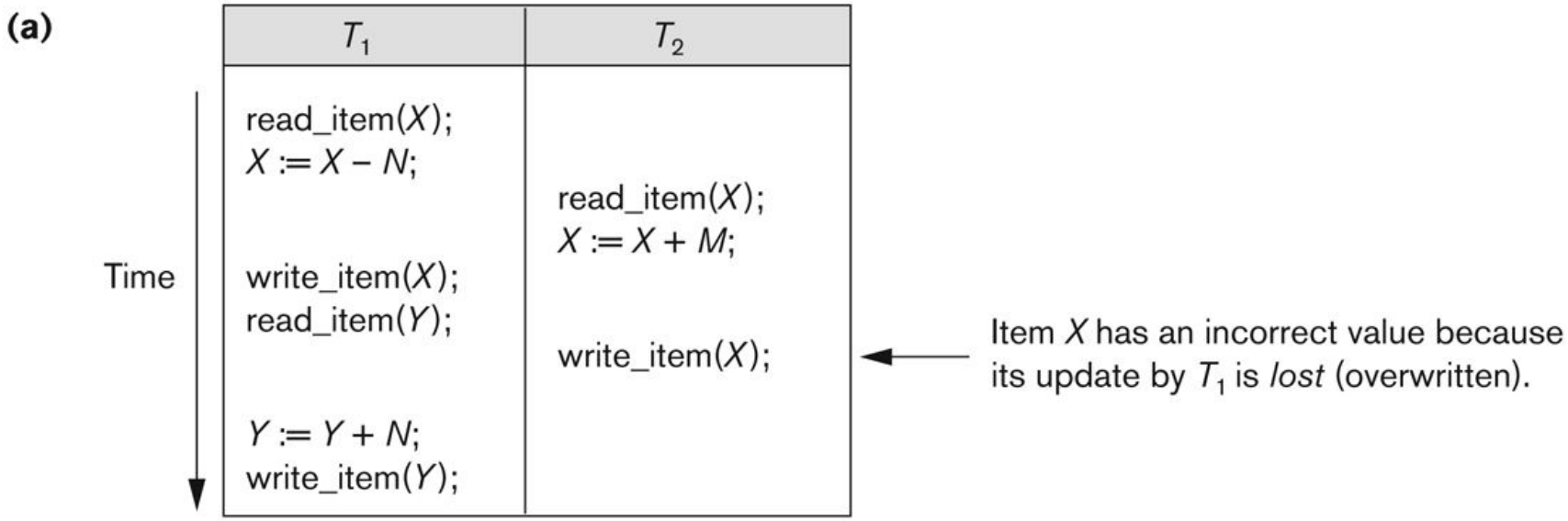
# Introduction to Transaction Processing (6)

Why Concurrency Control is needed:

- **The Lost Update Problem**
  - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
- **The Temporary Update (or Dirty Read) Problem**
  - This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4).
  - The updated item is accessed by another transaction before it is changed back to its original value.
- **The Incorrect Summary Problem**
  - If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

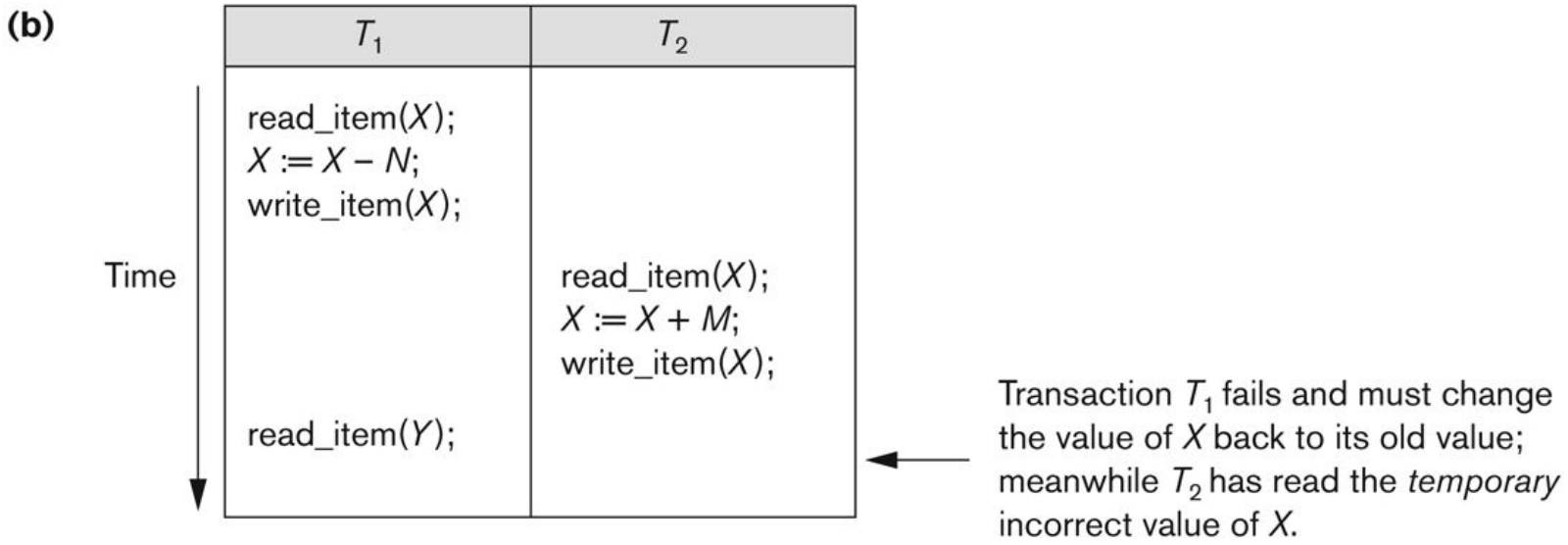
# Concurrent execution is uncontrolled: (a) The lost update problem.

