Outline

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

Transaction Concept

- 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*)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

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*)
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

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

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

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

ACID Properties

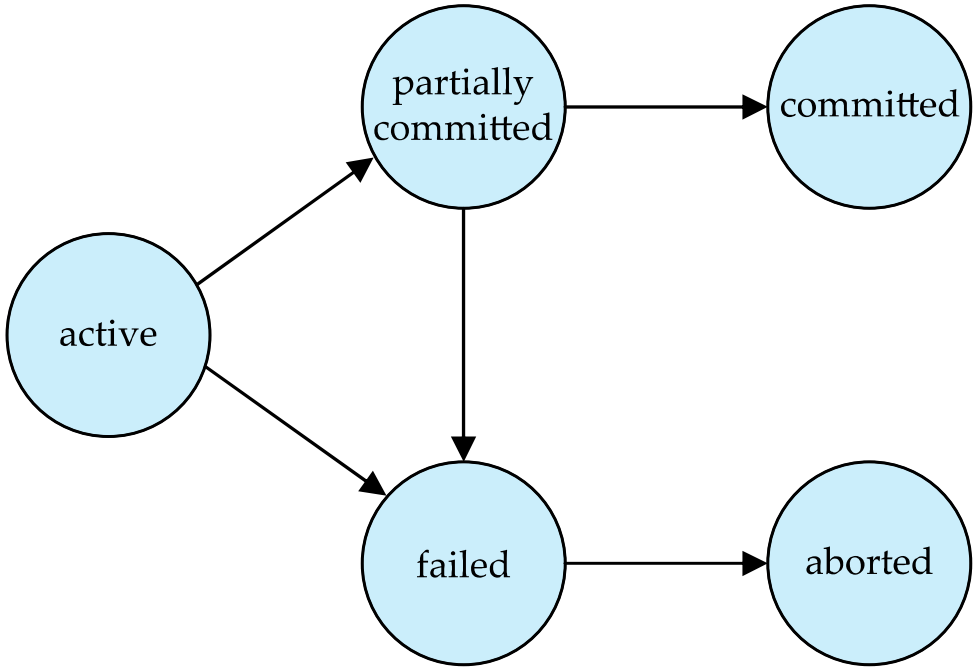
A **transaction** is a unit of program execution that accesses and possibly updates various data items.To preserve the integrity of data the database system must ensure:

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

Transaction State

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

Transaction State (Cont.)



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - e.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

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

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	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	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

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

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of ($A + B$).

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

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**

Simplified view of transactions

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

Conflicting Instructions

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

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

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

View Serializability

- Let *S* and *S'* be two schedules with the same set of transactions. *S* and *S'* are **view equivalent** if the following three conditions are met, for each data item *Q*,
  1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q*.
  2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'*.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

Other Notions of Serializability

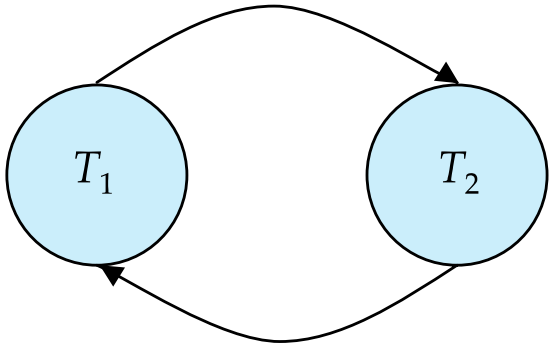
- The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.

T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B - 10$
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

- Determining such equivalence requires analysis of operations other than read and write.

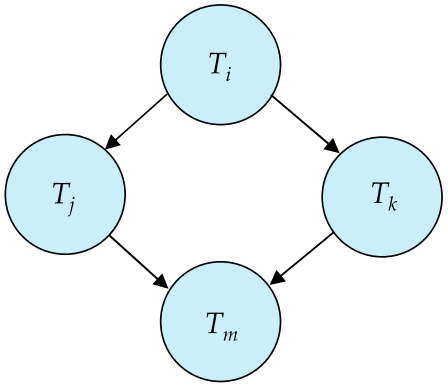
Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**

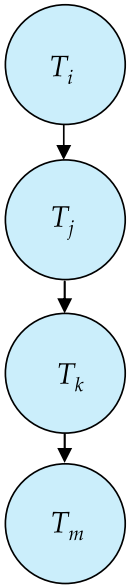


Test for Conflict Serializability

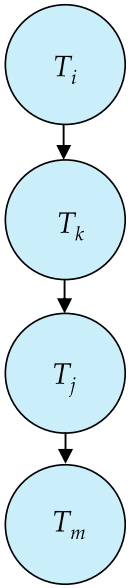
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be
    $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?



(a)



(b)



(c)

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus. existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Recoverable Schedules

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

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

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

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur;
  - For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Concurrency Control (Cont.)

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols (generally) do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids non-serializable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read.
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read.
  - Successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Levels of Consistency

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
- E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL

- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g., in JDBC -- connection.setAutoCommit(false);
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction
    - E.g. In SQL **set transaction isolation level serializable**
    - E.g. in JDBC -- connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)

Implementation of Isolation Levels

Overview

- Locking
  - Lock on whole database vs lock on items
  - How long to hold lock?
  - Shared vs exclusive locks
- Timestamps
  - Transaction timestamp assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a “snapshot” of the database

Transactions as SQL Statements

- E.g. Transaction 1:
  select *ID, name* **from** *instructor* **where** *salary* > 90000
- Transaction 2:
  insert into *instructor* **values** ('11111', 'James', 'Marketing', 100000)
- Suppose
  - T1 starts, finds tuples salary > 90000 using index and locks them
  - And then T2 executes.
  - Do T1 and T2 conflict? Does tuple level locking detect the conflict?
  - Instance of the **phantom phenomenon**
- Also consider T3 below, with Wu's salary = 90000
  update *instructor*
  set *salary* = *salary* * 1.1
  where *name* = 'Wu'
- Key idea: Detect "**predicate**" conflicts, and use some form of "**predicate locking**"

UNIT – V chapter 1

Transaction:

A **transaction** is a **unit** of program execution that accesses and possibly updates various data items.

Usually, a transaction is initiated by a user program written in a high-level data-manipulation language (typically SQL), or programming language with embedded database accesses in JDBC or ODBC.

A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**.

The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

A transaction is action, or series of actions, carried out by user or application, which accesses or updates contents of database.

It Transforms database from one consistent state to another, although consistency may be violated during transaction.

The concept of transaction provides a mechanism for describing logical units of database processing.

Transaction processing systems are systems with large databases and hundreds of concurrent users that are executing database transactions.

Examples of such systems include systems for reservations, banking, stock markets, super markets and other similar systems.

They require high availability and fast response time for hundreds of concurrent users.

Single User Vs. Multi User Systems:

- A DBMS is a single user if at most one user at a time can use the system.
- A DBMS is a multi user if many users can use the system and hence access the database concurrently.

- Multiple users can access databases and use the computer systems simultaneously because of the concept of Multiprogramming.
- Multiprogramming allows the computer to execute multiple programs or processes at the same time.
- If only a single central processing unit(CPU) exists, it can actually executes at most one process at a time.
- However multiprogramming operating systems executes some actions from one process then suspend that process and execute some actions of the next process,and so on.
- A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again.
- Hence concurrent execution of process is actually interleaved as illustrated in the following figure, which shows two processes A and B executing concurrently in an interleaved fashion.

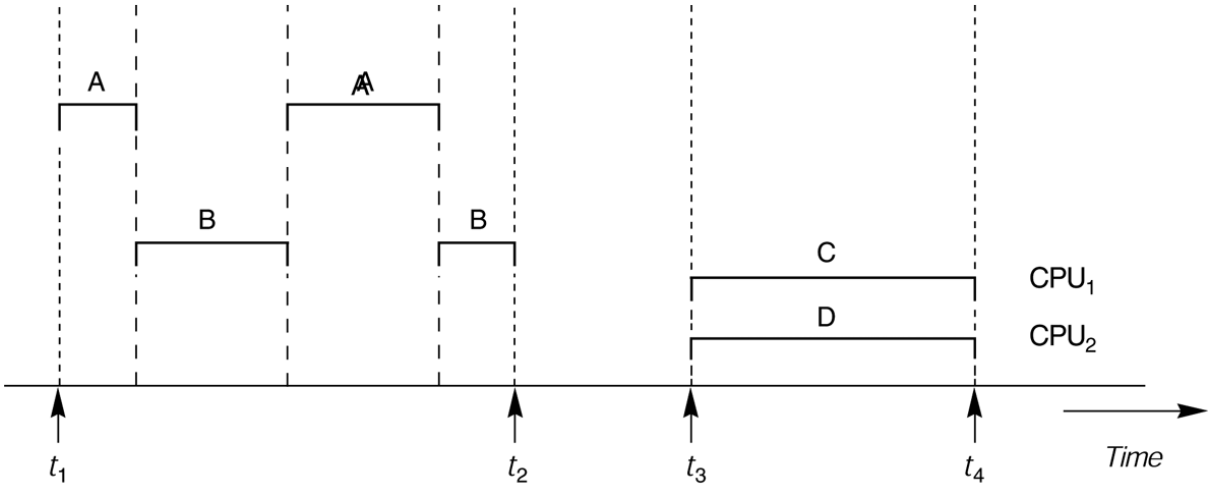


Fig. 4.1 Interleaved processing Vs. Parallel Processing of concurrent transactions.

Interleaving also prevents the long process from delaying other processes.

If the computer system has multiple hardware processors(CPUs), parallel processing of multiple processing is possible as illustrated the process C and D in the figure.

A transaction Can have one of two outcomes:

Success - transaction *commits* and database reaches a new consistent state.
Failure - transaction *aborts*, and database must be restored to consistent state before it started.. Such a transaction is *rolled back* or *undone*.
Committed transaction cannot be aborted.
Aborted transaction that is rolled back can be restarted later.

Transactions, Read and Write Operations, and DBMS Buffers

A **transaction** is an executing program that forms a logical unit of database processing.
A **transaction** includes one or more database access operations such as insertion, deletion, modification, or retrieval operations.
The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.
One way of specifying the transaction boundaries is by specifying explicit begin transaction and end transaction statements in an application program; in this case, all database access operations between the two are considered as forming one transaction.
A single application program may contain more than one transaction if it contains several transactions boundaries.
read-only transaction:
If the database operations in a transaction do not update the database but
Only retrieve data, the transaction is called a **read-only transaction.**
A database is basically represented as a collection of named data items.
Granularity:
The size of a data item is called its granularity, and it can be a field of some record in the database, or it may be a larger unit such as a record or even a whole disk block,
Basic database access operations:

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.

Executing a read_item(X) command includes the following steps:
1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory
3. Copy item X from the buffer to the program variable named X.

Executing a write_item(X) command includes the following steps:
1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk

Step 4 is the one that actually updates the database on disk. In some cases the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer.
Usually, the decision about when to store back a modified disk block that is in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system.
The DBMS will generally maintain a number of buffers in main memory that hold database disk blocks containing the database items being processed
A transaction includes read_item and wri te_item operations to access and update the database. Figure 4.2 shows examples of two very simple transactions.
The read-set of a transaction is the set of all items that the transaction reads, and the write-set is the set of all items that the transaction writes.

