


# Transaction Processing

# Basic Concept of Transaction

Suppose we have to calculate total marks of a student. Then what are the steps involved in it? Get the marks in each subject (say we have 3 subjects), calculate the total and then display it. Here all the 3 steps are called as transaction and it consists of 3 small actions which cannot be further divided.



```
graph LR; A[Get marks in Subject1] --- B[Get marks in Subject2]; B --- C[Get marks in Subject3]; C --- D[Calculate the Total]; D --- E[Display the Total]; subgraph Transaction [Transaction - Calculate Total marks]; A; B; C; D; E; end
```


Get marks in Subject1  
Get marks in Subject2  
Get marks in Subject3  
Calculate the Total  
Display the Total

Transaction – Calculate Total marks

# Transaction Introduction

- Set of operations that form a single logical unit of work are called as transactions.
- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer \$50 from account A to account B:

1. **read**(A)
2.  $A := A - 50$
3. **write**(A)
4. **read**(B)
5.  $B := B + 50$
6. **write**(B)



If the database was in consistent state before a transaction, then after execution of the transaction also, the database must be in a consistent state. For example, a transfer of money from one bank account to another requires two changes to the database both must succeed or fail together.

You are working on a system for a bank. A customer goes to the ATM and instructs it to transfer Rs. 1000 from savings to a checking account.

This simple transaction requires two steps:

- Subtracting the money from the savings account balance.
- Adding the money to the checking account balance.

Saving Accounts  
Rs. 5000 to 1000

Checking Accounts  
Rs. 1000 to 2000

Transaction  
1 Subtract Rs 1000 from saving  
(machine crashes)  
2 Add Rs. 1000 to checking  
(money disappears)

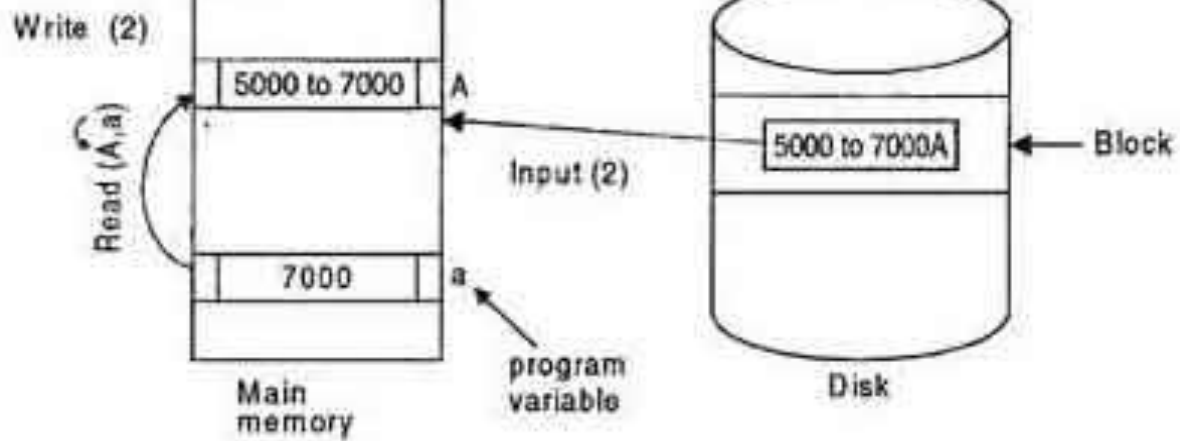
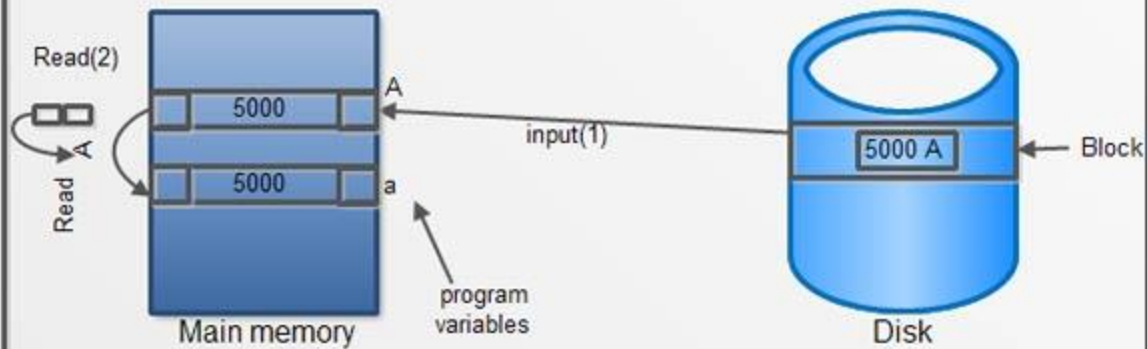
A Simple Transaction

# Processes of Transaction

- **read(X)**, which transfers the data item X from the database to a variable, also called X, in a buffer in main memory belonging to the transaction that executed the read operation.
- **write(X)**, which transfers the value in the variable X in the main-memory buffer of the transaction that executed the write to the data item X in the database.