**Figure 17.3**
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



# Concurrent execution is uncontrolled: (b) The temporary update problem.

**Figure 17.3**
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



# Concurrent execution is uncontrolled: (c) The incorrect summary problem.

**Figure 17.3**
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

**(c)**

| $T_1$ | $T_3$ |
|---|---|
| | $sum := 0;$<br>read_item($A$);<br>$sum := sum + A;$<br>• • • |
| read_item($X$);<br>$X := X - N;$<br>write_item($X$); | |
| | read_item($X$);<br>$sum := sum + X;$<br>read_item($Y$);<br>$sum := sum + Y;$ |
| read_item($Y$);<br>$Y := Y + N;$<br>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 $N$).

# Introduction to Transaction Processing

Why **recovery** is needed:

(What causes a Transaction to fail)

1. A computer failure (system crash):

   A **hardware or software error** occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. A transaction or system error:

   Some operation in the transaction may cause it to fail, such as **integer overflow or division by zero**. Transaction failure may also occur because of **erroneous parameter values or because of a logical programming error**. In addition, the **user may interrupt** the transaction during its execution.

# Introduction to Transaction Processing

Why **recovery** is needed (Contd.):

(What causes a Transaction to fail)

3. Local errors or exception conditions detected by the transaction:

   Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as **insufficient account balance in a banking database**, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

   A programmed abort in the transaction causes it to fail.

4. Concurrency control enforcement:

   The concurrency control method may decide to abort the transaction, to be restarted later, because it **violates serializability** or because several transactions are in a state of **deadlock** .

# Introduction to Transaction Processing

Why **recovery** is needed (contd.):

(What causes a Transaction to fail)

5. Disk failure:

   Some disk blocks may lose their data because of a **read or write malfunction** or because of **a disk read/write head crash**. This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes:

   This refers to an endless list of problems that includes **power or air-conditioning failure, fire, theft**, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

# Transaction and System Concepts

- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.
  - For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
- **Transaction states**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

# Transaction and System Concepts

- Recovery manager keeps track of the following operations:
  - **begin_transaction**: This marks the beginning of transaction execution.
  - **read** or **write**: These specify read or write operations on the database items that are executed as part of a transaction.
  - **end_transaction**: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
    - At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

# Transaction and System Concepts

- Recovery manager keeps track of the following operations (cont):
  - **commit_transaction**: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
  - **rollback** (or **abort**): This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

# Transaction and System Concepts (4)

- Recovery techniques use the following operators:
  - **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
  - **redo**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