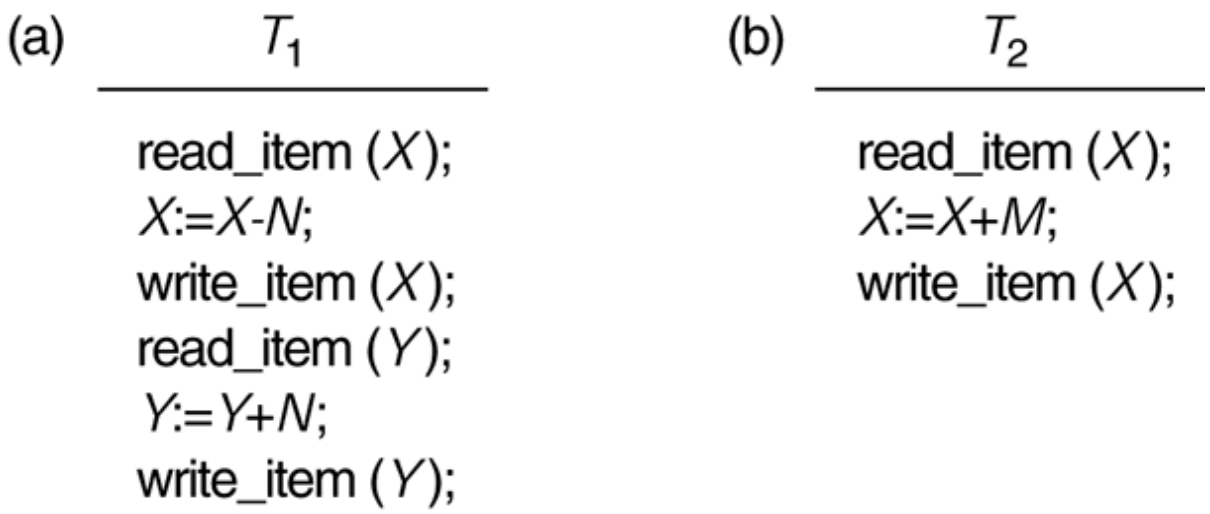
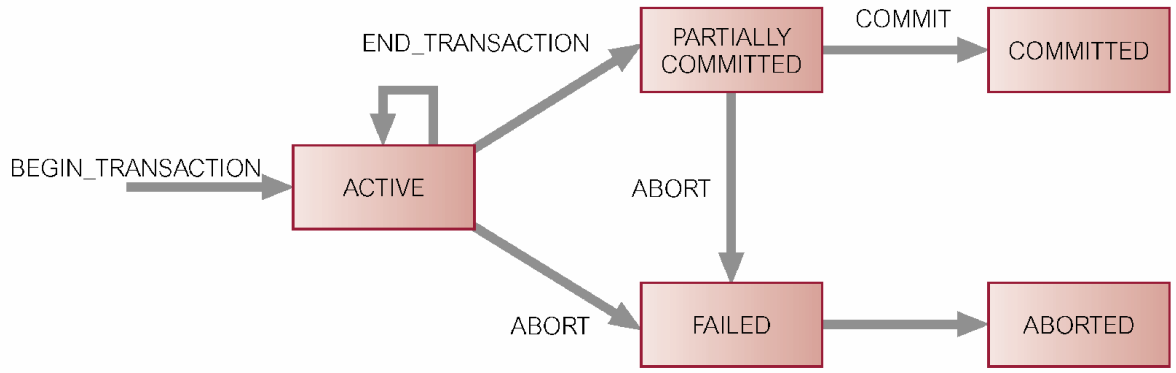


Fig 4.2: Two sample transactions. (a) Transaction T_1 . (b) Transaction T_2

Transaction States or State Transition Diagram and Additional Operations



A transaction is an atomic unit of work that is either completed 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.

Hence, the recovery manager keeps track of the following operations:

BEGIN_TRANSACTION: This marks the beginning of transaction execution.

READ DR 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 of transaction execution. However, at this point it may be necessary to check whether the changes introduced by

the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.

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.*

Figure 17.4 shows a state transition diagram that describes how a transaction moves through its execution states.

Active state: A transaction goes into an active state immediately after it , where it can issue READ and WRITE operations.

Partially committed state: When the transaction ends, it moves to the **partially** committed state. At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently

Committed state:

Once check in partially committed state is successful, the transaction is said to have reached its commit point and enters the committed state.

Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.

Failed state:

A transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state.

The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.

Terminated state:

The terminated state corresponds to the transaction leaving the system.

The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates.

ACID Properties or DESIRABLE PROPERTIES OF TRANSACTIONS

In DBMS **ACID** (*Atomicity*, *Consistency*, *Isolation*, *Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

Jim Gray defined these properties of a reliable transaction system in the late 1970s and developed technologies to achieve them automatically

Atomicity:

Atomicity refers to the ability of the DBMS to guarantee that either all of the operations of a transaction are performed or none of them are. Database modifications must follow an all or nothing rule. Each transaction is said to be atomic if when one part of the transaction fails, the entire transaction fails.

The atomicity property requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity.

If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.

Consistency:

The consistency property ensures that the database remains in a consistent state before the start of the transaction and after the transaction is over (whether successful or not).

The preservation of consistency is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints.

A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints that should hold on the database. A

database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction,

Isolation:

The isolation portion of the ACID Properties is needed when there are concurrent transactions. Concurrent transactions are transactions that occur at the same time, such as shared multiple users accessing shared objects.
Although multiple transactions may execute concurrently, each transaction must be independent of other concurrently executing transactions. A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Durability:

Maintaining updates of committed transactions is critical. These updates must never be lost. The ACID property of durability addresses this need. Durability refers to the ability of the system to recover committed transaction updates if either the system or the storage media fails. Features to consider for durability:

- recovery to the most recent successful commit after a database software failure
- recovery to the most recent successful commit after an application software failure
- recovery to the most recent successful commit after a CPU failure

- recovery to the most recent successful backup after a disk failure
- recovery to the most recent successful commit after a data disk failure

The System Log

To be able to recover from failures that affect transactions, the system maintains a log to keep track of all transaction operations that affect the values of database items.

This information may be needed to permit recovery from 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.

We now list the types of entries-called log records-that are written to the log and the action each performs.

In these entries, T refers to a unique transaction-id that is generated automatically by the system and is used to identify each transaction:

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.

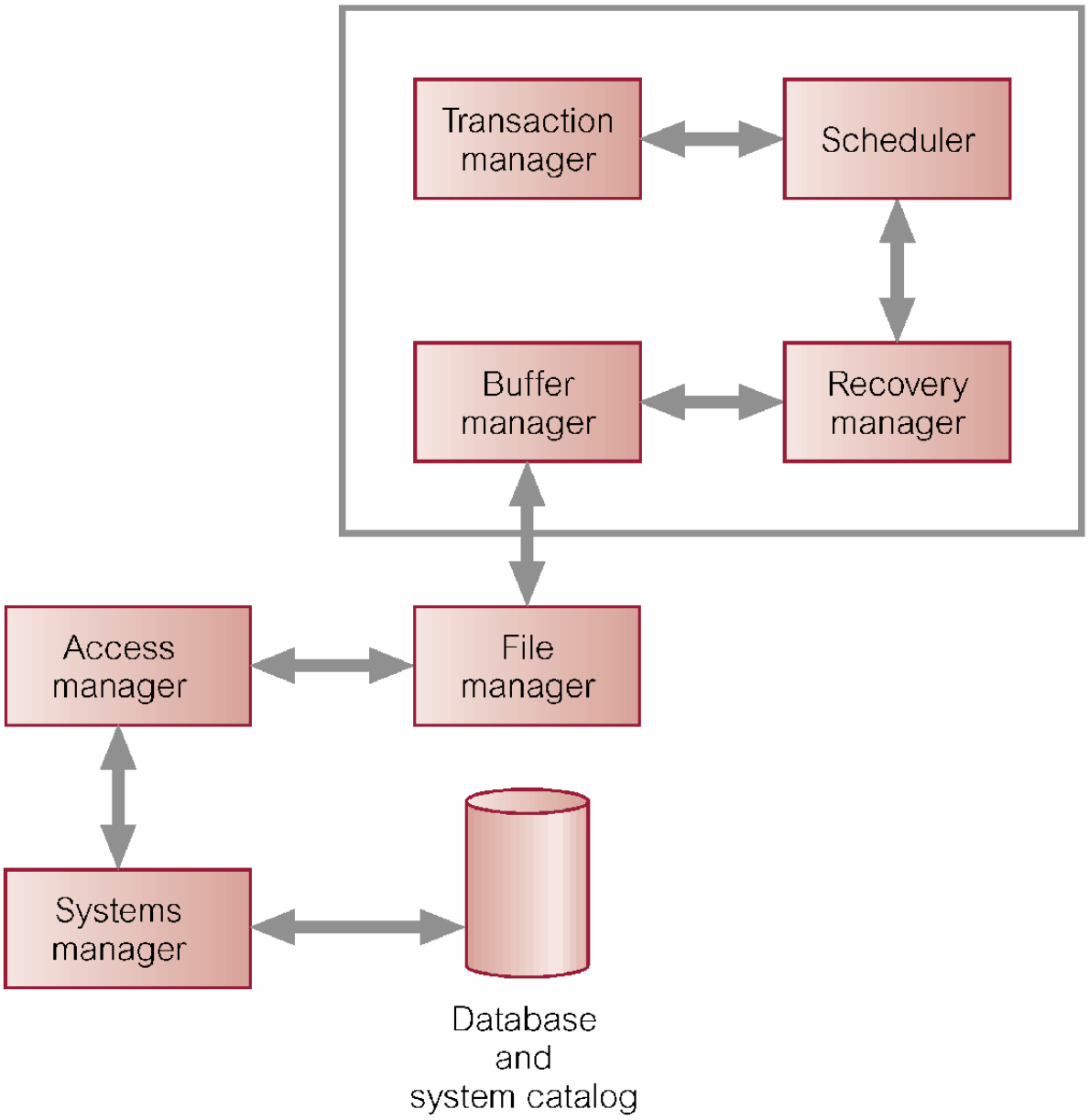
5. [abort.T]: Indicates that transaction T has been aborted.

Commit Point of a Transaction

- 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 have 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 a commit record [commit,T] into the log.
- If a system failure occurs, we search back in the log for all transactions T that have written a [start_transaction,T] record into the log but have not written their [commit,T] record yet; these transactions may have to be *rolled back* to undo their effect on the database during the recovery process. Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone*from the log records.
- The log file must be kept on disk. Updating a disk file involves copying the appropriate block of the file from disk to a buffer in main memory, updating the buffer in main memory, and copying the buffer to disk.
- It is common to keep one or more blocks of the log file in main memory buffers until they are filled with log entries and then to write them back to disk only once, rather than writing to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same log file block.
- At the time of a system crash, only the log entries that have been *written back* to *disk* are considered in the recovery process because the contents of main memory may be lost. Hence, *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction



Concurrency control:

Processes of managing simultaneous operations on the database without having them interfere with one another.

- Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.

- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Why Concurrency Control Is Needed

Concurrency control and recovery mechanisms are mainly concerned with the database access commands in a transaction. Transactions submitted by the various users mayexecute concurrently and may access and update the same database items. If this concurrent execution is uncontrolled, it may lead to problems, such as an inconsistent database. Several problems can occur when concurrent transactions execute in an uncontrolled manner.

These problems are

1. Lost update problem
2. The temporary update or Dirty Read Problem.
3. Incorrect summary problem.

The Lost Update Problem

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

Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in the figure a, then the final value of X is incorrect. Because T2 reads the value of X before T1 changes it in the database and hence the updated value resulting from T1 is lost.