## Read Operation



## Write Operation

# Transaction Properties (ACID)

A transaction has following properties:

1. Atomicity
2. Consistency
3. Isolation
4. Durability



# ATOMICITY

- If the transaction fails after step 3 and before step 6, money will be “**lost**” leading to an inconsistent database state
  - ▶ Failure could be due to software or hardware
- The system should ensure that updates of a partially executed transaction are not reflected in the database
- **All or nothing**, regarding the execution of the transaction

# CONSISTENCY

- The sum of A and B is unchanged by the execution of the transaction.
- Execution of a (single) transaction preserves the consistency of the database.
- A transaction must see a consistent database and must leave a consistent database
- During transaction execution the database may be temporarily inconsistent.
  - ▶ Constraints to be verified only at the end of the transaction

# ISOLATION

- Concurrent execution of two or more transactions **should be consistent** is called as Isolation.
- If between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum  $A + B$  will be less than it should be).

T1		T2
1.	<b>read</b> (A)	
2.	$A := A - 50$	
3.	<b>write</b> (A)	
	print(A+B)	read(A), read(B),
4.	<b>read</b> (B)	
5.	$B := B + 50$	
6.	<b>write</b> (B)	

# DURABILITY

- After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
- Database should be recoverable under any type of crash.

# Example of Fund Transfer

- Transaction to transfer \$50 from account A to account B:

1.**read**(A)

2. $A := A - 50$

3.**write**(A)

4.**read**(B)

5. $B := B + 50$

6.**write**(B)