# State transition diagram illustrating the states for transaction execution

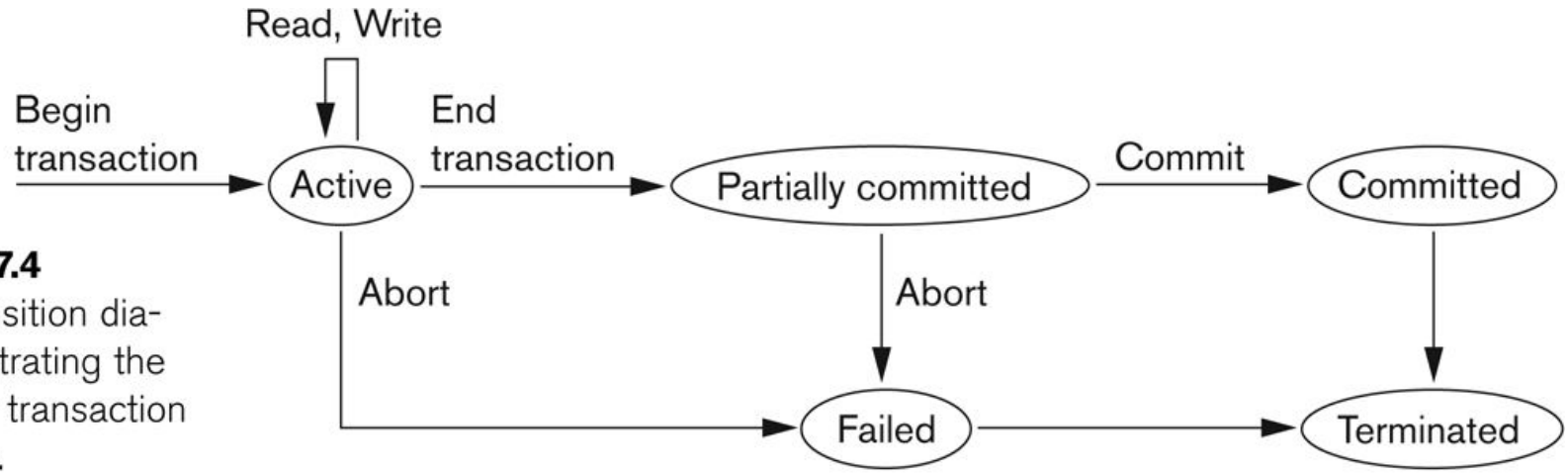



**Figure 17.4**
State transition diagram illustrating the states for transaction execution.

# Transaction and System Concepts

- The System Log
  - **Log** or **Journal**: The log keeps **track of all transaction operations that affect the values of database items.**
    - This information may be needed to permit recovery from transaction failures.
    - The **log is kept on disk**, so it is **not affected by any type of failure except for disk or catastrophic failure**.
    - In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

# Transaction and System Concepts

- The System Log (cont):
  - T in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:
  - Types of log record:
    - [start_transaction,T]: Records that transaction T has started execution.
    - [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
    - [read_item,T,X]: Records that transaction T has read the value of database item X.
    - [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
    - [abort,T]: Records that transaction T has been aborted.

# Transaction and System Concepts

Recovery using log records:

- If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques described in Chapter 19.
  1. Because the **log contains a record of every write operation that changes the value of some database item**, it is possible to **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.
  2. We can also **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

# Transaction and System Concepts

## Commit Point of a Transaction:

- **Definition a Commit Point:**
  - A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log.
  - Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database.
  - The transaction then writes an entry [commit,T] into the log.
- **Roll Back of transactions:**
  - Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

# 3 Desirable Properties of Transactions

ACID properties:

- **Atomicity**: A transaction is an atomic unit of processing; **it is either performed in its entirety or not performed at all**. Also called as 'All or Nothing Rule'.
- **Consistency preservation**: A **correct execution** of the transaction must take the database from one consistent state to another.

  -DB must be consistent before and after transaction.
- **Isolation**: A transaction **should not make its updates visible to other transactions until it is committed**; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary.

  -Multiple transactions occur independently without interference
- **Durability or permanency**: Once a transaction changes the database and the **changes are committed, these changes must never be lost because of subsequent failure.**

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.
If the transaction fails after completion of **T1** but before completion of **T2**.( say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

| **Before:** X : 500 | Y: 200 |
|---|---|
| Transaction T | |
| **T1** | **T2** |
| Read (X)<br>X: = X − 100<br>Write (X) | Read (Y)<br>Y: = Y + 100<br>Write (Y) |
| **After:** X : 400 | Y : 300 |

**Consistency**

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,
The total amount before and after the transaction must be maintained.
Total **before T** occurs = **500 + 200 = 700**.
Total **after T occurs** = **400 + 300 = 700**.
Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result T is incomplete.

# Characterizing Schedules based on Recoverability

- **Transaction schedule or history**:
  - When transactions are executing **concurrently in an interleaved fashion, the order of execution of operations** from the various transactions forms what is known as a transaction schedule (or history).
- A **schedule** (or **history**) S of n transactions T1, T2, …, Tn:
  - It is an **ordering of the operations** of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.
  - Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

# Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

- **Recoverable schedule**:
  - One where no transaction needs to be rolled back.
  - A schedule S is recoverable **if no transaction T in S commits** until all transactions T' that have written an item that T reads have committed.
- **Cascadeless schedule**:
  - One where **every transaction reads only the items** that are **written by committed transactions.**

# Characterizing Schedules based on Recoverability

Schedules classified on recoverability (contd.):

- **Schedules requiring cascaded rollback**:
  - A schedule in which **uncommitted transactions that read an item from a failed transaction** must be rolled back.
- **Strict Schedules**:
  - A schedule in which a **transaction can neither read or write an item X until the last transaction that wrote X has committed**.

# Characterizing Schedules based on Serializability

- Serial schedule:
  - A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
    - Otherwise, the schedule is called nonserial schedule.
- Serializable schedule:
  - A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

# Characterizing Schedules based on Serializability

- Result equivalent:
  - Two schedules are called result equivalent if they produce the same final state of the database.
- Conflict equivalent:
  - Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- Conflict serializable:
  - A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.