For example, if X = 80 at the start, N =5 and M = 4 the final result should be X =79; but in the interleaving of operations shown in Figure a, it is X = 84 because the update in T1 that removed the five from X was *lost.*

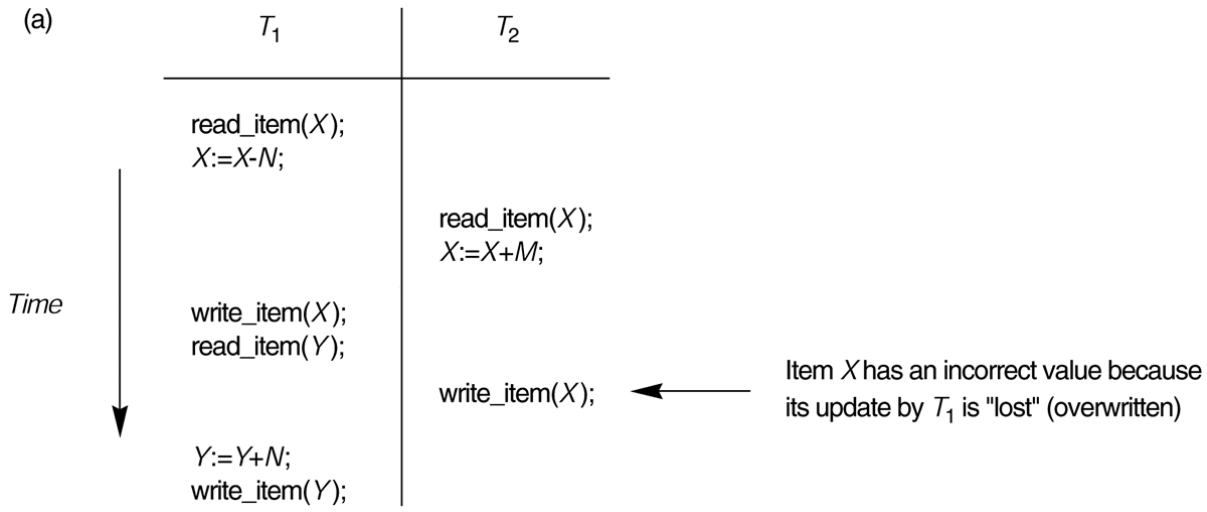
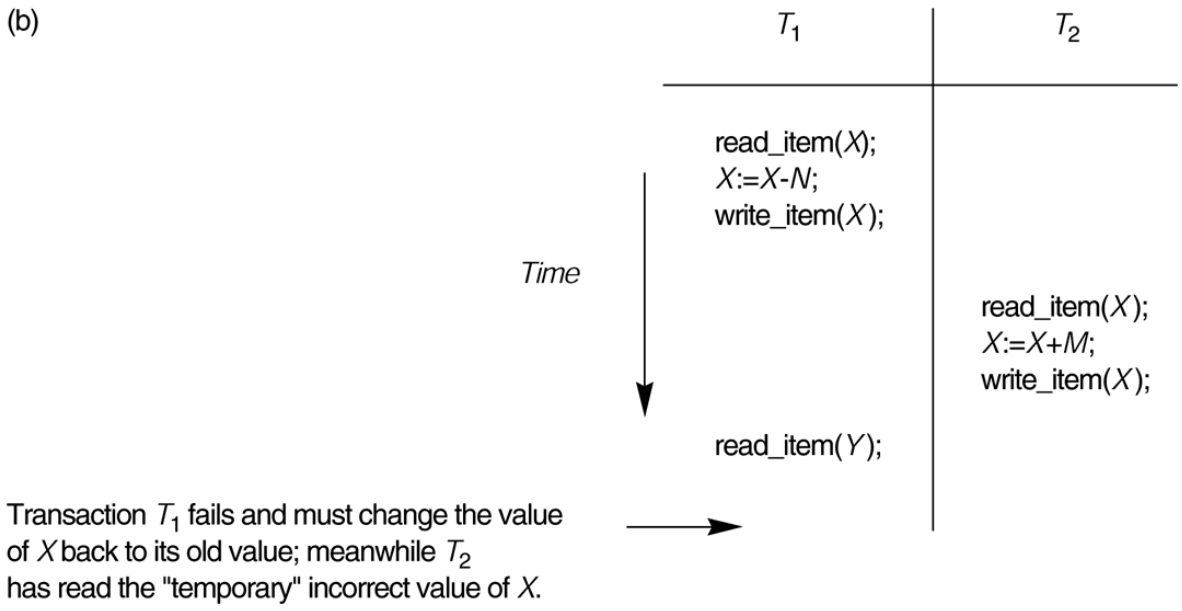


Fig a: The lost update problem.

The Temporary Update (or Dirty Read) Problem

This problem occurs when one transaction updates a database item and then the transaction fails for some reason.

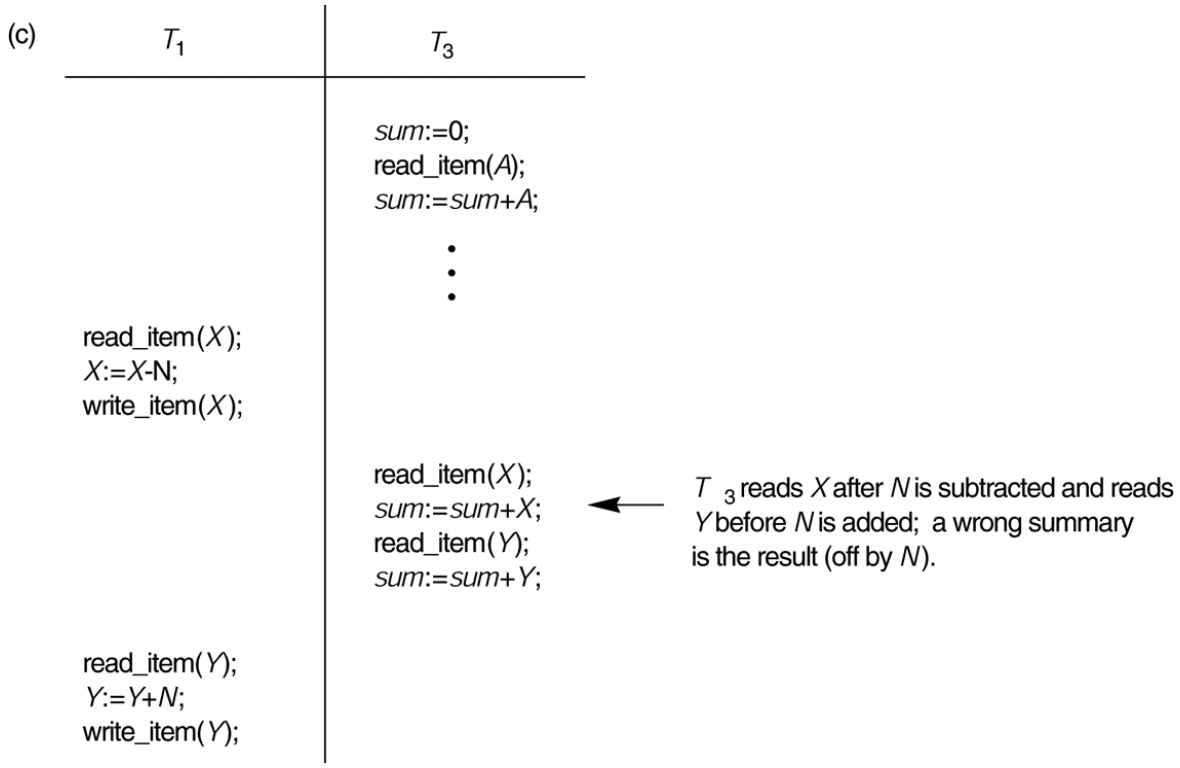
The updated item is accessed by another transaction before it is changed back to its original value. Figure b shows an example where T1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T r-The value of item X that is read by T2 is called *dirty data,* because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem.*



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. For example, suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1is executing. If the interleaving of operations shown in Figure c occurs, the result of *T3* will be off by an amount N because T3 reads the value of X *after* N seats have been Subtracted from it but reads the value of *Y before* those N seats have been added to it.

Another problem that may occur is called unrepeatable read, where a transaction T reads an item twice and the item is changed by another transaction T' between the two reads. Hence, T receives *different values* for its two reads of the same item.



Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either (1) all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or (2) the transaction has no effect whatsoever on the database or on any other transactions.

The DBMS must not permit some operations of a transaction *T* to be applied to the database while other operations of T are not.

This may happen if a transaction fails after executing some of its operations but before executing all of them.

Types of Failures

Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution

1. A *computer failure (system crash):* A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures-for example, main memory failure.

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.

3. Local errors* or *exception conditions detected by the transaction: During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found.

Notice that an exception condition," such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be cancelled.

This exception should be programmed in the transaction itself, and hence would not be considered a failure.

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.

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.

Schedule

When transactions are executing concurrently in an interleaved fashion, then the order Of execution of operations from the various transactions is known as a schedule.

A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed

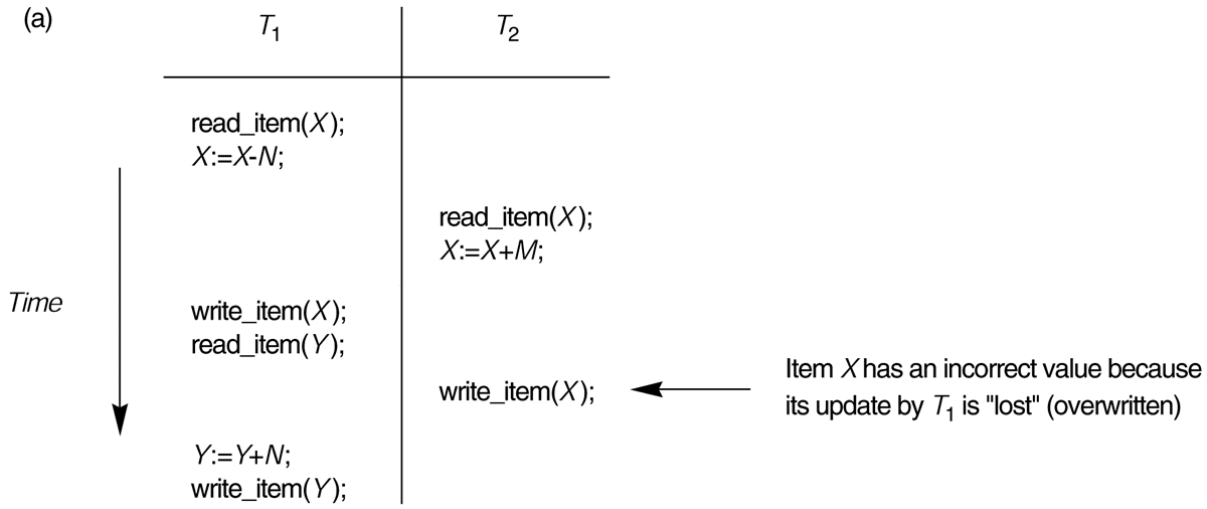
Schedules (Histories) of Transactions

A schedule (or history) S of n transactions T1 , T2, ... , Tn 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 *Ti,* in S must appear in the same order in which they occur in *Ti.*

That means a schedule for a set of transactions must consist of all instructions of those transactions.

For example, the schedule of below Figure which we shall call S_a can be written as follows in this notation:

S_a: r1(X); r2(X); *W1(X); r1(Y); ω2(X); W1(Y);*



Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:

(l) they belong to different transactions;

(2) they access the same item X; and

(3) at least one of the operations is a write_item(X).