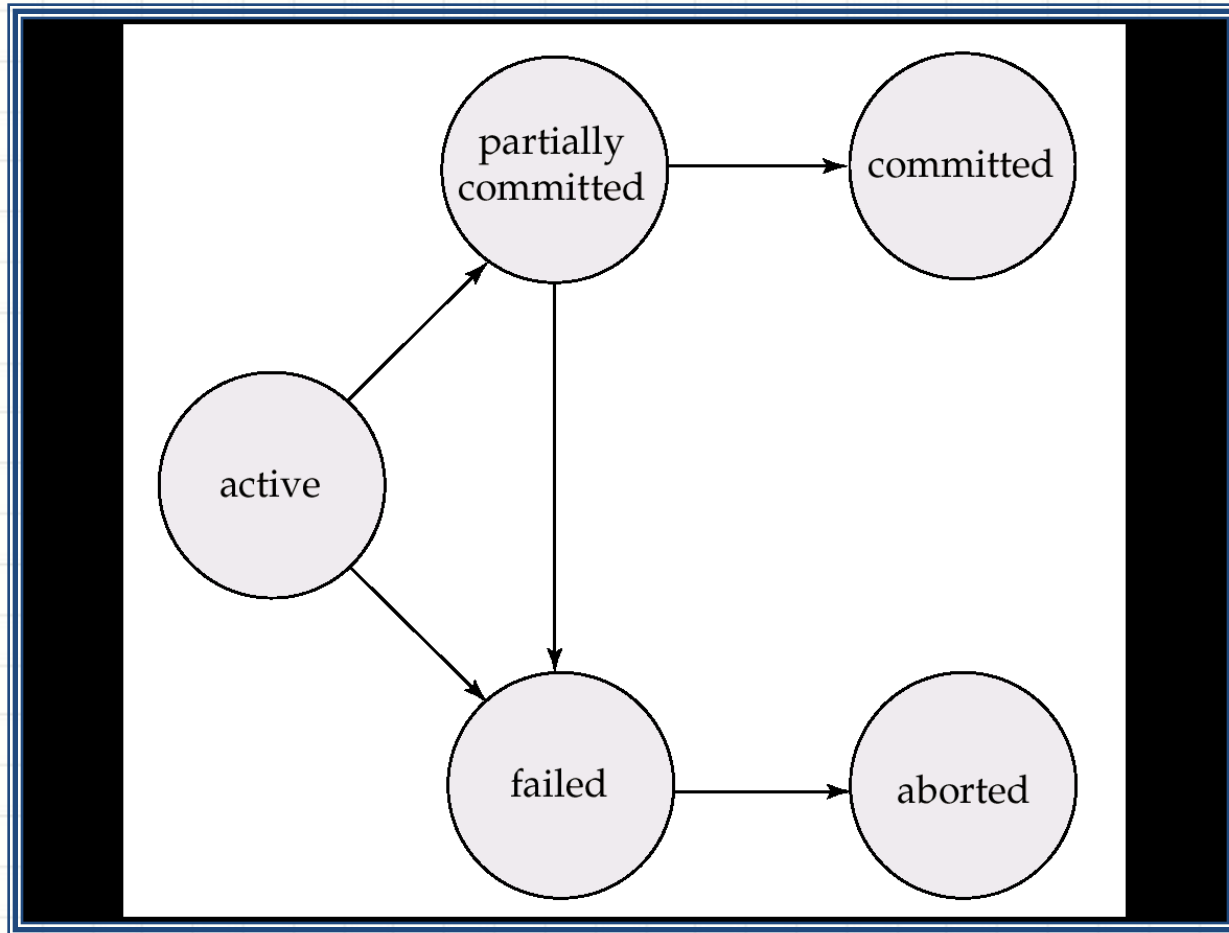
# **Transaction and System Concepts**

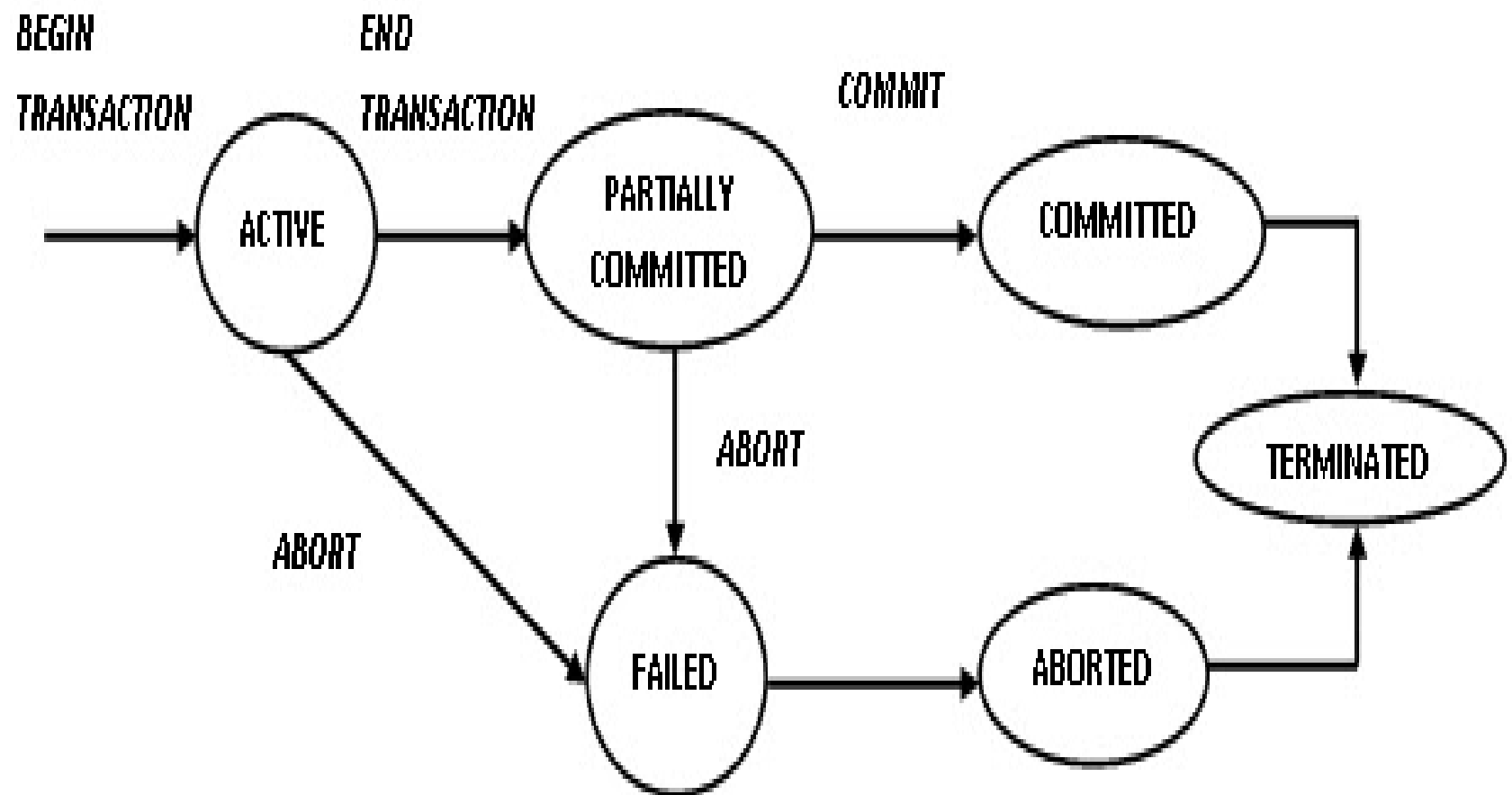


# Transaction States and Additional Operations

- BEGIN\_TRANSACTION
- READ or WRITE
- END\_TRANSACTION
- COMMIT\_TRANSACTION
- ROLLBACK (or ABORT)

# TRANSACTION STATES





# TRANSACTION STATES

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** – after the discovery that normal execution can no longer proceed.

# TRANSACTION STATES

- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - hardware or software error
  - kill the transaction
    - Internal logical error
- **Committed** – after successful completion. Once a transaction has committed, we cannot undo its effects by aborting it.