# Characterizing Schedules Based on Serializability (cont'd.)

- Testing for serializability of a schedule

1. For each transaction $T_i$ participating in schedule $S$, create a node labeled $T_i$ in the precedence graph.
2. For each case in $S$ where $T_j$ executes a read_item($X$) after $T_i$ executes a write_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in $S$ where $T_j$ executes a write_item($X$) after $T_i$ executes a read_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in $S$ where $T_j$ executes a write_item($X$) after $T_i$ executes a write_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule $S$ is serializable if and only if the precedence graph has no cycles.

Algorithm 20.1 Testing conflict serializability of a schedule S

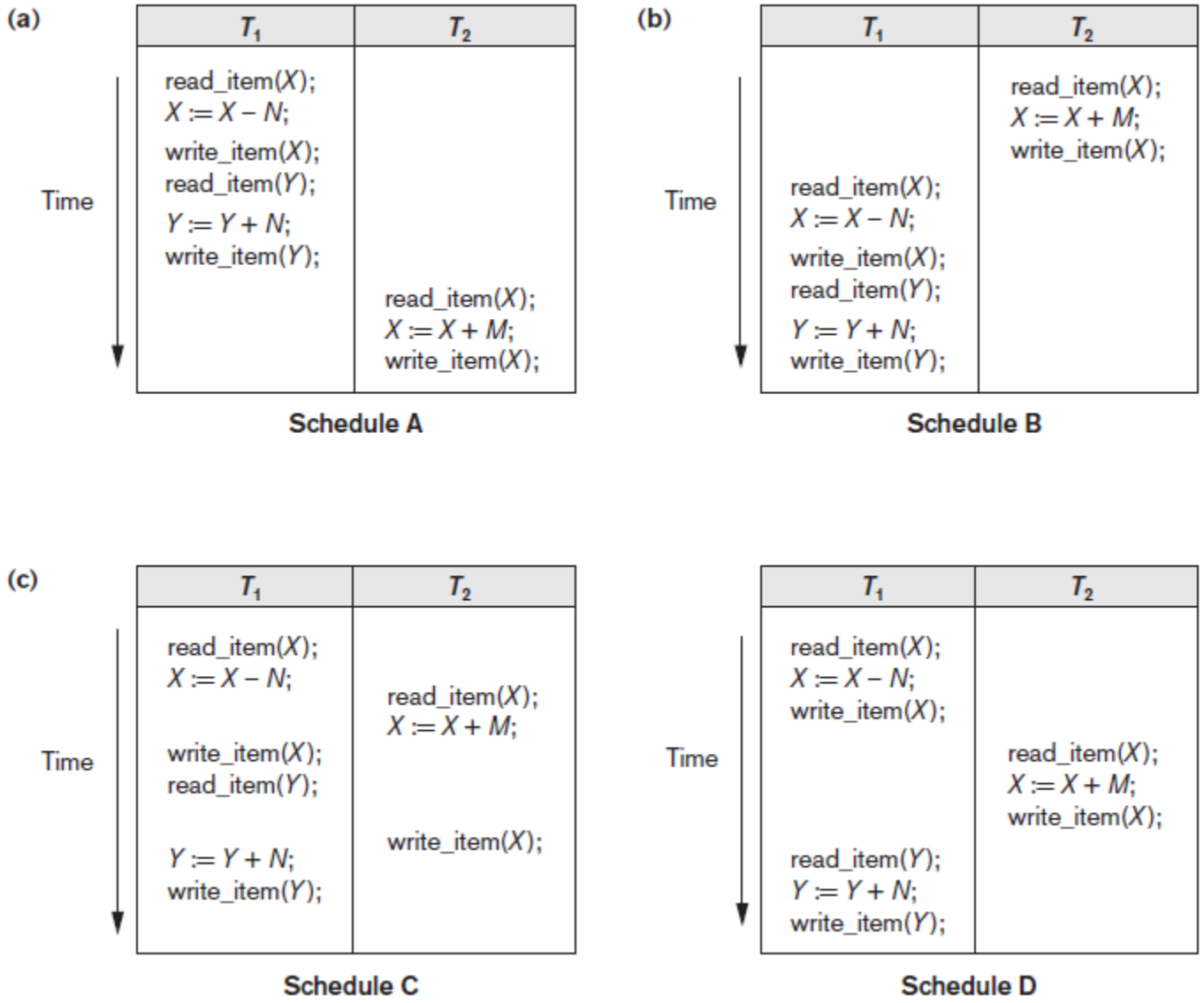
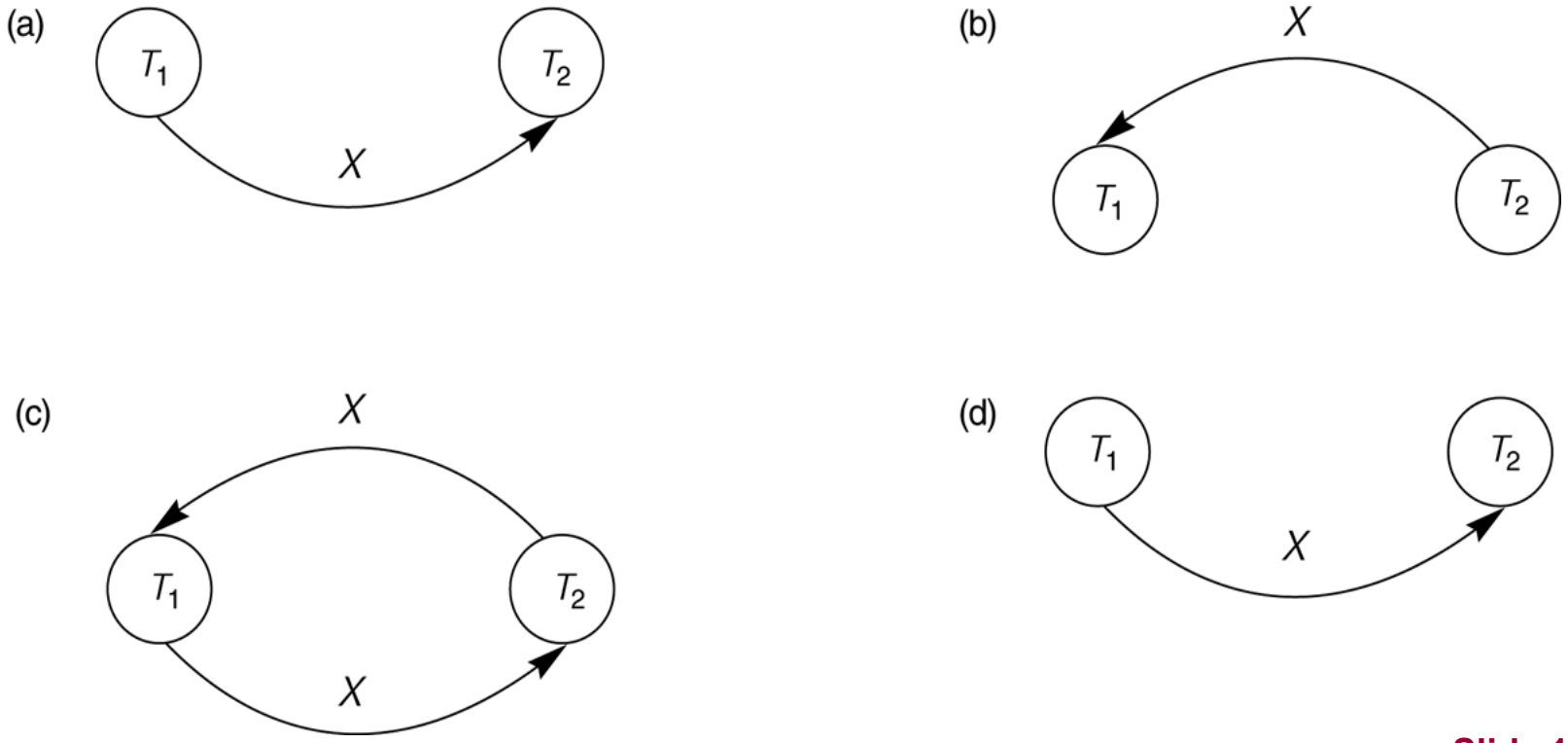