For example, in schedule *Sa* the operations r l(X) and *w2(X)* , the operations r2(X)and W1(X), and the operations w1(X) and *W2(X) are conflict.* However, the operations r l(X) and *r2(X)* do not conflict, since they are both read operations; the operations *W2(X)* and W1 *(Y)* do not conflict, because they operate on distinct data items X and *Y;* and the operations rl(X)and *W1* (X) do not conflict, because they belong to the same transaction.

A schedule S of n transactions *T1, T2,* ••• , *Tn* is said to be a **complete schedule** if the following conditions hold:

1. The operations in S are exactly those operations in *T1,T2,* •.• , *Tn* including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction *Ti,* their order of appearance in S is the same as their order of appearance in *T;*
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

Characterizing Schedules Based on Recoverability

For some schedules it is easy to recover from transaction failures, whereas for other schedules the recovery process can be quite involved.

Hence, it is important to characterize the types of schedules for which recovery is possible, as well as those for which recovery is relatively simple. These characterizations do not actually provide the recovery algorithm but instead only attempt to theoretically characterize the different types of schedules.

Following are the schedules classified based on recoverability

1.Recoverable schedule
2.Cascadeless schedule
3.Strict schedules

1. **Recoverable schedule**

A schedule S is recoverable if no transaction in T1 in S commits until all Transactions of T2 that have written an item that T1 reads have committed.

A recoverable schedule is one where, for each pair of Transaction Ti and Tj such that Tj reads data item previously written by Ti then the commit operation of Ti appears before the commit operation Tj.

A transaction T1 reads from transaction T2 in a schedule S if some item X is first written by T2 and later read by T1.

Recoverable schedules require a complex recovery process

Sa: T1(X); T2(X); *W1*(X); *T1(Y); W2(X);* C2; *ω1(Y);* c1;

Sa is recoverable, even though it suffers from the lost update problem.

However,

consider the two (schedules Sc and Sd that follow:

Sc: rl(X); *W1*(X); r2(X); *r1(Y); ω2(X);* C2; al;

Sd: rl (X); *W1* (X); r2(X); rl*(Y); ω2(X); W1(Y);* C1; C2;

Se: rl(X); *W1*(X); T2(X); rl(Y); *W2(X); W1(Y);* a1; a2;

Sc is not recoverable, because T2 reads item X from T1 , and then T2 commits before *T1* commits. If TI aborts after the C2 operation in *Sc* then the value of X that T2 read is no longer valid and T2 must be aborted *after* it had been committed, leading to a schedule that is not recoverable. For the schedule to be recoverable, the C2 operation in Sc must be postponed until after T I commits, as shown in *Sd;* if T I aborts instead of committing, then

T2 should also abort as shown in *Se'* because the value of X it read is no longer valid.

2. **Cascadeless schedule**

A schedule is said to be cascadeless, if every transaction in the schedule reads only items that were written by committed transactions.
To satisfy this criterion, the *r2(X)* command in schedules *Sd* and *Se* must be postponed until after T1 has committed or aborted.

3. **Strict schedules**

   A schedule, called a strict schedule, in which transactions can *neither read nor write* an item X until the last transaction that wrote X has committed (or aborted). Strict schedules simplify the recovery process.

Schedules classified 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 non serial schedule.

Serializable schedule:

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

Assumption: Every serial schedule is correct

Goal: Like to find *non-serial* schedules which are also correct, because in serial schedules one transaction have to wait for another transaction to complete, Hence serial schedules are unacceptable in practice.

Result equivalent:

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

Problem: May produce same result by accident!

S1	S2
read_item(X);	read_item(X);
X:=X+10;	X:=X*1.1;
write_item(X);	write_item(X);

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'.
Can reorder the *non-conflicting* operations to improve efficiency

Non-conflicting operations:

- Reads and writes from same transaction
- Reads from different transactions
- Reads and writes from different transactions on different data items

Conflicting operations:

- Reads and writes from different transactions on same data item

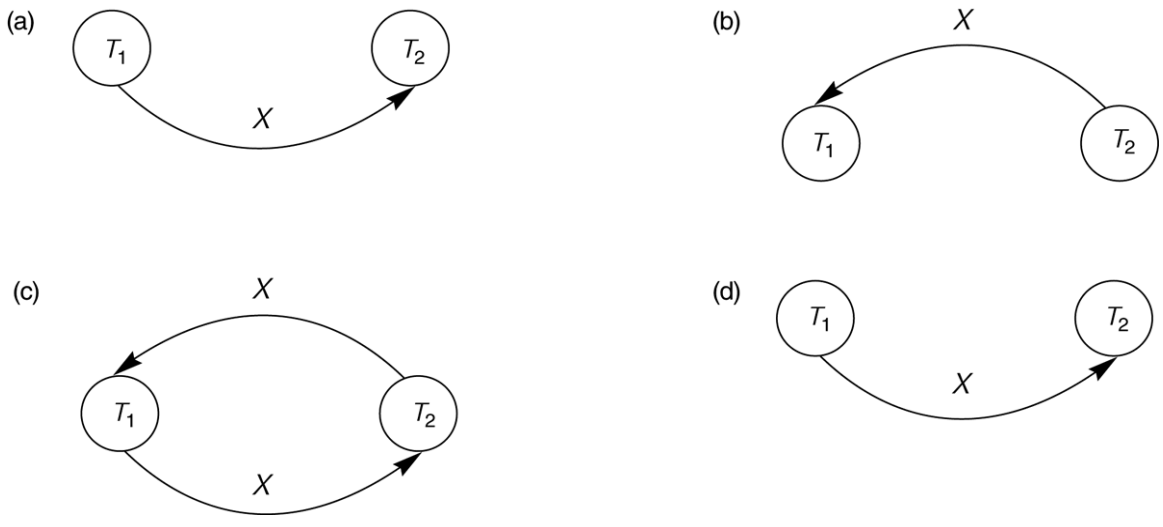
Test for Serializability:-

- Construct a directed graph, *precedence graph,* G = (V, E)
  - V: set of all transactions participating in schedule

- E: set of edges Ti → Tj for which one of the following holds:
  - Ti executes a write_item(X) before Tj executes read_item(X)
  - Ti executes a read_item(X) before Tj executes write_item(X)
  - Ti executes a write_item(X) before Tj executes write_item(X)
- An edge Ti → Tj means that in any serial schedule equivalent to S, Ti must come before Tj
- If G has a cycle, than S is not conflict serializable
- If not, use topological sort to obtain serialiazable schedule (linear order consistent with precedence order of graph)

FIGURE 17.7 Constructing the precedence graphs for schedules A and D from Figure 17.5 to test for conflict serializability.

- Precedence graph for serial schedule A.
- Precedence graph for serial schedule B.
- Precedence graph for schedule C (not serializable).
- Precedence graph for schedule D (serializable, equivalent to schedule A).



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.

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'.

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 musr produce the same results.
- **"The view"**: the read operations are said to see the *the same view* in both schedules.

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 serialiable.
- Any conflict serializable schedule is also view serializable, but not vice versa.

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.

Introduction to Concurrency

What is concurrency?

Concurrency in terms of databases means allowing multiple users to access the data contained within a database at the same time. If concurrent access is not managed by the Database Management System (DBMS) so that simultaneous operations don't interfere with one another problems can occur when various transactions interleave, resulting in an inconsistent database.

Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions. Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins. Concurrent execution of user programs is essential for good DBMS performance. Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently. Interleaving actions of different user programs can lead to inconsistency: e.g., check is cleared while account balance is being computed. DBMS ensures such problems don't arise: users can pretend they are using a single-user system.

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.

LOCK

Definition : Lock is a variable associated with data item which gives the status whether the possible operations can be applied on it or not.

Two-Phase Locking Techniques:

Binary locks: Locked/unlocked

The simplest kind of lock is a binary on/off lock. This can be created by storing a lock bit with each database item. If the lock bit is on (e.g. = 1) then the item cannot be accessed by any transaction either for reading or writing, if it is off (e.g. = 0) then the item is available. Enforces mutual exclusion

Binary locks are:

- Simple but are restrictive.
- Transactions must lock every data item that is read or written
- No data item can be accessed concurrently

  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.
- 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 . One simple way to implement a lock table is through linked list.

< locking_transaction ,data item, LOCK >

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;
```

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;
```

Multiple-mode locks: Read/write

- a.k.a. Shared/Exclusive

- Three operations
  - read_lock(X)
  - write_lock(X)
  - unlock(X)
- Each data item can be in one of three lock states
  - Three locks modes:
    - (a) shared (read) (b) exclusive (write) (c) unlock(release)
  - **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.

_ **Unlock mode**: Unlock(X)

After reading or writing the corresponding transaction releases by issuing this.

The rules for multiple-mode locking schemes are a transaction T:

- Issue a **read_lock(X)** or a **write_lock(X)** before **read(X)**
- Issue a **write_lock(X)** before **write(X)**
- Issue an **unlock(X)** after all **read(X)** and **write(X)** are finished

The transaction T

- Will not issue **read_lock (X)** or **write_lock(X)** if it already holds a lock on **X**
- Will not issue **unlock(X)** unless it already holds a lock on X

Lock table:

Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and no of transactions that are currently reading the data item . It looks like as below

<data item,read_ LOCK,nooftransactions,transaction id >

This protocol isn't enough to guarantee serializability. If locks are released too early, you can create problems. This usually happens when a lock is released before another lock is acquired.

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;
```

The following code performs the **write lock operation**:

```
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;
```

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;
```

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)

Two-Phase Locking Techniques: The algorithm

The timing of locks is also important in avoiding concurrency problems. A simple requirement to ensure transactions are serializable is that all read and write locks in a transaction are issued before the first unlock operation known as a two-phase locking protocol.

Transaction divided into 2 phases:

- *growing* - new locks acquired but none released
- *shrinking* - existing locks released but no new ones acquired

During the shrinking phase no new locks can be acquired!

- Downgrading ok
- Upgrading is not

Rules of 2PL are as follows:

- If T wants to read an object it needs a read_lock
- If T wants to write an object, it needs a write_lock
- Once a lock is released, no new ones can be acquired.

The 2PL protocol guarantees serializability

- Any schedule of transactions that follow 2PL will be serializable
- We therefore do not need to test a schedule for serializability

But, it may limit the amount of concurrency since transactions may have to hold onto locks longer than needed, creating the new problem of deadlocks.

Two-Phase Locking Techniques: The algorithm

Here is a example without 2PL:-

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);	

T1	**T2**	**Result**
read_lock (Y);		X=50; Y=50
read_item (Y);		Nonserializable because it.
unlock (Y);		violated two-phase policy.
	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);		

Here is a example with 2PL:-

T'1	**T'2**	**Result**

read_lock (Y);	read_lock (X);	T1 and T2 follow two-phase
read_item (Y);	read_item (X);	policy but they are subject to
write_lock (X);	Write_lock (Y);	deadlock, which must be
unlock (Y);	unlock (X);	dealt with.
read_item (X);	read_item (Y);	
X:=X+Y;	Y:=X+Y;	
write_item (X);	write_item (Y);	
unlock (X);	unlock (Y);	

Two-phase policy generates four locking algorithms:-

1. BASIC
2. CONSERVATIVE
3. STRICT
4. RIGOUROUS

- Previous technique knows as *basic 2PL*
- ***Conservative* 2PL (static) 2PL**: Lock all items needed BEFORE execution begins by predeclaring its read and write set
  - If any of the items in read or write set is already locked (by other transactions), transaction waits (does not acquire any locks)
  - Deadlock free but not very realistic
- ***Strict* 2PL**: Transaction does not release its write locks until AFTER it aborts/commits

  - Not deadlock free but guarantees recoverable schedules (strict schedule: transaction can neither read/write X until last transaction that wrote X has committed/aborted)
  - Most popular variation of 2PL
- ***Rigorous* 2PL**: No lock is released until after abort/commit
  - Transaction is in its expanding phase until it ends

ADBMS

Unit 2

Transaction processing and Concurrency control

Transaction

- The transaction is a set of logically related operation. It contains a group of tasks.
- A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

Example: Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

X's Account

1. Open_Account(X)
2. Old_Balance = X.balance
3. New_Balance = Old_Balance - 800
4. X.balance = New_Balance
5. Close_Account(X)

Y's Account

1. Open_Account(Y)
2. Old_Balance = Y.balance
3. New_Balance = Old_Balance + 800
4. Y.balance = New_Balance
5. Close_Account(Y)

Operations of Transaction:

Read(X): Read operation is used to read the value of X from the database and stores it in a buffer in main memory.