# The System Log

- To be able to recover from failures that affect transactions, the system maintains a log.
- 1. [**start\_transaction**,  $T$ ]. Indicates that transaction  $T$  has started execution.
- 2. [**write\_item**,  $T$ ,  $X$ , *old\_value*, *new\_value*]. Indicates that transaction  $T$  has
  - changed the value of database item  $X$  from *old\_value* to *new\_value*.
- 3. [**read\_item**,  $T$ ,  $X$ ]. Indicates that transaction  $T$  has read the value of database
  - item  $X$ .
- 4. [**commit**,  $T$ ]. Indicates that transaction  $T$  has completed successfully, and
  - affirms that its effect can be committed (recorded permanently) to the database.
- 5. [**abort**,  $T$ ]. Indicates that transaction  $T$  has been aborted



# Example

1

## Transaction T1

BEGIN TRANSACTION

    READ(A);

    A := A - 500;

    WRITE(A);

    READ(B);

    B := B + 500;


    WRITE(B);

    COMMIT;

END TRANSACTION;

# Transaction support in SQL

- **ACID properties** can ensure the concurrent execution of multiple transactions without conflict.
- Following commands are used to control transactions. It is important to note that these statements **cannot be used while creating tables** and are only used with the DML Commands such as – **INSERT, UPDATE and DELETE.**

- 
1. BEGIN TRANSACTION: It indicates the start point of an explicit or local transaction.

**BEGIN TRANSACTION transaction\_name ;**

2. SET TRANSACTION: Places a name on a transaction.


**SET TRANSACTION [ READ WRITE | READ ONLY ];**

3. COMMIT: If everything is in order with all statements within a single transaction, all changes are recorded together in the database is called committed. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

Student				
Rol_No	Name	Address	Phone	Age
1	Ram	Delhi	9455123451	18
2	Ramesh	Gurgaon	9652431543	18
3	Sujit	Rohtak	9156253131	20
4	Suresh	Delhi	9156768971	18
3	Sujit	Rohtak	9156253131	20
2	Ramesh	Gurgaon	9652431543	18


DELETE FROM Student WHERE AGE = 20;  
COMMIT;

Rol_No	Name	Address	Phone	Age
1	Ram	Delhi	9455123451	18
2	Ramesh	Gurgaon	9652431543	18
4	Suresh	Delhi	9156768971	18
2	Ramesh	Gurgaon	9652431543	18



4. **ROLLBACK**: If any error occurs with any of the SQL grouped statements, all changes need to be aborted. The process of reversing changes is called rollback. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

```
DELETE FROM Student WHERE AGE = 20;  
ROLLBACK;
```



5. **SAVEPOINT**: creates points within the groups of transactions in which to ROLLBACK.

- A SAVEPOINT is a point in a transaction in which you can roll the transaction back to a certain point without rolling back the entire transaction.



- Syntax

*SAVEPOINT SAVEPOINT\_NAME;*

*Example:*

SAVEPOINT SP1;

//Savepoint created.

DELETE FROM Student WHERE AGE = 20;

//deleted

SAVEPOINT SP2;

//Savepoint created.

- Output


Rol_No	Name	Address	Phone	Age
1	Ram	Delhi	9455123451	18
2	Ramesh	Gurgaon	9652431543	18
4	Suresh	Delhi	9156768971	18
2	Ramesh	Gurgaon	9652431543	18




# **concurrency control techniques**


# Introduction


- a number of concurrency control techniques that are used to ensure the **noninterference or isolation** property of concurrently executing transactions.
- Most of these techniques ensure **serializability of schedules**
- using concurrency control protocols (sets of rules) that guarantee serializability.

- 
- One important set of protocols—known as two-phase locking protocols.
  - Locking
  - Timestamps
  - multiversion concurrency control protocols

- 
- **validation or certification** of a transaction after it executes its operations.
  - optimistic protocols
  - Granularity – affects the concurrency control




- 
- what portion of the database a data item represents.
  - item can be as small as a single attribute (field) value.
  - as large as a disk block
  - even a whole
  - file or the entire database.



Concurrency control is provided in a database to:

- (i) enforce isolation among transactions.
- (ii) preserve database consistency through consistency preserving execution of transactions.
- (iii) resolve read-write and write-read conflicts.




# Various concurrency control techniques are:

- 1. Two-phase locking Protocol**
- 2. Time stamp ordering Protocol**
3. Multi version concurrency control
4. Validation concurrency control

# Two-Phase Locking Techniques

- Used to **control concurrent execution** of transactions are based on the concept of locking data items.
- A **lock** is a variable associated with a data item that **describes the status of the item** with respect to **possible operations** that can be applied to it.
- **one lock** for each data item in the database

- 
- Locks are used as synchronizing the access by concurrent transactions to the database items.
  - nature and types of locks Guarantee serializability of transaction schedules.
  - Two problems associated with the use of locks—**deadlock and starvation.**

# Types of Locks and System Lock Tables

- binary locks
- shared/exclusive locks
- certify lock

# Binary Locks

- A binary lock can have two states or values.
- locked and unlocked (or 1 and 0, for simplicity).
- A **distinct lock** is associated with each database item X.
- If the value of the lock on **X is 1**, item **X cannot be accessed by a database operation that requests the item.**
- If the value of the lock on **X is 0**, the **item can be accessed when requested**, and the lock value is changed to 1.



- We refer to the current value (or state) of the lock associated with item **X as lock(X).**
- **lock\_item and unlock\_item**, are used with binary locking.

### Steps:

1. If  $LOCK(X) = 1$ , the transaction is forced to wait.
2. If  $LOCK(X) = 0$ , it is set to 1 (the transaction locks the item) and the transaction is allowed to access item X.