Figure 20.5 Examples of serial and nonserial schedules involving transactions *T*1 and *T*2 (a) Serial schedule A: *T*1 followed by *T*2 (b) Serial schedule B: *T*2 followed by *T*1 (c) Two nonserial schedules C and D with interleaving of operations

# Constructing the Precedence Graphs

- FIGURE 17.7 Constructing the precedence graphs for schedules A and D from Figure 17.5 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).



# Another example of serializability Testing

**Figure 17.8**
Another example of serializability testing. (a) The read and write operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule E. (c) Schedule *F*.

**(a)**

| Transaction $T_1$ |
|---|
| read_item($X$); |
| write_item($X$); |
| read_item($Y$); |
| write_item($Y$); |

| Transaction $T_2$ |
|---|
| read_item($Z$); |
| read_item($Y$); |
| write_item($Y$); |
| read_item($X$); |
| write_item($X$); |

| Transaction $T_3$ |
|---|
| read_item($Y$); |
| read_item($Z$); |
| write_item($Y$); |
| write_item($Z$); |

# Another Example of Serializability Testing

**Figure 17.8**
Another example of serializability testing. (a) The read and write operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule E. (c) Schedule *F*.

**(b)**

Time

| 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($X$); | |

**Schedule E**

# Another Example of Serializability Testing

**Figure 17.8**
Another example of serializability testing. (a) The read and write operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule E. (c) Schedule *F*.

**(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**

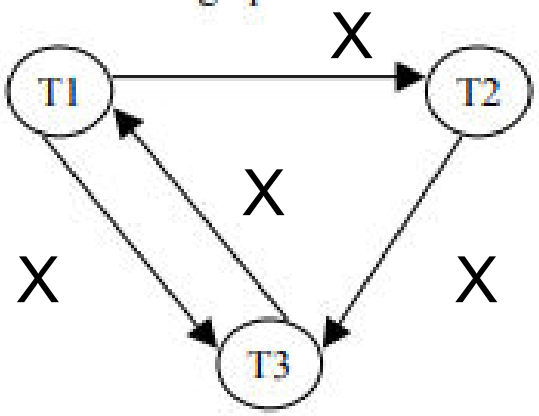
# Which of the following Schedules is conflict serializable? For each serializable schedule , determine equivalent serial schedules

- a). r1(X); r3(X); w1(X); r2(X); w3(X);
- b). r1(X); r3(X); w3(X); w1(X); r2(X);
- c). r3(X); r2(X); w3(X); r1(X); w1(X);
- d). r3(X); r2(X); r1(X); w3(X); w1(X);

# Solution

a). *r1(X); r3(X); w1(X); r2(X); w3(X);*

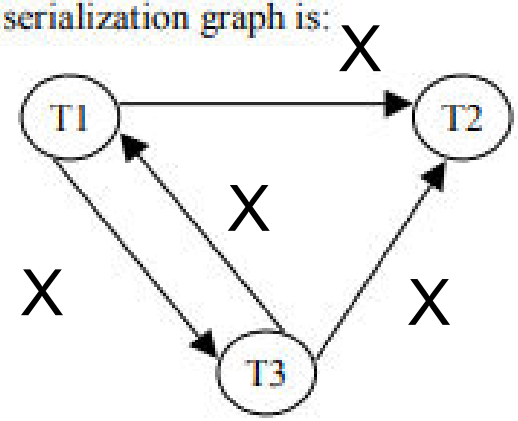
The serialization graph is:




Not serializable.

b). *r1(X); r3(X); w3(X); w1(X); r2(X);*

The serialization graph is:

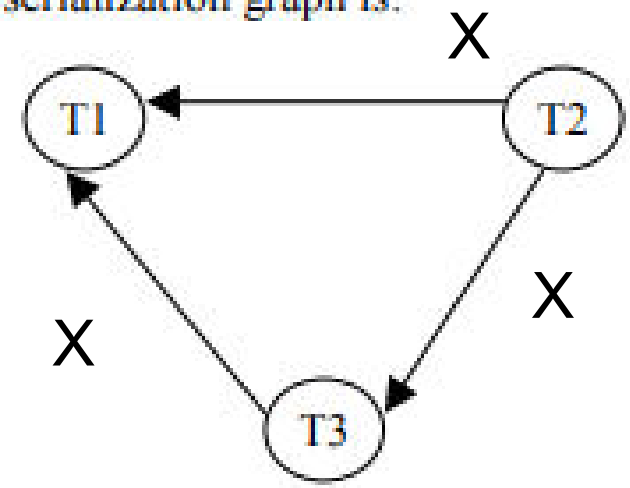



Not serializable.

# Solution

c). *r3(X); r2(X); w3(X); r1(X); w1(X);*

The serialization graph is:



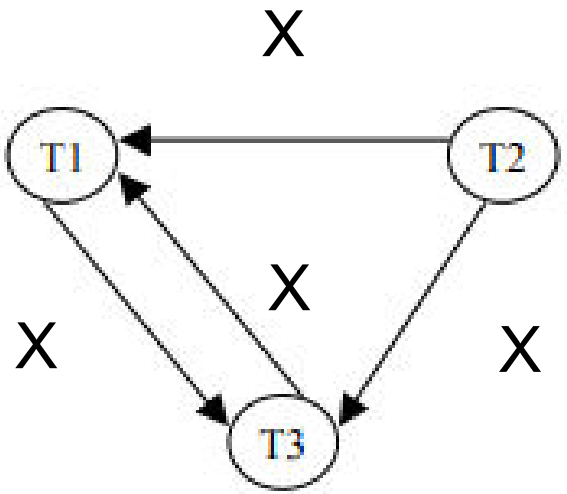

Serializable.

The equivalent serial schedule is: *r2(X); r3(X); w3(X); r1(X); w1(X);*

# Solution

d). *r3(X); r2(X); r1(X); w3(X); w1(X);*

The serialization graph is:




Not serializable.