Write(X): Write operation is used to write the value back to the database from the buffer.

Let's take an example to debit transaction from an account which consists of following operations:

1. 1. R(X);
2. 2. X = X - 500;
3. 3. W(X);

Let's assume the value of X before starting of the transaction is 4000.

Following are the main operations of transaction:

- he first operation reads X's value from database and stores it in a buffer.
- The second operation will decrease the value of X by 500. So buffer will contain 3500.
- The third operation will write the buffer's value to the database. So X's final value will be 3500.

But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.

For example: If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

To solve this problem, we have two important operations:

Commit: It is used to save the work done permanently.

Rollback: It is used to undo the work done.

ACID Properties

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.

Property of Transaction

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

Atomicity

- It states that all operations of the transaction take place at once if not, the transaction is aborted.
- There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

Atomicity involves the following two operations:

Abort: If a transaction aborts then all the changes made are not visible.

Commit: If a transaction commits then all the changes made are visible.

Example: Let's assume that following transaction T consisting of T1 and T2. A consists of Rs 600 and B consists of Rs 300. Transfer Rs 100 from account A to account B.

T1	T2
Read(A) A:= A-100 Write(A)	Read(B) Y:= Write(B)

After completion of the transaction, A consists of Rs 500 and B consists of Rs 400.

If the transaction T fails after the completion of transaction T1 but before completion of transaction T2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed in entirety.

Consistency

- The integrity constraints are maintained so that the database is consistent before and after the transaction.
- The execution of a transaction will leave a database in either its prior stable state or a new stable state.
- The consistent property of database states that every transaction sees a consistent database instance.
- The transaction is used to transform the database from one consistent state to another consistent state.

For example: The total amount must be maintained before or after the transaction.

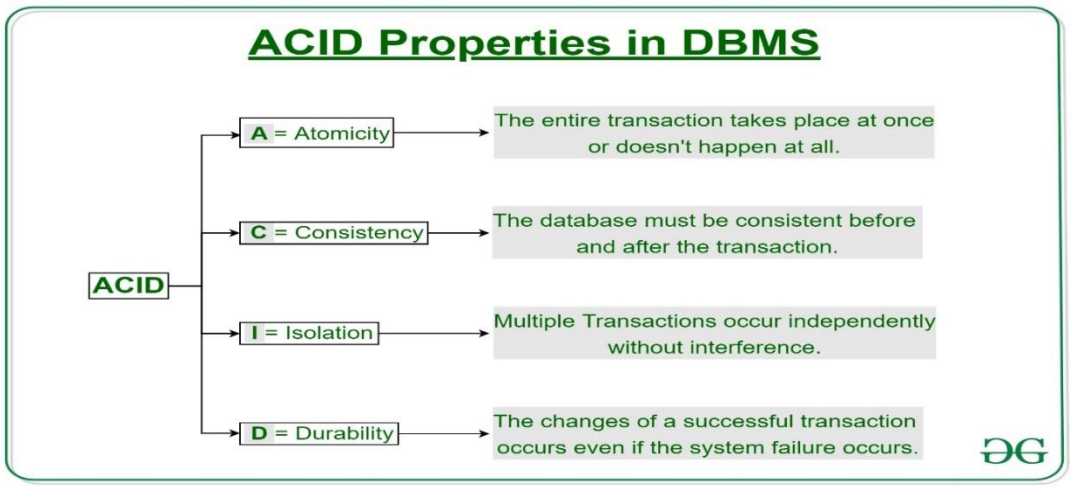
1. Total before T occurs = 600+300=900
2. Total after T occurs= 500+400=900
   Therefore, the database is consistent. In the case when T1 is completed but T2 fails, then inconsistency will occur.

Isolation

- It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
- In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
- The concurrency control subsystem of the DBMS enforced the isolation property.

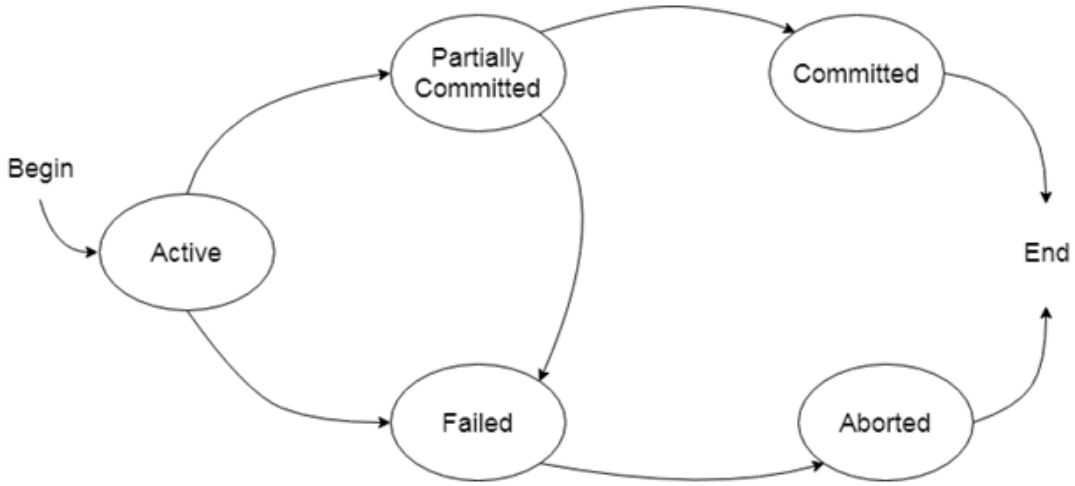
Durability

- The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
- They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
- The recovery subsystem of the DBMS has the responsibility of Durability property.



States of Transaction

The different stages a transaction goes through during its lifecycle are known as the transaction states. The following is a diagrammatic representation of the different stages of a transaction.



Active state

- The active state is the first state of every transaction. In this state, the transaction is being executed.
- For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

Partially committed

- In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
- In the total mark calculation example, a final display of the total marks step is executed in this state.

Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

Failed state

- If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
- In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

Aborted

- If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
- If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
- After aborting the transaction, the database recovery module will select one of the two operations:
    1. Re-start the transaction
    2. Kill the transaction

Example

Let us take a very simple example of Railway ticket booking. Can you think of the things that need to be retrieved from the database when you initiate the booking process?

You will need the train details, the already booked ticket details, the platform details, and many more such things. Now, once these details are retrieved the transaction of booking a ticket enters the **active state**.

After the user has completed the entire process of booking a ticket from their end, the transaction enters the **partially committed state**. In case any error occurred during the process, then the transaction will enter the **failed state**.

Now, say the process was successful and the transaction entered the partially committed state, now if the saving in the database is completed successfully then the transaction enters the **committed state**. In case there is any error while saving in the database then it enters the **failed state**.

Anything from the failed state enters the **aborted state** so that rollbacks can take place and the database consistency is maintained.

Now, let's talk about the terminated state. If the booking is permanently saved in the database, or it has been aborted due to some unforeseen reasons then the transaction enters the **terminated state**.

Serializability

- In the field of computer science, serializability is a term that is a property of the system that describes how the different process operates the shared data.
- If the result given by the system is similar to the operation performed by the system, then in this situation, we call that system serializable.
- Here the cooperation of the system means there is no overlapping in the execution of the data. In DBMS, when the data is being written or read then, the DBMS can stop all the other processes from accessing the data.
- A schedule is serialized if it is equivalent to a serial schedule. A concurrent schedule must ensure it is the same as if executed serially means one after another. It refers to the sequence of actions such as read, write, abort, commit are performed in a serial manner.

Schedules in DBMS are of two types:

1. **Serial Schedule** - A schedule in which only one transaction is executed at a time, i.e., one transaction is executed completely before starting another transaction.

Example:

Transaction-1	Transaction-2
R(a)	
W(a)	
R(b)	
W(b)	
	R(b)
	W(b)
	R(a)
	W(a)

2. Here, we can see that Transaction-2 starts its execution after the completion of Transaction-1.

2. Non serial schedule − When a transaction is overlapped between the transaction T1 and T2.

Example:

Transaction-1	Transaction-2
R(a)	
W(a)	
	R(b)
	W(b)
R(b)	
	R(a)
W(b)	
	W(a)

We can see that Transaction-2 starts its execution before the completion of Transaction-1, and they are interchangeably working on the same data, i.e., "a" and "b".

Types of serializability

There are two types of serializability −

1. Conflict serializability

Conflict serializability is a type of conflict operation in serializability that operates the same data item that should be executed in a particular order and maintains the consistency of the database. In DBMS, each transaction has some unique value, and every transaction of the database is based on that unique value of the database.

This unique value ensures that no two operations having the same conflict value are executed concurrently. For example, let's consider two examples, i.e., the order table and the customer table. One customer can have multiple orders, but each order only belongs to one customer. There is some condition for the conflict serializability of the database. These are as below.

- Both operations should have different transactions.
- Both transactions should have the same data item.

- There should be at least one write operation between the two operations.

If there are two transactions that are executed concurrently, one operation has to add the transaction of the first customer, and another operation has added by the second operation. This process ensures that there would be no inconsistency in the database.

The conflicting pairs are:

1. READ(a) - WRITE(a)
2. WRITE(a) - WRITE(a)
3. WRITE(a) - READ(a)

2. View serializability

If a non-serial schedule is **view equivalent** to some other serial schedule then the schedule is called View Serializable Schedule. It is needed to ensure the consistency of a schedule.

What is view equivalency?

The two conditions needed by schedules(S1 and S2) to be view equivalent are:

1. **Initial read** must be on the same piece of data.

Example: If transaction t1 is reading "A" from database in schedule S1, then in schedule S2, t1 must read A.

2. **Final write** must be on the same piece of data.

Example: If a transaction t1 updated A at last in S1, then in S2, t1 should perform final write as well.

3. The mid sequence should also be in the same order.

Example: If t1 is reading A which is updated by t2 in S1, then in S2, t1 should read A which should be updated by t2.

This process of checking view equivalency of a schedule is called View Serializability.

Example: We have a schedule "S" having two transactions t1, and t2 working simultaneously.

S:

t1	t2
R(x)	
W(x)	

t1	t2
	R(x)
	W(x)
R(y)	
W(y)	
	R(y)
	W(y)

Let's form its view equivalent schedule (S') by interchanging mid-read-write operations of both the transactions. **S':**

t1	t2
R(x)	
W(x)	
R(y)	
W(y)	
	R(x)
	W(x)
	R(y)
	W(y)

Since a view equivalent schedule is possible, it is a view serializable schedule.

Prioritization

Prioritization is useful for browsing tasks, and tasks that use a lot of processor time. Input/Output bound tasks can take the required amount of CPU, and move on to the next read/write wait. CPU-intensive tasks take higher priority over the less intensive tasks. Prioritization can be implemented in all CICS® systems. It is more important in a high-activity system than in a low-activity system. With careful priority selection, you can improve overall throughput and response time. Prioritization can minimize resource usage of certain resource-bound transactions. Prioritization increases the response time for lower-priority tasks, and can distort the regulating effects of MXT and the MAXACTIVE attribute of the transaction class definition.