When the transaction is through using the item, it issues an `unlock_item(X)` operation, which sets  $LOCK(X)$  back to 0 (unlocks the item) so that X may be accessed by other transactions

# Rules for Binary lock

1. A transaction must issue the operation lock\_item (A) before any read\_item (A) or write\_item (A) operations are performed in T.
2. A transaction T must issue the operation unlock\_item (A) after all read\_item (A) and write\_item (A) operations are completed in T.
3. A transaction T will not issue a lock\_item (A) operation if it already holds the lock on Item A.
4. A transaction T will not issue an unlock\_item (A) operation unless it already holds the lock on item A.
5. The lock manager module of the DBMS can enforce these rules. Between the lock\_item (A) and unlock\_item (A) operations in transaction T, it is said to hold the lock on item A. At most one transaction can hold the lock on a particular item. Thus no two transactions can access the same item concurrently.

- a binary lock enforces **mutual exclusion** on the data item.

```
lock_item(X):  
  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);  
          go to B  
        end;  
unlock_item(X):  
  LOCK(X) ← 0;                (* unlock the item *)  
  if any transactions are waiting  
  then wakeup one of the waiting transactions;
```

---

## **Merits of Binary Locks :**


- They are simple to implement since they are effectively mutually exclusive and establish isolation perfectly.
- Binary Locks demand less from the system since the system must only keep a record of the locked items. The system is the lock manager subsystem which is a feature of all DBMSs today.

## **Drawbacks of Binary Locks :**


- Binary locks are highly restrictive.
- They do not even permit reading of the contents of item X. As a result, they are not used commercially.

# Shared/Exclusive (or Read/Write) Locks

- **Exclusive Locks (write):** Transactions request exclusive locks on data items they need to access. This ensures that **only one transaction can modify a data item at a time.**
- **Shared Locks (read):** Transactions request shared locks on data items when **they need to read data.** Multiple transactions can hold shared locks simultaneously, but exclusive locks cannot be acquired while shared locks are held.

- 
- We should allow several transactions to access the **same item X** if they **all access X for reading purposes only**.
  - However, if a transaction is to **write an item X**, it must have **exclusive access to X**.
  - For this purpose, a different type of lock called a **multiple-mode lock is used**.





Three locking operations:


1. `read_lock(X)`,
2. `write_lock(X)`
3. `unlock(X)`

three possible states:

read-locked, write-locked, or  
unlocked.



- A **read-locked item** is also called **share-locked** because other transactions are allowed to **read the item**, whereas a **write-locked** item is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.
- One method for implementing the preceding operations on a **read/write lock** is to **keep track of the number of transactions** that hold a shared (read) lock on an item in the lock table. Each record in the lock table will have **four fields**

- 
- Data\_item\_name,
  - LOCK,
  - No\_of\_reads,
  - Locking\_transaction(s)

- The value (state) of LOCK is either read-locked or write-locked, suitably coded (if we assume no records are kept in the lock table for unlocked items).
- If **LOCK(X)=write-locked**, the value of locking\_transaction(s) is a single transaction that holds the **exclusive (write) lock on X**.
- If **LOCK(X)=read-locked**, the value of locking\_transaction(s) is a list of one or more transactions that hold the **shared (read) lock on X**.

# shared/exclusive locking scheme

## rules

1. A transaction  $T$  must issue the operation  $\text{read\_lock}(X)$  or  $\text{write\_lock}(X)$  before any  $\text{read\_item}(X)$  operation is performed in  $T$ .
2. A transaction  $T$  must issue the operation  $\text{write\_lock}(X)$  before any  $\text{write\_item}(X)$  operation is performed in  $T$ .
3. A transaction  $T$  must issue the operation  $\text{unlock}(X)$  after all  $\text{read\_item}(X)$  and  $\text{write\_item}(X)$  operations are completed in  $T$ .
4. A transaction  $T$  will not issue a  $\text{read\_lock}(X)$  operation if it already holds a read (shared) lock or a write (exclusive) lock on item  $X$ .

- 5. A transaction  $T$  will not issue a `write_lock(X)` operation if it already holds a read (shared) lock or write (exclusive) lock on item  $X$ .
- 6. A transaction  $T$  will not issue an `unlock(X)` operation unless it already holds a read (shared) lock or a write (exclusive) lock on item  $X$ .

**read\_lock(X):**

```
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;
```

**write\_lock(X):**

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

**unlock (X):**

```
  if LOCK(X) = "write-locked"
    then begin LOCK(X) ← "unlocked";
               wakeup one of the waiting 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";
                     wakeup one of the waiting transactions, if any
                   end
        end;
```

**Figure 22.2**

Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.



# Examples

- **Example of shared lock:** Consider the situation where two transactions read a person's account balance. By establishing a shared lock on the database, they will read it. However, a shared lock prevents another transaction from updating that account's balance until the reading procedure is completed.
- **Example of exclusive lock:** When a transaction needs to update a person's account balance, for example, you can allow it to proceed by putting an X lock on this transaction. As a result, the exclusive lock prevents the second transaction from reading or writing.