# Characterizing Schedules Based on Recoverability

- A shorthand notation for describing a schedule uses the symbols ***b*, *r*, *w*, *e*, *c*, and *a*** for the operations

begin_transaction, read_item, write_item, end_transaction, commit, and abort, respectively

# Schedules

- $S_a$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;
- Same as $S_a$ *except 2 commit operations*
- $S_b$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$;$c_2$; $w_1(Y)$;$c_1$
- $S_b$ *is recoverable , even though it suffers from lost update problem*

- $S_c$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $c_2$ ;$a_1$;
- Is not recoverable because T2 reads item X from T1, and the T2 commits before T1 commits.

# Isolation levels in Transaction Processing

- A transaction isolation level is defined by the following phenomena –
- **Dirty Read –** A Dirty read is the situation when a transaction reads a data that has not yet been committed.
- **Non Repeatable read –** Non Repeatable read occurs when a transaction reads same row twice, and get a different value each time.
- **Phantom Read –** Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different.

Based on these phenomena, The SQL standard defines four isolation levels :

# Isolation levels in Transaction Processing

- **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.

- **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.

# Isolation levels in Transaction Processing

- **Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.
- **Serializable –** This is the Highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

# Characterizing Schedules based on Serializability (3)

- Being serializable is not the same as being serial
- Being serializable implies that the schedule is a correct schedule.
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

# Characterizing Schedules based on Serializability (4)

- Serializability is hard to check.
  - Interleaving of operations occurs in an operating system through some scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved.

# Characterizing Schedules based on Serializability (5)

Practical approach:

- Come up with methods (protocols) to ensure serializability.
- It's not possible to determine when a schedule begins and when it ends.
  - Hence, we reduce the problem of checking the whole schedule to checking only a **committed project** of the schedule (i.e. operations from only the committed transactions.)
- Current approach used in most DBMSs:
  - Use of locks with two phase locking

# Characterizing Schedules based on Serializability (6)

- View equivalence:
  - A less restrictive definition of equivalence of schedules
- View serializability:
  - Definition of serializability based on view equivalence.
  - A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

# Characterizing Schedules based on Serializability (7)

- Two schedules are said to be view equivalent if the following three conditions hold:
  1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
  2. For any operation Ri(X) of Ti in S, if the value of X read by the operation has been written by an operation Wj(X) of Tj (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation Ri(X) of Ti in S'.
  3. If the operation Wk(Y) of Tk is the last operation to write item Y in S, then Wk(Y) of Tk must also be the last operation to write item Y in S'.

# Characterizing Schedules based on Serializability (8)

- The premise behind view equivalence:
  - 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.
  - "**The view**": the read operations are said to see *the same view* in both schedules.

# Characterizing Schedules based on Serializability (9)

- **Relationship between view and conflict equivalence**:
  - The two are same under **constrained write assumption** which assumes that if T writes X, it is constrained by the value of X it read; i.e., new X = f(old X)
  - Conflict serializability is **stricter** than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serializable.
  - Any conflict serializable schedule is also view serializable, but not vice versa.

# Characterizing Schedules based on Serializability (10)

- Relationship between view and conflict equivalence (cont):
  - Consider the following schedule of three transactions
    - T1: r1(X), w1(X); T2: w2(X); and T3: w3(X):
  - Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;

- In Sa, the operations w2(X) and w3(X) are blind writes, since T1 and T3 do not read the value of X.
  - Sa is view serializable, since it is view equivalent to the serial schedule T1, T2, T3.
  - However, Sa is not conflict serializable, since it is not conflict equivalent to any serial schedule.

# Characterizing Schedules based on Serializability (14)

## Other Types of Equivalence of Schedules

- Under special **semantic constraints**, schedules that are otherwise not conflict serializable may work correctly.
  - Using commutative operations of addition and subtraction (which can be done in any order) certain non-serializable transactions may work correctly

# Characterizing Schedules based on Serializability (15)

## Other Types of Equivalence of Schedules (contd.)

- Example: bank credit / debit transactions on a given item are **separable** and **commutative**.
  - Consider the following schedule S for the two transactions:
  - Sh : r1(X); w1(X); r2(Y); w2(Y); r1(Y); w1(Y); r2(X); w2(X);
  - Using conflict serializability, it is **not serializable**.
  - However, if it came from a (read,update, write) sequence as follows:
    - r1(X); X := X – 10; w1(X); r2(Y); Y := Y – 20;r1(Y);
    - Y := Y + 10; w1(Y); r2(X); X := X + 20; (X);
  - Sequence explanation: debit, debit, credit, credit.
  - It is a *correct schedule* *for the given semantics*

# 6 Transaction Support in SQL2 (1)

- A **single** SQL statement is always considered to be **atomic**.
  - Either the statement completes execution without error or it fails and leaves the database unchanged.
- With SQL, there is no explicit Begin Transaction statement.
  - Transaction initiation is done implicitly when particular SQL statements are encountered.
- Every transaction must have an explicit end statement, which is either a COMMIT or ROLLBACK.