Priorities do not affect the order of servicing terminal input messages and, therefore, the time they wait to be attached to the transaction manager. Because

prioritization is determined in three sets of definitions (terminal, transaction, and operator), it can be a time-consuming process for you to track many transactions in a system. CICS prioritization is not interrupt-driven as is the case with operating system prioritization, but determines the position on a ready queue. This means that, after a task is given control of the processor, the task does not relinquish that control until it issues a CICS command that calls the CICS dispatcher. After the dispatch of a processor-bound task, CICS can be tied up for long periods if CICS requests are infrequent. For that reason, prioritization should be implemented only if MXT and the MAXACTIVE attribute of the transaction class definition adjustments have proved to be insufficient.

You should use prioritization sparingly, if at all, and only after you have already adjusted task levels using MXT and the MAXACTIVE attribute of the transaction class definition. It is probably best to set all tasks to the same priority, and then prioritize some transactions either higher or lower on an exception basis, and according to the specific constraints in a system. Do not prioritize against slow tasks unless you can accept the longer task life and greater dispatch overhead; these tasks are slow, in any case, and give up control each time they have to wait for I/O. Use small priority values and differences and concentrate on transaction priority. Give priority to control operator tasks rather than the person, or at least to the control operator's signon ID rather than to a specific physical terminal (the control operator may move around).

Consider for high priority a task that uses large resources. However, the effects of this on the overall system need careful monitoring to ensure that loading a large transaction of this type does not lock out other transactions. Also consider for high priority those transactions that cause enqueues to system resources, thus locking out other transactions. As a result, these can process quickly and then release resources. Here are some examples:

- Using intrapartition transient data with logical recovery
- Updating frequently used records
- Automatic logging
- Tasks needing fast application response time, for example, data entry.

Lower priority should be considered for tasks that:

- Have long browsing activity
- Are process-intensive with minimal I/O activity
- Do not require terminal interaction, for example:
    - Auto-initiate tasks (except when you use transient data intrapartition queues that have a destination of terminal defined and a trigger level that is greater than zero).

- Batch update controlling tasks.

There is no direct measurement of transaction priority. Indirect measurement can be made from:

- Task priorities
- Observed transaction responses
- Overall processor, storage, and data set I/O usage.

The following table shows how you might prioritize three typical transactions.

- Establish a HIGH priority for transactions that are vital to the operations of the organization. For example, you might specify a HIGH priority for a transaction that services the information needs of upper-level managers in the organization.

Sample Transactions

Transaction	Processing Mode	Time	Priority
Add or delete a claim	Online	3 seconds	High
List of employees for an office	Batch	15 minutes	Medium
Show salary grade for all jobs	Online	6 seconds	Low

Failure Classification

To find that where the problem has occurred, we generalize a failure into the following categories:

1. Transaction failure
2. System crash
3. Disk failure

1. Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

1. **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.

2. **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. **For example,** The system aborts an active transaction, in case of deadlock or resource unavailability.

2. System Crash

- System failure can occur due to power failure or other hardware or software failure. **Example:** Operating system error.

  Fail-stop assumption: In the system crash, non-volatile storage is assumed not to be corrupted.

3. Disk Failure

- It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
- Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

Recoverability of Schedule

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		
Failure Point				
Commit;				

The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

Irrecoverable schedule: The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
Failure Point				
Commit;				
		Commit;		

The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

Recoverable with cascading rollback: The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
Commit;		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		

The above Table 3 shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

Levels of Transaction Consistency

In a Distributed System, consistency means that ***"all reads executed at a given time must return the same result irrespective of the server that executed the read***." For one server to read the data written by another server, we need to maintain a global ordering of reads and writes. So that each copy of the database will execute the events in the same order, and they will all result in the same final database state. Therefore, our data will be consistent.

The consistency levels differ in the way they implement this ordering of events. Some are strict, w.r.t maintaining a time-based order while others are more relaxed.

Consistency Levels

Sequential Consistency: All writes must be globally ordered. Each thread of execution must see the same ordering of writes, irrespective of which thread executed and which data items were written to. However, this ordering needn't

be the same as the real-time ordering of writes. We can have any ordering of writes, as long as such an ordering is agreed upon by all the threads.

W: A = 1				
	W: B = 2			
		R: B = 2	R: A = 0	R:A = 1
		R: B = 2	R: A = 1	

Figure 1: sequentially consistent

Threads 3 and 4 see that B has been updated before A (which is different from what actually happened), but this is still considered sequentially consistent as all the threads are in agreement.

W: A = 1				
	W: B = 2			
		R: B = 2	R: A = 0	R:A = 1
		R: B = 0	R: A = 1	R: B = 2

Figure 2: Not sequentially consistent

In the above image, thread 3 sees the update to B before the update to A, while thread 4 sees the update to A first. This violates sequential consistency.

Strict Consistency: This is the highest level of consistency. Strict Consistency requires events to be ordered in the same real-time sequence in which they occurred. In other words, ***"an earlier write always has to be seen before a later write."*** This is almost impossible to implement in distributed systems. Hence it remains in the realm of theoretical discussions.

Strict Consistency also demands that any write by a given thread should be immediately visible to all other threads. **Hence the above two images do not satisfy strict consistency, as they read zero values from data items that were already written to**.

W: A = 1			
	W: B = 2		
		R: B = 2	R: A = 1
		R: A = 1	R: B = 2

Figure 3: Strictly Consistent

The above image satisfies strict consistency as immediate subsequent reads see the new value of the data item.

Linearizability/ Atomic Consistency: Although strict consistency is very hard to achieve, we can come pretty close to it. Linearizability also requires the writes to be ordered in a real-time fashion, but it acknowledges that there's some time gap between when an operation is submitted to the system and when the system acknowledges it.

1	W: A = 1			
2		W: B = 2		
3	R: A = 0		R: B = 2	
4			R: A = 1	R: B = 2

Figure 4: Linearizable but not strictly consistent

The write of thread 1 overlaps with the read of thread 3. Therefore, there needn't be a real-time ordering between them. Hence the above image satisfies Linearizability even though thread three's read doesn't fetch the latest value.

Causal Consistency: ***Causal Consistency requires only related operations to have a global ordering between them.*** Two operations can be related because they acted on the same data item, or because they originated from the same thread. Operations that are not related can be seen in any order by any thread. Causal consistency is a weaker form of sequential consistency.

W: A = 1					
	R: A = 1	W: B = 2			
			R: A = 1	R: B = 0	R: B = 2
			R: B = 2	R: A = 1	

Figure 5: Causally consistent

The write to B by thread 2 follows the read from A. Hence these two operations are causally related. Hence, **the write to A must appear before the write to B**. Figure 5 is causally consistent since threads 3 and 4 see the writes in the same order.

Eventual Consistency: This can be called the weakest level of consistency. The only guarantee offered here is that **if there are no writes for a long period of time, the threads will eventually agree on the value of the last write.** That is, eventually, all the copies of the database will reflect the same value.

W: A = 1					
	R: A = 1	W: B = 2			
			R: A = 1	R: B = 0	R: B = 2
			R: B = 2	R: A = 0	

Figure 6: Eventually consistent

The above figure violates all consistency levels except eventual consistency. If there isn't any other write for a long time, all the threads will see the same value of A and B.

Deadlock

A deadlock is a condition where two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as no task ever gets finished and is in waiting state forever.

For example: In the student table, transaction T1 holds a lock on some rows and needs to update some rows in the grade table. Simultaneously, transaction T2 holds locks on some rows in the grade table and needs to update the rows in the Student table held by Transaction T1.

Now, the main problem arises. Now Transaction T1 is waiting for T2 to release its lock and similarly, transaction T2 is waiting for T1 to release its lock. All activities come to a halt state and remain at a standstill. It will remain in a standstill until the DBMS detects the deadlock and aborts one of the transactions.

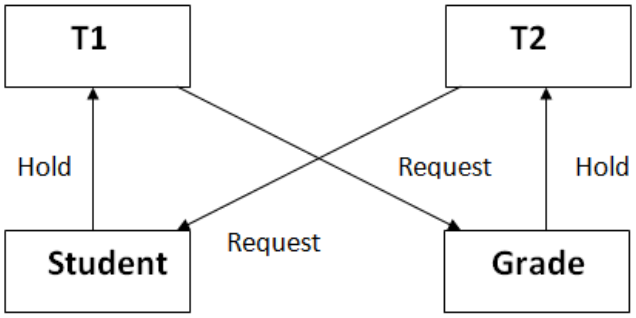


Figure: Deadlock in DBMS

Deadlock Avoidance

- When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restating the database. This is a waste of time and resource.

- Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

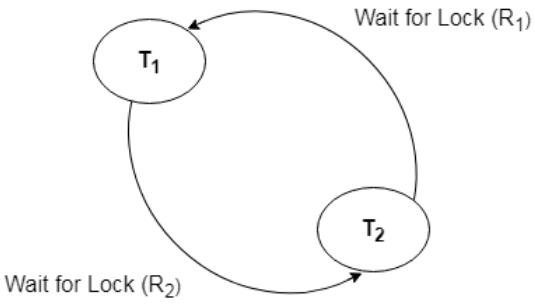
Deadlock Detection

In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait for the graph to detect the deadlock cycle in the database.

Wait for Graph

- This is the suitable method for deadlock detection. In this method, a graph is created based on the transaction and their lock. If the created graph has a cycle or closed loop, then there is a deadlock.
- The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

The wait for a graph for the above scenario is shown below:



Deadlock Prevention

- Deadlock prevention method is suitable for a large database. If the resources are allocated in such a way that deadlock never occurs, then the deadlock can be prevented.

- The Database management system analyzes the operations of the transaction whether they can create a deadlock situation or not. If they do, then the DBMS never allowed that transaction to be executed.

Wait-Die scheme

In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It allows the older transaction to wait until the resource is available for execution.

Let's assume there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS:

1. Check if TS(Ti) < TS(Tj) - If Ti is the older transaction and Tj has held some resource, then Ti is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.
2. Check if TS(T_i) < TS(Tj) - If Ti is older transaction and has held some resource and if Tj is waiting for it, then Tj is killed and restarted later with the random delay but with the same timestamp.

Wound wait scheme

- In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.
- If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.

Features of deadlock in a DBMS:

- **Mutual Exclusion:** Each resource can be held by only one transaction at a time, and other transactions must wait for it to be released.

- **Hold and Wait:** Transactions can request resources while holding on to resources already allocated to them.

- **No Preemption:** Resources cannot be taken away from a transaction forcibly, and the transaction must release them voluntarily.

- **Circular Wait:** Transactions are waiting for resources in a circular chain, where each transaction is waiting for a resource held by the next transaction in the chain.

- **Indefinite Blocking:** Transactions are blocked indefinitely, waiting for resources to become available, and no transaction can proceed.

- **System Stagnation:** Deadlock leads to system stagnation, where no transaction can proceed, and the system is unable to make any progress.

- **Inconsistent Data:** Deadlock can lead to inconsistent data if transactions are unable to complete and leave the database in an intermediate state.

- **Difficult to Detect and Resolve:** Deadlock can be difficult to detect and resolve, as it may involve multiple transactions, resources, and dependencies.