# Compatibility of the locking

States		Current state of locking of data items		
		Unlocked	Shared	Exclusive
Lock mode of request	Unlocked		Yes	Yes
	Shared	Yes	Yes	No*
	Exclusive	Yes	No*	NO

# Conversion of Locks

- a transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another.
- Upgrade
- downgrade




## Important Note:

Any no. of transactions can hold shared lock, but Exclusive lock can be hold only one transaction at a time.

# Guaranteeing Serializability by Two-Phase Locking

- A transaction is said to follow the two-phase locking protocol if all locking operations (read\_lock, write\_lock) precede the first unlock operation in the transaction.
- expanding or growing (first) phase
- shrinking (second) phase

- 
- an **expanding or growing (first) phase**, during which new locks on items can be acquired but none can be released;
  - a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired.
  - If Conversion is allowed

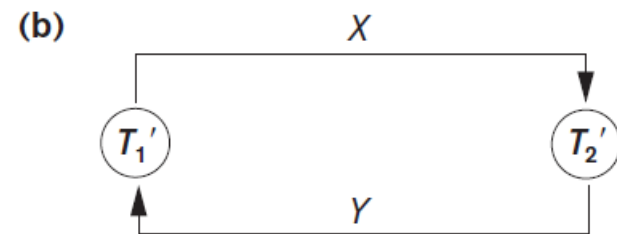
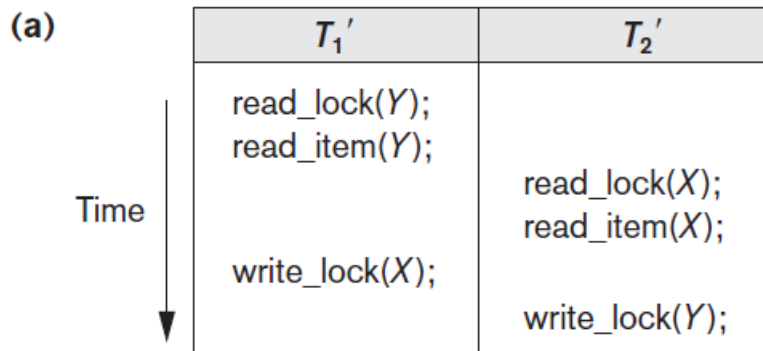
$T_1$	$T_2$
<pre> read_lock(Y); read_item(Y); unlock(Y); write_lock(X); read_item(X); X := X + Y; write_item(X); unlock(X); </pre>	<pre> read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); Y := X + Y; write_item(Y); unlock(Y); </pre>

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

- Basic, Conservative, Strict, and Rigorous Two-Phase Locking.

# Dealing with Deadlock and Starvation

- Deadlock occurs when each transaction  $T$  in a set of two or more transactions is waiting for some item that is locked by some other transaction  $T'$  in the set.





# Deadlock Prevention Protocols

- One way to prevent deadlock is to use a deadlock prevention protocol.
- conservative two-phase locking
- A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation.

- transaction timestamp  $TS(T)$ .
- Two schemes that prevent deadlock are called **wait-die** and **woundwait**.
- **Wait-die.** If  $TS(T_i) < TS(T_j)$ , then ( $T_i$  older than  $T_j$ )  $T_i$  is allowed to wait; otherwise ( $T_i$  younger than  $T_j$ ) abort  $T_i$  ( $T_i$  dies) and restart it later with the same timestamp.
- **Wound-wait.** If  $TS(T_i) < TS(T_j)$ , then ( $T_i$  older than  $T_j$ ) abort  $T_j$  ( $T_i$  wounds  $T_j$ ) and restart it later with the same timestamp; otherwise ( $T_i$  younger than  $T_j$ )  $T_i$  is allowed to wait.

## 1. Wait-Die Scheme (Non-preemptive)

- Older transactions are allowed to wait.
- Younger transactions are killed (aborted and restarted) if they request a resource held by an older one

For example:

- Consider two transaction-  $T1 = 10$  and  $T2 = 20$
- If  $T1$  (older) wants a resource held by  $T2 \rightarrow T1$  waits
- If  $T2$  (younger) wants a resource held by  $T1 \rightarrow T2$  dies and restarts
- Prevents deadlock by not allowing a younger transaction to wait and form a wait cycle.

## 2. Wound-Wait Scheme (Preemptive)

- Older transactions are aggressive (preemptive) and can force younger ones to abort.
- Younger transactions must wait if they want a resource held by an older one.

For example:

- Consider two transaction-  $T1 = 10$  and  $T2 = 20$
- If  $T1$  (older) wants a resource held by  $T2 \rightarrow T2$  is killed,  $T1$  proceeds.
- If  $T2$  (younger) wants a resource held by  $T1 \rightarrow T2$  waits
- Prevents deadlock by not allowing younger transactions to block older ones.

### Wait - Die

It is based on a non-preemptive technique.

In this, older transactions must wait for the younger one to release its data items.


The number of aborts and rollbacks is higher in these techniques.

### Wound -Wait

It is based on a preemptive technique.

In this, older transactions never wait for younger transactions.

In this, the number of aborts and rollback is lesser.

- 
- Another group of protocols that prevent deadlock **do not require timestamps**.
  - These include the **no waiting (NW)** and **cautious waiting (CW) algorithms**.
  - In the no waiting algorithm, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not.





- In this case, **no transaction ever waits**, so no deadlock will occur. However, this scheme can cause **transactions to abort and restart needlessly**.
- The **cautious waiting algorithm** was proposed to **try to reduce the number of needless aborts/restarts**.
- Cautious waiting: If  $T_j$  is not blocked (not waiting for some other locked item), then  $T_i$  is blocked and allowed to wait; otherwise abort  $T_i$ .



# Deadlock Detection

- A second, more practical approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists.
- A simple way to detect a state of deadlock is for the system to construct and maintain a **wait-for graph**.

- 
- **Timeouts:** Another simple scheme to deal with deadlock is the use of timeouts.
  - In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists or not.

- 
- **Starvation:** Another problem that may occur when we use locking is starvation, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
  - One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served queue**

# Concurrency Control Based on Timestamp Ordering

- Timestamps
- The Timestamp Ordering Algorithm

# Timestamps

- **Timestamp** is a **unique identifier** created by the DBMS to identify a transaction.
- Timestamp values are assigned in the order in which the transactions are submitted to the system.
- Concurrency control techniques based on timestamp ordering **do not use locks**; hence, **deadlocks cannot occur**.



Timestamps can be generated in several ways.

- One possibility is to use a **counter** that is incremented each time its value is assigned to a transaction.
- A computer counter has a finite maximum value, so the system must periodically reset the **counter to zero** when **no transactions are executing for some short period of time**.
- Another possibility is to use the **current date/time value of the system clock** and ensure that no two timestamp values are generated during the same tick of the clock.




# The Timestamp Ordering Algorithm

- A schedule in which the transactions participate is then serializable, and **the only equivalent serial schedule permitted** has the transactions in order of their timestamp values. This is called **timestamp ordering (TO)**.



# Differences

- a schedule is serializable by **being equivalent to some serial schedule** allowed by the locking protocols.
- In timestamp ordering, however, the **schedule is equivalent to the particular serial order corresponding to the order of the transaction timestamps.**

- 
- The algorithm must ensure that, for each item accessed by **conflicting operations** in the schedule, the order in which the item is accessed **does not violate the timestamp order**.
  - To do this, the algorithm associates with each database item X **two timestamp (TS) values**

1. **read\_TS(X)**. The read timestamp of item X is the **largest timestamp among all the timestamps of transactions** that have successfully read item X—that is,  $\text{read\_TS}(X) = \text{TS}(T)$ , where **T is the youngest transaction that has read X successfully**.

2. **write\_TS(X)**. The write timestamp of item **X is the largest of all the timestamps of transactions** that have successfully written item X—that is,  $\text{write\_TS}(X) = \text{TS}(T)$ , where **T is the youngest transaction that has written X successfully**.

## Basic Timestamp Ordering (TO)

1. Whenever a transaction  $T$  issues a  $\text{write\_item}(X)$  operation, the following is checked:

a. If  $\text{read\_TS}(X) > \text{TS}(T)$  or if  $\text{write\_TS}(X) > \text{TS}(T)$ , then abort and roll back  $T$  and reject the operation. This should be done because some younger transaction with a timestamp greater than  $\text{TS}(T)$ —and hence after  $T$  in the timestamp ordering—has already read or written the value of item  $X$  before  $T$  had a chance to write  $X$ , thus violating the timestamp ordering.

b. If the condition in part (a) does not occur, then execute the  $\text{write\_item}(X)$  operation of  $T$  and set  $\text{write\_TS}(X)$  to  $\text{TS}(T)$ .

2. Whenever a transaction  $T$  issues a  $\text{read\_item}(X)$  operation, the following is checked:

a. If  $\text{write\_TS}(X) > \text{TS}(T)$ , then abort and roll back  $T$  and reject the operation. This should be done because some younger transaction with timestamp greater than  $\text{TS}(T)$ —and hence after  $T$  in the timestamp ordering—has already written the value of item  $X$  before  $T$  had a chance to read  $X$ .

b. If  $\text{write\_TS}(X) \leq \text{TS}(T)$ , then execute the  $\text{read\_item}(X)$  operation of  $T$  and set  $\text{read\_TS}(X)$  to the larger of  $\text{TS}(T)$  and the current  $\text{read\_TS}(X)$ .

# cascading rollback Problem

- If  $T$  is aborted and rolled back, any transaction  $T_1$  that may have used a value written by  $T$  must also be rolled back. Similarly, any transaction  $T_2$  that may have used a value written by  $T_1$  must also be rolled back, and so on.

## Strict Timestamp Ordering (TO)

- A variation of basic TO called strict TO ensures that the schedules are both strict (for easy recoverability) and (conflict) serializable.
- This algorithm does not cause deadlock.



# Example of Timestamp Ordering Protocol

- Assume there are two transactions  $T_1$  and  $T_2$  with timestamps  $TS(T_1)=1$
- $TS(T_1)=1$  and  $TS(T_2)=2$  Let's say both transactions want to access a data item  $Q$ .




- **Case 1:**  $T1$  reads  $Q$  Since  $T1$  is the oldest transaction, it reads  $Q$ , and the read timestamp  $RTS(Q)$  is updated to 1.
- **Case 2:**  $T2$  writes to  $Q$  Since  $T2$  has a newer timestamp (2), it can write to  $Q$  without conflict. The write timestamp  $WTS(Q)$  is updated to 2.
- **Case 3:**  $T1$  tries to write to  $Q$  Since  $TS(T1)=1$  is less than  $WTS(Q)=2$ , the write operation is rejected, and  $T1$  is rolled back because it would overwrite changes made by a newer transaction.

## Thomas's Write Rule:

A modification of the basic TO algorithm, known as Thomas's write rule, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the `write_item(X)` operation

## Advantages

- **Simple to implement:** Timestamp-based ordering is straightforward, relying purely on timestamps.
- **Deadlock-Free:** Since transactions are ordered by timestamps, there is no circular waiting, eliminating deadlocks.



## Disadvantages


- **Unnecessary Rollbacks:** Older transactions may be rolled back frequently, leading to wasted resources.
- **Starvation:** If newer transactions continue to access the same data items, older transactions may suffer from starvation and fail to commit.



# **Database Recovery Techniques**

# Recovery Concepts

- Database Systems like any other computer system, are subject to failures but the data stored in them must be available as and when required. When a database fails it must possess the facilities for fast recovery.
- It must also have atomicity i.e. either transactions are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database.

- 
- The main goal of recovery techniques is to ensure data integrity and consistency and prevent data loss.
  - Rollback/Undo Recovery Technique
  - Commit/Redo Recovery Technique



# Rollback/Undo Recovery Technique


- The rollback/undo recovery technique is based on the principle of backing out or undoing the effects of a transaction that has not been completed successfully due to a system failure or error.
- This technique is accomplished by undoing the changes made by the transaction using the log records stored in the transaction log.


# Commit/Redo Recovery Technique


- The commit/redo recovery technique is based on the principle of reapplying the changes made by a transaction that has been completed successfully to the database.
- This technique is accomplished by using the log records stored in the transaction log to redo the changes made by the transaction that was in progress at the time of the failure or error.
- The system uses the log records to reapply the changes made by the transaction and restore the database to its most recent consistent state.

# Checkpoint Recovery


- Checkpoint Recovery is a technique used to reduce the recovery time by periodically saving the state of the database in a checkpoint file.
- In the event of a failure, the system can use the checkpoint file to restore the database to the most recent consistent state before the failure occurred, rather than going through the entire log to recover the database.

- 
- Conceptually, we can distinguish two main techniques for recovery from noncatastrophic transaction failures: **deferred update and immediate update.**

- 
- The deferred update techniques **do not physically update the database on disk** until after a transaction reaches its commit point; then the updates are recorded in the database.
  - Before reaching commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers that the DBMS maintains (the DBMS main memory cache).
  - Before commit, the updates are recorded persistently in the log, and then after commit, the updates are written to the database on disk.

- 
- If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed.
  - Deferred update is also known as the NO-UNDO/REDO algorithm.





- 
- In the immediate update techniques, the database may be updated by some operations of a transaction before the transaction reaches its commit point.
  - In the general case of immediate update, both undo and redo may be required during recovery.
  - This technique, known as the UNDO/REDO algorithm





# Caching (Buffering) of Disk Blocks

- DBMS cache - purpose of holding these buffers
- replace (or flush) - some of the cache buffers to make space available for the new item.
- dirty bit - to indicate whether or not the buffer has been modified.
- dirty bit is set to 0 (zero)
- entry is set to 1 (one)

- 
- Two main strategies can be employed when flushing a modified buffer back to disk.
  - in-place updating – writes the buffer to the same original disk location
  - Shadowing - writes an updated buffer at a different disk location, so multiple versions of data items can be maintained, but this approach is not typically used in practice.

- 
- In general, the old value of the data item before updating is called the before image (BFIM), and the new value after updating is called the after image (AFIM). If shadowing is used, both the BFIM and the AFIM can be kept on disk; hence, it is not strictly necessary to maintain a log for recovering.

- 
- The purpose of a checkpoint is to write all of the modified pages out to disk from the buffer cache and thus provide a physically consistent point for recovery.
  - A "new" type of checkpoint, called a fuzzy checkpoint, was introduced to reduce the time it takes to checkpoint the database. The "old" type of checkpoint is now referred to as a sync checkpoint.



The main events that occur during a fuzzy checkpoint are as follow:

1. Database server prevents user threads from entering the critical section and also blocks other users from entering the critical section.
2. Modified partition partition entries are stored to disk.
3. All modified pages in the buffer pool are flushed to disk by the page cleaners that are not modified by the fuzzy operations. A Dirty Page Table (DPT) is constructed. The physical log and logical log buffer are also flushed to disk as a side effect of this operation.
4. The DPT is logged.
5. A checkpoint record is written to the logical log and the checkpoint reserved page is updated on disk.
6. The physical log is logically emptied.
7. Restriction on entering the critical section is released.