# Transaction Support in SQL2 (2)

Characteristics specified by a SET TRANSACTION statement in SQL2:

- **Access mode**:
  - READ ONLY or READ WRITE.
    - The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.
- **Diagnostic size** n, specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area.

# Transaction Support in SQL2 (3)

Characteristics specified by a SET TRANSACTION statement in SQL2 (contd.):

- **Isolation level** <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.
  - With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability.
  - However, if any transaction executes at a lower level, then serializability may be violated.

# Transaction Support in SQL2 (4)

Potential problem with lower isolation levels:

- **Dirty Read**:
  - Reading a value that was written by a transaction which failed.
- **Nonrepeatable Read**:
  - Allowing another transaction to write a new value between multiple reads of one transaction.
  - A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.
    - Consider that T1 reads the employee salary for Smith. Next, T2 updates the salary for Smith. If T1 reads Smith's salary again, then it will see a different value for Smith's salary.

# Transaction Support in SQL2 (5)

- Potential problem with lower isolation levels (contd.):
  - Phantoms:
    - New rows being read using the same read with a condition.
      - A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.
      - Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1.
      - If T1 is repeated, then T1 will see a row that previously did not exist, called a phantom.

# Transaction Support in SQL2 (6)

- Sample SQL transaction:

```
EXEC SQL whenever sqlerror go to UNDO;
EXEC SQL SET TRANSACTION
          READ WRITE
          DIAGNOSTICS SIZE 5
          ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT
          INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
          VALUES ('Robert','Smith','991004321',2,35000);
EXEC SQL UPDATE EMPLOYEE
          SET SALARY = SALARY * 1.1
          WHERE DNO = 2;
EXEC SQL COMMIT;
          GOTO  THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END:  ...
```

# Transaction Support in SQL2 (7)

- Possible violation of serializabilty:

| Isolation level | Type of Violation: Dirty read | nonrepeatable read | phantom |
|---|---|---|---|
| READ UNCOMMITTED | yes | yes | yes |
| READ COMMITTED | no | yes | yes |
| REPEATABLE READ | no | no | yes |
| SERIALIZABLE | no | no | no |

# Summary

- Transaction and System Concepts
- Desirable Properties of Transactions
- Characterizing Schedules based on Recoverability
- Characterizing Schedules based on Serializability
- Transaction Support in SQL

# Isolation levels in Transaction Processing

A transaction isolation level is defined by the following phenomena –

- **Dirty Read –** A Dirty read is the situation when a transaction reads a data that has not yet been committed. Reading a value that was written by a transaction which failed.

  $S_b$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$;$c_2$; $w_1(Y)$;$c_1$

- **Non Repeatable read –** Non Repeatable read occurs when a transaction reads same row twice, and get a different value each time. Allowing another transaction to write a new value between multiple reads of one transaction.
  - A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.
- **Phantom Read –** Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different.

Based on these phenomena, The SQL standard defines four isolation levels :

# Isolation levels in Transaction Processing

- **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.

- **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.

# Isolation levels in Transaction Processing

- **Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.

- **Serializable –** This is the Highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

# Transaction Support in SQL

- Possible violation of serializabilty:

| Isolation level | Type of Violation: Dirty read | nonrepeatable read | phantom |
|---|---|---|---|
| READ UNCOMMITTED | yes | yes | yes |
| READ COMMITTED | no | yes | yes |
| REPEATABLE READ | no | no | yes |
| SERIALIZABLE | no | no | no |

# Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

- **Recoverable schedule**:
  - One where no transaction needs to be rolled back.
  - A schedule S is recoverable **if no transaction T in S commits** until all transactions T’ that have written an item that T reads have committed.
- **Cascadeless schedule**:
  - One where **every transaction reads only the items** that are **written by committed transactions.**

# Characterizing Schedules based on Recoverability

Schedules classified on recoverability (contd.):

- **Schedules requiring cascaded rollback**:
  - A schedule in which **uncommitted transactions that read an item from a failed transaction** must be rolled back.
- **Strict Schedules**:
  - A schedule in which a **transaction can neither read or write an item X until the last transaction that wrote X has committed**.

# Characterizing Schedules Based on Recoverability

- A shorthand notation for describing a schedule uses the symbols ***b*, *r*, *w*, *e*, *c*, and *a*** for the operations

begin_transaction, read_item, write_item, end_transaction, commit, and abort, respectively

# Schedules

- $S_a$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;
- Same as $S_a$ *except 2 commit operations*
- $S_b$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$;$c_2$; $w_1(Y)$;$c_1$
- $S_b$ *is recoverable , even though it suffers from lost update problem*

- $S_c$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $c_2$ ;$a_1$;
- Is not recoverable because T2 reads item X from T1, and the T2 commits before T1 commits.