Disadvantages:

- **System downtime:** Deadlock can cause system downtime, which can result in loss of productivity and revenue for businesses that rely on the DBMS.
- **Resource waste:** When transactions are waiting for resources, these resources are not being used, leading to wasted resources and decreased system efficiency.
- **Reduced concurrency:** Deadlock can lead to a decrease in system concurrency, which can result in slower transaction processing and reduced throughput.
- **Complex resolution:** Resolving deadlock can be a complex and time-consuming process, requiring system administrators to intervene and manually resolve the deadlock.
- **Increased system overhead:** The mechanisms used to detect and resolve deadlock, such as timeouts and rollbacks, can increase system overhead, leading to decreased performance.

Transaction processing as implemented in contemporary databases

Transaction processing in contemporary databases refers to the management of database transactions in a way that ensures data integrity, consistency, and reliability. It is a critical aspect of database management systems (DBMS) and is essential in various applications, including e-commerce, banking, healthcare, and more. Here are some key aspects of transaction processing as implemented in contemporary databases:

ACID Properties: Contemporary databases typically adhere to the ACID (Atomicity, Consistency, Isolation, Durability) properties to guarantee the reliability of transactions.

Atomicity: A transaction is treated as a single, indivisible unit. It is either entirely completed or entirely aborted. There are no partial changes to the database.

Consistency: Transactions bring the database from one consistent state to another. They must follow predefined rules and constraints to maintain data integrity.

Isolation: Transactions are executed in isolation from each other. This means that the changes made by one transaction are not visible to others until the transaction is completed. Isolation levels like Read Uncommitted, Read Committed, Repeatable Read, and Serializable control the degree of isolation.

Durability: Once a transaction is committed, its changes are permanent and will survive system failures. This ensures data is not lost in the event of a crash.

Concurrency Control: Contemporary databases employ various techniques to manage concurrent access to the database by multiple transactions. This includes locking, multi-version concurrency control, and timestamp-based ordering, among others.

Transaction Logs: Databases maintain transaction logs to record all changes made by transactions. These logs are crucial for recovery in the event of system failures.

Commit and Rollback: Transactions provide mechanisms for committing changes to the database or rolling back changes in case of errors or user requests.

Savepoints: Some contemporary databases support savepoints within transactions. This allows a transaction to be partially rolled back to a specific point, which can be useful in error recovery.

Two-Phase Commit (2PC): In distributed databases, the two-phase commit protocol is often used to ensure that transactions are either committed on all distributed nodes or rolled back in a coordinated manner.

Optimistic Concurrency Control: In some scenarios, databases use optimistic concurrency control, where conflicts are detected and resolved only when committing a transaction. This can reduce contention but may require more complex conflict resolution mechanisms.

Snapshot Isolation: Some databases support snapshot isolation, which provides a consistent view of the database as of the start of a transaction. This can simplify concurrency control and improve performance.

Deadlock Detection and Resolution: Databases incorporate mechanisms to detect and resolve deadlocks, which occur when transactions are waiting for each other to release locks.

Distributed Transactions: In distributed databases, managing transactions across multiple nodes and ensuring they maintain ACID properties can be more complex. Distributed transaction managers are used to coordinate such transactions.

In-Memory Databases: Some contemporary databases are designed to work entirely or partially in memory, offering high-speed transaction processing.

NoSQL Databases: While traditional SQL databases follow ACID properties, some NoSQL databases may relax these constraints to provide better scalability and performance, opting for BASE (Basically Available, Soft state, Eventually consistent) properties instead.

Contemporary databases, whether SQL or NoSQL, are designed to meet specific application requirements while ensuring data consistency and reliability. The choice of database system and transaction processing methods depends on the specific needs of an application, such as performance, scalability, and data integrity.

Concurrency Control

Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.

Concurrent Execution in DBMS

- In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.
- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
- The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

Problems with Concurrent Execution

In a database transaction, the two main operations are **READ** and **WRITE** operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

Problem 1: Lost Update Problems (W - W Conflict)

The problem occurs *when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent*.

For example:

Consider the below diagram where two transactions T_X and T_Y, are performed on the same account A where the balance of account A is $300.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

- At time t1, transaction T_X reads the value of account A, i.e., $300 (only read).
- At time t2, transaction T_X deducts $50 from account A that becomes $250 (only deducted and not updated/write).
- Alternately, at time t3, transaction T_Y reads the value of account A that will be $300 only because T_X didn't update the value yet.
- At time t4, transaction T_Y adds $100 to account A that becomes $400 (only added but not updated/write).
- At time t6, transaction T_X writes the value of account A that will be updated as $250 only, as T_Y didn't update the value yet.
- Similarly, at time t7, transaction T_Y writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by T_X is lost, i.e., $250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

Dirty Read Problems (W-R Conflict)

The dirty read problem occurs *when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.*

For example:

Consider two transactions T_X and T_Y in the below diagram performing read/write operations on account A where the available balance in account A is $300:

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

- At time t1, transaction T_X reads the value of account A, i.e., $300.
- At time t2, transaction T_X adds $50 to account A that becomes $350.
- At time t3, transaction T_X writes the updated value in account A, i.e., $350.
- Then at time t4, transaction T_Y reads account A that will be read as $350.
- Then at time t5, transaction T_X rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction T_Y as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Unrepeatable Read Problem (W-R Conflict)

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

For example:

Consider two transactions, T_X and T_Y, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

Time	T_X	T_Y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

- At time t1, transaction T_X reads the value from account A, i.e., $300.
- At time t2, transaction T_Y reads the value from account A, i.e., $300.
- At time t3, transaction T_Y updates the value of account A by adding $100 to the available balance, and then it becomes $400.
- At time t4, transaction T_Y writes the updated value, i.e., $400.
- After that, at time t5, transaction T_X reads the available value of account A, and that will be read as $400.
- It means that within the same transaction T_X, it reads two different values of account A, i.e., $ 300 initially, and after updation made by transaction T_Y, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role

Concurrency Control Protocols

The concurrency control protocols ensure the *atomicity, consistency, isolation, durability* and *serializability* of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

- Lock Based Concurrency Control Protocol

- Time Stamp Concurrency Control Protocol
- Validation Based Concurrency Control Protocol

Lock-Based Protocol

- In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

1. Shared lock:

- It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

2. Exclusive lock:

- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

There are four types of lock protocols available:

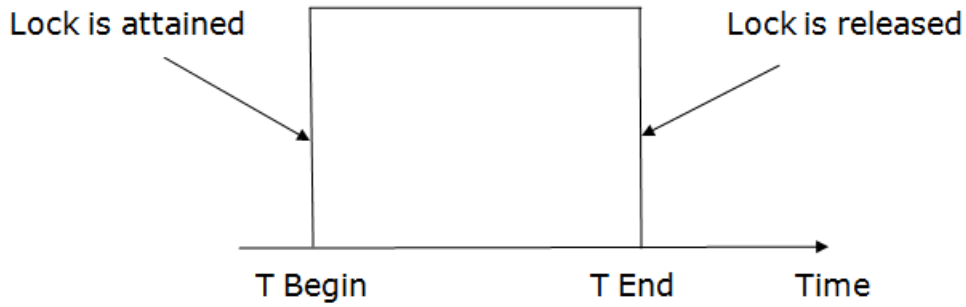
1. Simplistic lock protocol

It is the simplest way of locking the data while transaction. Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it. It will unlock the data item after completing the transaction.

2. Pre-claiming Lock Protocol

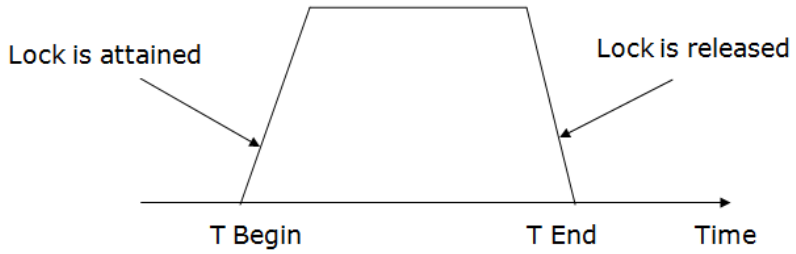
- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.

- If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.



3. Two-phase locking (2PL)

- The two-phase locking protocol divides the execution phase of the transaction into three parts.
- In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
- In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
- In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.



There are two phases of 2PL:

Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Example:

	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	——	——
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	——	——

The following way shows how unlocking and locking work with 2-PL.

Transaction T1:

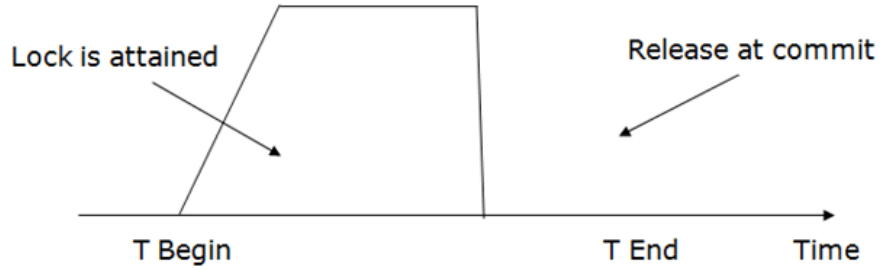
- **Growing phase:** from step 1-3
- **Shrinking phase:** from step 5-7
- **Lock point:** at 3

Transaction T2:

- **Growing phase:** from step 2-6
- **Shrinking phase:** from step 8-9
- **Lock point:** at 6

4. Strict Two-phase locking (Strict-2PL)

- The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
- The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
- Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock release.



It does not have cascading abort as 2PL does.

Timestamp Ordering Protocol

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
- Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has

entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.

- The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

Basic Timestamp ordering protocol works as follows:

1. Check the following condition whenever a transaction Ti issues a **Read (X)** operation:

- If W_TS(X) >TS(Ti) then the operation is rejected.
- If W_TS(X) <= TS(Ti) then the operation is executed.
- Timestamps of all the data items are updated.

2. Check the following condition whenever a transaction Ti issues a **Write(X)** operation:

- If TS(Ti) < R_TS(X) then the operation is rejected.
- If TS(Ti) < W_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.

Where,

TS(TI) denotes the timestamp of the transaction Ti.

R_TS(X) denotes the Read time-stamp of data-item X.

W_TS(X) denotes the Write time-stamp of data-item X.

Advantages and Disadvantages of TO protocol:

- TO protocol ensures serializability since the precedence graph is as follows:

Image: Precedence Graph for TS ordering

- TS protocol ensures freedom from deadlock that means no transaction ever waits.
- But the schedule may not be recoverable and may not even be cascade-free.

Validation Based Protocol

Validation phase is also known as optimistic concurrency control technique. In the validation based protocol, the transaction is executed in the following three phases:

1. **Read phase:** In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.
2. **Validation phase:** In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.
3. **Write phase:** If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

Here each phase has the following different timestamps:

Start(Ti): It contains the time when Ti started its execution.

Validation (T_i): It contains the time when Ti finishes its read phase and starts its validation phase.

Finish(Ti): It contains the time when Ti finishes its write phase.

- This protocol is used to determine the time stamp for the transaction for serialization using the time stamp of the validation phase, as it is the actual phase which determines if the transaction will commit or rollback.
- Hence TS(T) = validation(T).
- The serializability is determined during the validation process. It can't be decided in advance.

- While executing the transaction, it ensures a greater degree of concurrency and also less number of conflicts.
- Thus it contains transactions which have less number of rollbacks.

Multiple Granularity

Granularity: It is the size of data item allowed to lock.

Multiple Granularity:

- It can be defined as hierarchically breaking up the database into blocks which can be locked.
- The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
- It maintains the track of what to lock and how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.

For example: Consider a tree which has four levels of nodes.

- The first level or higher level shows the entire database.
- The second level represents a node of type area. The higher level database consists of exactly these areas.
- The area consists of children nodes which are known as files. No file can be present in more than one area.
- Finally, each file contains child nodes known as records. The file has exactly those records that are its child nodes. No records represent in more than one file.
- Hence, the levels of the tree starting from the top level are as follows:
    1. Database
    2. Area
    3. File
    4. Record

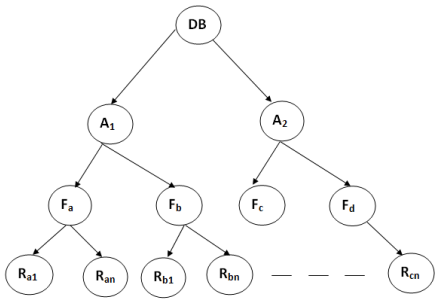


Figure: Multi Granularity tree Hierarchy

In this example, the highest level shows the entire database. The levels below are file, record, and fields.

There are three additional lock modes with multiple granularity:

Intention Mode Lock

Intention-shared (IS): It contains explicit locking at a lower level of the tree but only with shared locks.

Intention-Exclusive (IX): It contains explicit locking at a lower level with exclusive or shared locks.

Shared & Intention-Exclusive (SIX): In this lock, the node is locked in shared mode, and some node is locked in exclusive mode by the same transaction.

Compatibility Matrix with Intention Lock Modes: The below table describes the compatibility matrix for these lock modes:

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

It uses the intention lock modes to ensure serializability. It requires that if a transaction attempts to lock a node, then that node must follow these protocols:

- Transaction T1 should follow the lock-compatibility matrix.
- Transaction T1 firstly locks the root of the tree. It can lock it in any mode.
- If T1 currently has the parent of the node locked in either IX or IS mode, then the transaction T1 will lock a node in S or IS mode only.

- If T1 currently has the parent of the node locked in either IX or SIX modes, then the transaction T1 will lock a node in X, SIX, or IX mode only.
- If T1 has not previously unlocked any node only, then the Transaction T1 can lock a node.
- If T1 currently has none of the children of the node-locked only, then Transaction T1 will unlock a node.

Observe that in multiple-granularity, the locks are acquired in top-down order, and locks must be released in bottom-up order.

- If transaction T1 reads record R_{a9} in file F_a, then transaction T1 needs to lock the database, area A_1 and file F_a in IX mode. Finally, it needs to lock R_{a2} in S mode.
- If transaction T2 modifies record R_{a9} in file F_a, then it can do so after locking the database, area A_1 and file F_a in IX mode. Finally, it needs to lock the R_{a9} in X mode.
- If transaction T3 reads all the records in file F_a, then transaction T3 needs to lock the database, and area A in IS mode. At last, it needs to lock F_a in S mode.
- If transaction T4 reads the entire database, then T4 needs to lock the database in S mode.

Crash Recovery

DBMS is a highly complex system with hundreds of transactions being executed every second. The durability and robustness of a DBMS depends on its complex architecture and its underlying hardware and system software. If it fails or crashes amid transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data.

Failure Classification

To see where the problem has occurred, we generalize a failure into various categories, as follows −

Transaction failure

A transaction has to abort when it fails to execute or when it reaches a point from where it can't go any further. This is called transaction failure where only a few transactions or processes are hurt.

Reasons for a transaction failure could be −

- **Logical errors** − Where a transaction cannot complete because it has some code error or any internal error condition.
- **System errors** − Where the database system itself terminates an active transaction because the DBMS is not able to execute it, or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability, the system aborts an active transaction.

System Crash

There are problems − external to the system − that may cause the system to stop abruptly and cause the system to crash. For example, interruptions in power supply may cause the failure of underlying hardware or software failure.

Examples may include operating system errors.

Disk Failure

In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

Storage Structure

The storage structure can be divided into two categories −

- **Volatile storage** − As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto the chipset itself. For example, main memory and cache memory are examples of volatile storage. They are fast but can store only a small amount of information.
- **Non-volatile storage** − These memories are made to survive system crashes. They are huge in data storage capacity, but slower in accessibility. Examples may include hard-disks, magnetic tapes, flash memory, and non-volatile (battery backed up) RAM.

Recovery and Atomicity

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Log-based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe.

Log-based recovery works as follows −

- The log file is kept on a stable storage media.
- When a transaction enters the system and starts execution, it writes a log about it.

<T_n, Start>

- When the transaction modifies an item X, it write logs as follows −

<T_n, X, V_1, V_2>

It reads T_n has changed the value of X, from V_1 to V_2.

- When the transaction finishes, it logs −

<T_n, commit>

The database can be modified using two approaches −

- **Deferred database modification** − All logs are written on to the stable storage and the database is updated when a transaction commits.
- **Immediate database modification** − Each log follows an actual database modification. That is, the database is modified immediately after every operation.

Recovery with Concurrent Transactions

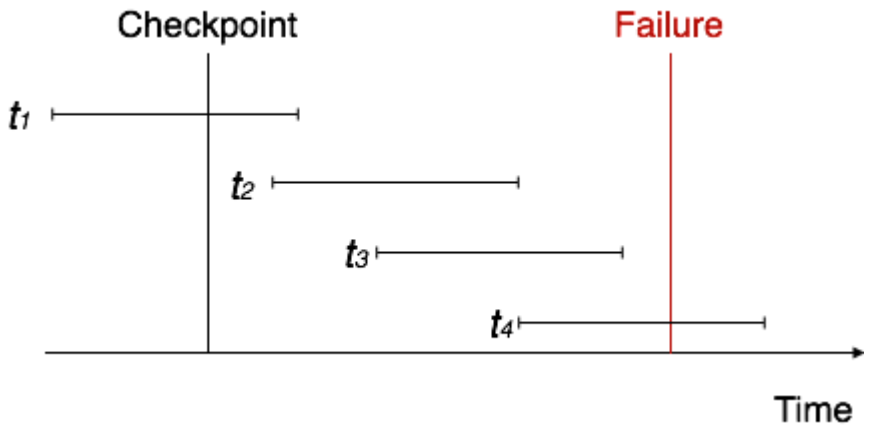
When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

Checkpoint

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −



- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

Database Backup – Physical and Logical

A database backup is a copy of storage that is stored on a server. Backup is used to prevent unexpected data loss. If original data gets lost, then with the help of a backup, it is easy to gain access to the data again.

There are two types of database backup.

- Physical backup
- Logical backup

Physical Backup:

Physical database backups are backups of physical files that are used to store and recover databases. These include different data files, control files, archived redo logs, and many more. Typically, physical backup data is kept in the cloud, offline storage, magnetic tape, or on a disc.

There are two methods to perform a physical backup :

1. Operating system utilities
2. Recovery manager

This type of backup is useful when the user needs to restore the complete database in a short period. It is beneficial to provide details of transactions and changes made in databases. It is considered the foundation of the recovery mechanism. This form of backup has the drawback of slowing down database operations.

Advantages:

- It is useful when the user needs to restore the complete database in a short period.
- They provide details of transactions and changes made in databases.

Disadvantage:

- This slows down database operations.

Logical Backup:

It contains logical data which is retrieved from the database. It contains a view, procedure, function, and table. This is useful When users want to restore or transfer a copy of the database to a different location. Logical backups are not as secure as physical backups in preventing data loss. It only provides structural details. Every week, complete logical backups should be performed. Logical backups are used as a supplement to a physical backup.

Advantages:

- This is useful when the user needs to restore the complete database to the last time.
- It was more complex and provides granular recovery capabilities.

Disadvantages:

- Critical for recovery of special components.
- less secure compared to physical backup.
- It only provides structural details.

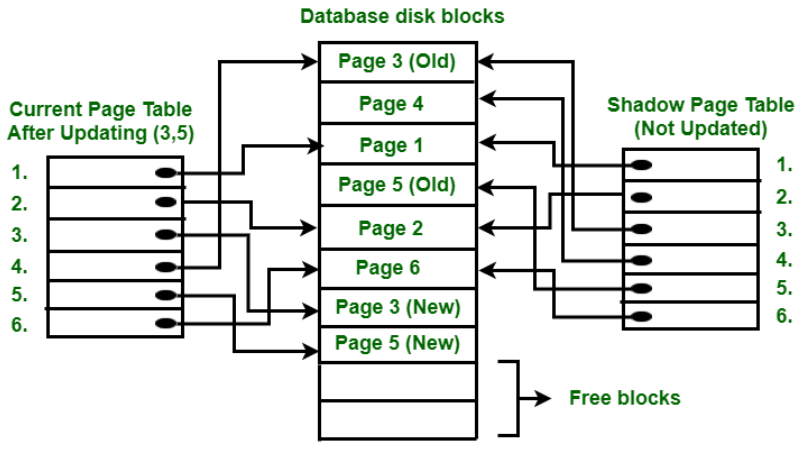
1. Deferred Update: It is a technique for the maintenance of the transaction log files of the DBMS. It is also called NO-UNDO/REDO technique. It is used for the recovery of transaction failures that occur due to power, memory, or OS failures. Whenever any transaction is executed, the updates are not made immediately to the database. They are first recorded on the log file and then those changes are applied once the commit is done. This is called the “Re-doing” process. Once the rollback is done none of the changes are applied to the database and the changes in the log file are also discarded. If the commit is done before crashing the system, then after restarting the system the changes that have been recorded in the log file are thus applied to the database.

2. Immediate Update: It is a technique for the maintenance of the transaction log files of the DBMS. It is also called UNDO/REDO technique. It is used for the recovery of transaction failures that occur due to power, memory, or OS

failures. Whenever any transaction is executed, the updates are made directly to the database and the log file is also maintained which contains both old and new values. Once the commit is done, all the changes get stored permanently in the database, and records in the log file are thus discarded. Once rollback is done the old values get restored in the database and all the changes made to the database are also discarded. This is called the "Un-doing" process. If the commit is done before crashing the system, then after restarting the system the changes are stored permanently in the database.

Shadow Paging

Shadow Paging is recovery technique that is used to recover database. In this recovery technique, database is considered as made up of fixed size of logical units of storage which are referred as **pages.** pages are mapped into physical blocks of storage, with help of the **page table** which allow one entry for each logical page of database. This method uses two page tables named **current page table** and **shadow page table**. The entries which are present in current page table are used to point to most recent database pages on disk. Another table i.e., Shadow page table is used when the transaction starts which is copying current page table. After this, shadow page table gets saved on disk and current page table is going to be used for transaction. Entries present in current page table may be changed during execution but in shadow page table it never get changed. After transaction, both tables become identical. This technique is also known as **Cut-of-Place updating.**



To understand concept, consider above figure. In this 2 write operations are performed on page 3 and 5. Before start of write operation on page 3, current page table points to old page 3. When write operation starts following steps are performed :

1. Firstly, search start for available free block in disk blocks.
2. After finding free block, it copies page 3 to free block which is represented by Page 3 (New).
3. Now current page table points to Page 3 (New) on disk but shadow page table points to old page 3 because it is not modified.
4. The changes are now propagated to Page 3 (New) which is pointed by current page table.

COMMIT Operation : To commit transaction following steps should be done :

1. All the modifications which are done by transaction which are present in buffers are transferred to physical database.
2. Output current page table to disk.
3. Disk address of current page table output to fixed location which is in stable storage containing address of shadow page table. This operation overwrites address of old shadow page table. With this current page table becomes same as shadow page table and transaction is committed.

Checkpoint

- The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
- The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.
- When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.
- The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

Recovery using Checkpoint

In the following manner, a recovery system recovers the database from this failure: