Module 6

Transaction Processing Concepts: overview of concurrency control and recovery acid properties, serial and concurrent schedules, conflict serializability. Two-phase locking, failure classification, storage structure, stable storage, log based recovery, deferred database modification, check-pointing, Recent topics (preliminary ideas only): Semantic Web and RDF, GIS, biological databases, Big Data

6.1 Transaction Processing Concepts

A transaction can be defined as an action or series of actions that is carried out by a single user or application program to perform operations for accessing the contents of the database. The operations can include retrieval, (Read), insertion (Write), deletion and modification. A transaction must be either completed or aborted. Each transaction should access shared data without interfering with the other transactions and whenever a transaction successfully completes its execution; its effect should be permanent.

Let's take an example of a simple transaction. Suppose a bank employee transfers Rs 500 from A's account to B's account

A's Account

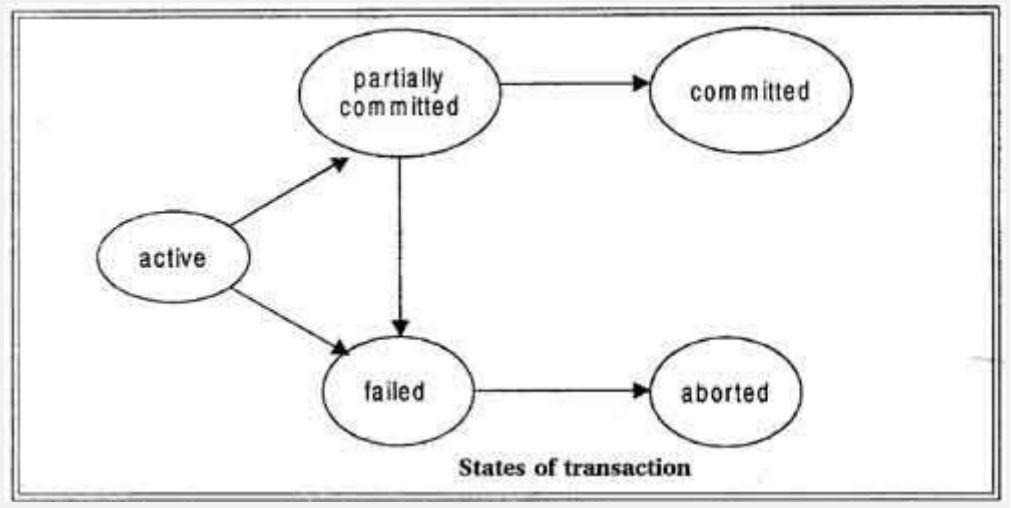
```
Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)
```

B's Account

```
Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)
```

States of Transaction

In a database, the transaction can be in one of the following states -



States of 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

Transaction Properties :

A transaction must have the following four properties, called ACID properties (also called ACIDITY of a transaction), to ensure that a database remains stable state after the transaction is executed:

1. **A**tomicity.
2. **C**onsistency.
3. **I**solation.
4. **D**urability.

Atomicity − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed.

Consistency is the property that every transaction sees a consistent database instance.In other words, execution of a transaction must leave a database in either its prior stable state or a new stable state that reflects the new modifications (updates) made by the transaction.If the transaction fails, the database must be returned to the state it was in prior to the execution of the failed transaction.If the transaction commits, the database must reflect the new changes.Thus, all resources are always in a consistent state

Isolation − 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: The effect of completed or committed transactions should persist even after a crash. It means once a transaction commits, the system must guarantee that the result of its operations will never be lost

6.2 Transaction Schedule

A chronological execution sequence of a transaction is called a schedule. A schedule can have many transactions in it, each comprising of a number of instructions/tasks.

Two type,

- Serial
- Non-serial/Interleaved/concurrent

Serial Schedule − It is a schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

Non-serial / Interleaved- A non-serial schedule is a schedule where the operations of a group of concurrent transactions are interleaved. Transactions are performed in non-serial order, but result should be same as serial

Non-serial schedule

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

Schedule S1

Serial Schedule

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

Schedule S2

Serializable: A transaction is said to be Serializable if a non serial schedule is equivalent to a serial schedule. Depending on the type of schedules, we have two types of serializability:

- Conflict serializability
- View serializability.

Conflict serializability

A schedule is called conflict serializable if we can convert a non serial schedule into a serial schedule after swapping its non-conflicting operations.

Conflicting operations

Two operations are said to be in conflict, if they satisfy all the following three conditions:

1. Both the operations should belong to different transactions.
2. Both the operations are working on same data item.
3. At least one of the operation is a write operation.

Lets see some examples to understand this:

Example 1: Operation W(X) of transaction T1 and operation R(X) of transaction T2 are conflicting operations, because they satisfy all the three conditions mentioned above. They belong to different transactions, they are working on same data item X, one of the operation in write operation.

Example 2: Similarly Operations W(X) of T1 and W(X) of T2 are conflicting operations.

Example 3: Operations W(X) of T1 and W(Y) of T2 are non-conflicting operations because both the write operations are not working on same data item so these operations don't satisfy the second condition.

Example 4: Similarly R(X) of T1 and R(X) of T2 are non-conflicting operations because none of them is write operation.

Example 5: Similarly W(X) of T1 and R(X) of T1 are non-conflicting operations because both the operations belong to same transaction T1.

Conflict Equivalent Schedules

Two schedules are said to be conflict Equivalent if one schedule can be converted into other schedule after swapping non-conflicting operations.

Consider the following schedule:

Q1. S1: $R_1(A)$, $W_1(A)$, $R_2(A)$, $W_2(A)$, $R_1(B)$, $W_1(B)$, $R_2(B)$, $W_2(B)$

we can get two transactions of schedule S1 as:

T1: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$

T2: $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$

Possible Serial Schedules are: T1->T2 or T2->T1

-> Swapping non-conflicting operations $R_2(A)$ and $R_1(B)$ in S1, the schedule becomes,

S11: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_2(A)$, $R_2(A)$, $W_1(B)$, $R_2(B)$, $W_2(B)$

-> Similarly, swapping non-conflicting operations $W_2(A)$ and $W_1(B)$ in S11, the schedule becomes,

S12: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$, $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$

S12 is a serial schedule in which all operations of T1 are performed before starting any operation of T2. Since S has been transformed into a serial schedule S12 by swapping non-conflicting operations of S1, S1 is conflict serializable.

Q2. Let us take another Schedule:

S2: $R_2(A)$, $W_2(A)$, $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$, $R_2(B)$, $W_2(B)$

Two transactions will be:

T1: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$

T2: $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$

Possible Serial Schedules are: T1->T2 or T2->T1

Original Schedule is:

S2: $R_2(A)$, $W_2(A)$, **$R_1(A)$,** $W_1(A)$, $R_1(B)$, $W_1(B)$, **$R_2(B)$,** $W_2(B)$

T1:$R_1(A)$,$W_1(A)$, $R_1(B)$, $W_1(B)$

T2:$R_2(A)$, $W_2(A)$,$R_2(B)$**,** $W_2(B)$

Possible Serial Schedules are: T1->T2 or T2->T1

Swapping non-conflicting operations $R_1(A)$ and $R_2(B)$ in S2, the schedule becomes,

S21: $R_2(A)$, $W_2(A)$, $R_2(B)$, **$W_1(A)$,** $R_1(B)$, $W_1(B)$, $R_1(A)$, **$W_2(B)$**

Similarly, swapping non-conflicting operations $W_1(A)$ and $W_2(B)$ in S21, the schedule becomes,

S22: $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$, $R_1(B)$, $W_1(B)$, $R_1(A)$, $W_1(A)$

In schedule S22, all operations of T2 are performed first, but operations of T1 are not in order (order should be $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$). So S2 is not conflict serializable.

Precedence Graph For Testing Conflict Serializability

It is a directed Graph (V, E) consisting of a set of nodes V = {T_1, T_2, T_3…….T_n} and a set of directed edges E = {e_1, e_2, e_3……e_m}.

The Algorithm can be written as:

1. Create a node T in the graph for each participating transaction in the schedule.
2. For the conflicting operation – If a Transaction T_j executes a read_item (X) after T_i executes a write_item (X), draw an edge from T_i to T_j in the graph.
3. For the conflicting operation – If a Transaction T_j executes a write_item (X) after T_i executes a read_item (X), draw an edge from T_i to T_j in the graph.
4. For the conflicting operation – If a Transaction T_j executes a write_item (X) after T_i executes a write_item (X), draw an edge from T_i to T_j in the graph.
5. **The Schedule S is serializable if there is no cycle in the precedence graph**.

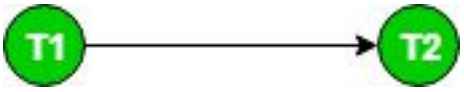
Consider the schedule S :

S : r1(x) r1(y) w2(x) w1(x) r2(y)

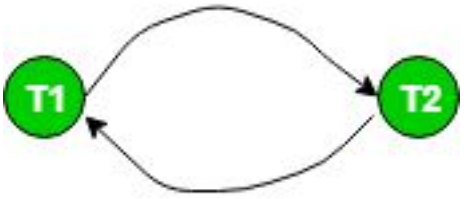
Creating Precedence graph:

1. Make two nodes corresponding to Transaction T_1 and T_2.

2. For the conflicting pair r1(x) w2(x), where r1(x) happens before w2(x), draw an edge from T_1 to T_2.



3. For the conflicting pair w2(x) w1(x), where w2(x) happens before w1(x), draw an edge from T_2 to T_1.



Since the graph is cyclic, we can conclude that it is **not conflict serializable** to any schedule serial schedule.

Let us try to infer a serial schedule from this graph using topological ordering.

The edge T_1–>T_2 tells that T_1 should come before T_2 in the linear ordering.

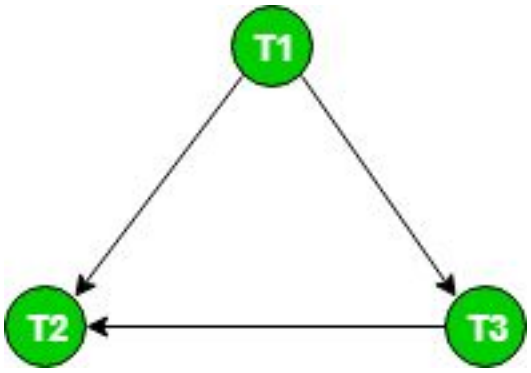
The edge T_2 –> T_1 tells that T_2 should come before T_1 in the linear ordering.

So, we can not predict any particular order (when the graph is cyclic). Therefore, no serial schedule can be obtained from this graph.

Q2.Consider the another schedule S1 :

S1: r1(x) r3(y) w1(x) w2(y) r3(x) w2(x)

The graph for this schedule is :



Since the graph is acyclic, the schedule is conflict serializable. Performing Topological Sort on this graph would give us a possible serial schedule which is conflict equivalent to schedule S1.

In Topological Sort, we first select the node with indegree 0, which is T1. This would be followed by T3 and T2.

6.3 overview of concurrency control

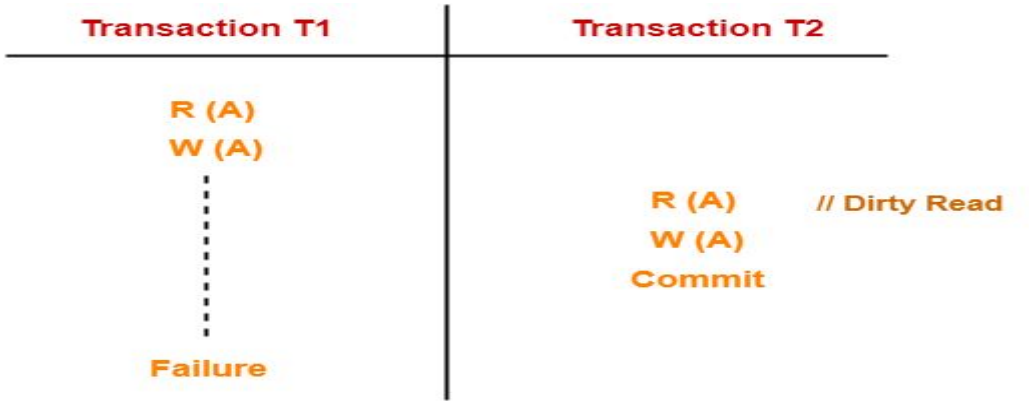
Concurrent execution of database transactions in a multi-user system means that any number of users can use the same database at the same time. Concurrency control is needed in order to avoid inconsistencies in the database. It improve the cpu utilization. Concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions. Several problems can occur when concurrent transactions are executed in an uncontrolled manner. Following are the three problems in concurrency control.

- Lost updates
- Dirty read
- Unrepeatable read
- Incorrect summary problem

Dirty Read Problem- Reading the data written by an uncommitted transaction is called as dirty read. There is always a chance that the uncommitted transaction might roll back later.

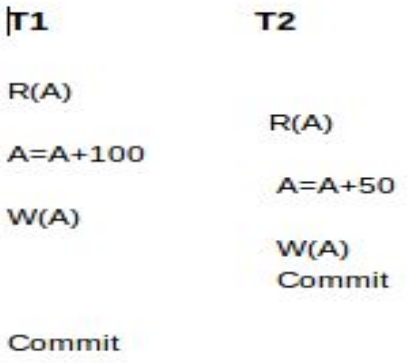
Thus, uncommitted transaction might make other transactions read a value that does not even exist. This leads to inconsistency of the database. Dirty read does not lead to

inconsistency always. It becomes problematic only when the uncommitted transaction fails and roll backs later due to some reason



1. T1 reads the value of A (= 10).
2. T1 write the value of A (= 15).
3. T2 read the value of A (15)
4. T2 write the value of A (20) and commit
5. T1 fails, so T2 become dirty read

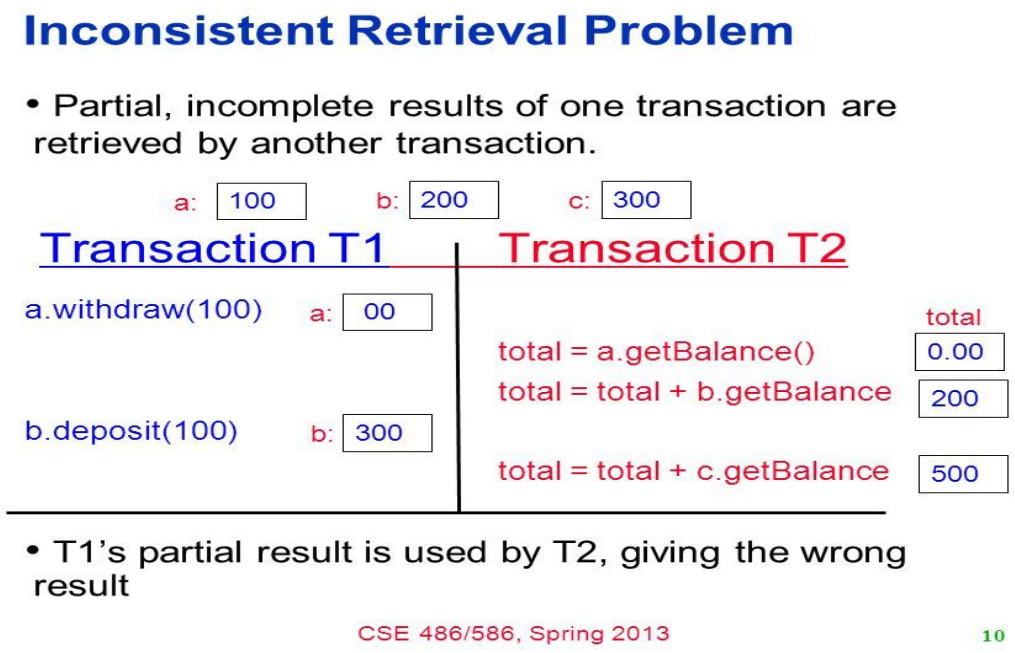
Lost Update Problem: Update operation of one user is overwritten by other user. The lost update problem occurs when 2 concurrent transactions try to read and update the same data.



1. T1 reads the value of A (= 100).
2. T2 read the value of A (= 100).
3. T1 change the value of A (=200)
4. T2 change the value of A (=150)
5. T1 write the value of A (=200)

6. T2 write the value of A (=150) and commit
7. T1 commit
8. T2 lost its update, T1 overwrite it.

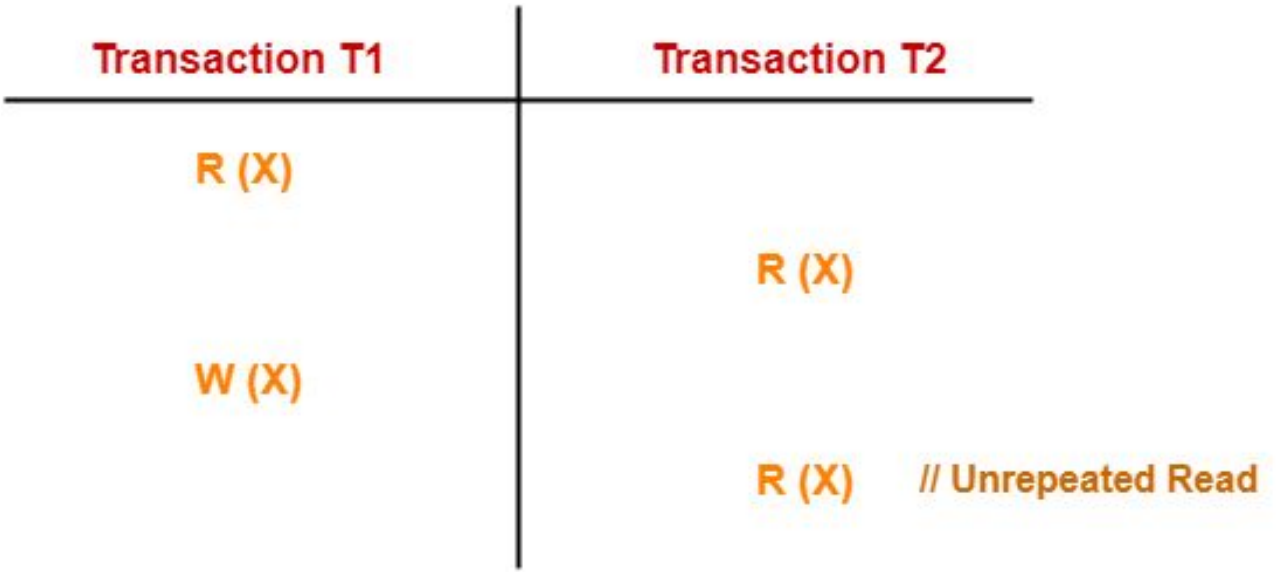
Inconsistent Retrievals Problem/ Incorrect summary : When a transaction calculates some summary function over a set of data while the other transactions are updating the data, then the Inconsistent Retrievals Problem occurs.



Unrepeatable read

This problem occurs when a transaction gets to read unrepeated i.e. different values of the same variable in its different read operations even when it has not updated its value.

Example-



T1 reads the value of A (= 100).

T2 read the value of A (= 100).

T1 change the value of A (=200)

T2 read the value of A (=200)

Concurrency control protocols can be broadly divided into two categories −

- Lock based protocols
- Time stamp based protocols
- Optimistic concurrency technique

6.3.1 Lock-based Protocols

Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds −

- **Binary Locks** − A lock on a data item can be in two states; it is either locked or unlocked.
- **Shared or exclusive** − This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation,

it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions.

Two-Phase Locking 2PL: Two-phase locking has two phases, one is **growing**, where all the locks are being acquired by the transaction; and the second phase is **shrinking**, where the locks held by the transaction are being released.

Conservative Two-Phase Locking Protocol

Conservative Two – Phase Locking Protocol is also called as Static Two – Phase Locking Protocol. This protocol is almost free from deadlocks as all required items are listed in advanced. The difference between 2PL and C2PL is that C2PL's transactions obtain all the locks they need before the transactions begin and release each lock after use.

Strict Two-Phase Locking: The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a exclusive lock after using it. Strict-2PL holds all the exclusive locks until the commit point and releases all the locks at a time. Strict Two-Phase Locking Protocol avoids cascaded rollbacks. It is not deadlock free.

Rigorous 2PL: In Rigorous 2PL all locks are released only after commit(both shared and exclusive)

6.3.2 Timestamp-based Protocols

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.

In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last ‘read and write’ operation was performed on the data item.

Every transaction is issued a timestamp based on when it enters the system. Suppose, if an old transaction T_i has timestamp $TS(T_i)$, a new transaction T_j is assigned timestamp $TS(T_j)$ such that **$TS(T_i) < TS(T_j)$**.The protocol manages concurrent execution such that the timestamps determine the serializability order. The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. Whenever some Transaction *T* tries to issue a R_item(X) or a W_item(X), the Basic TO algorithm compares the timestamp of *T* with **R_TS(X) & W_TS(X)** to ensure that the Timestamp order is not violated. This describe the Basic TO protocol in following two cases.

1. Whenever a Transaction *T* issues a **W_item(X)** operation, check the following conditions:

    - If ***R_TS(X) > TS(T)*** or if ***W_TS(X) > TS(T)***, then abort and rollback T and reject the operation.

      Else write

2. Whenever a Transaction *T* issues a **R_item(X)** operation, check the following conditions:

    If ***W_TS(X) > TS(T)***, then abort and reject T and reject the operation, else

    - If W_TS(X) <= TS(T), then execute the R_item(X) operation of T and

6.3.3 Optimistic Concurrency Control Algorithm

In this approach, a transaction's life cycle is divided into the following three phases −

- **Execution Phase** − A transaction fetches data items to memory and performs operations upon them.
- **Validation Phase** − A transaction performs checks to ensure that committing its changes to the database passes serializability test.

- **Commit Phase** − A transaction writes back modified data item in memory to the disk.

This algorithm uses three rules to enforce serializability in validation phase −
Rule 1 − Given two transactions T_i and T_j, if T_i is reading the data item which T_j is writing. T_j can commit only after T_i has finished execution.
Rule 2 − Given two transactions T_i and T_j, if T_i is writing the data item that T_j is reading, T_j can start executing only after T_i has already committed.
Rule 3 − Given two transactions T_i and T_j, if T_i is writing the data item which T_j is also writing. T_j can start to commit only after T_i has already committed.

6.4 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

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.

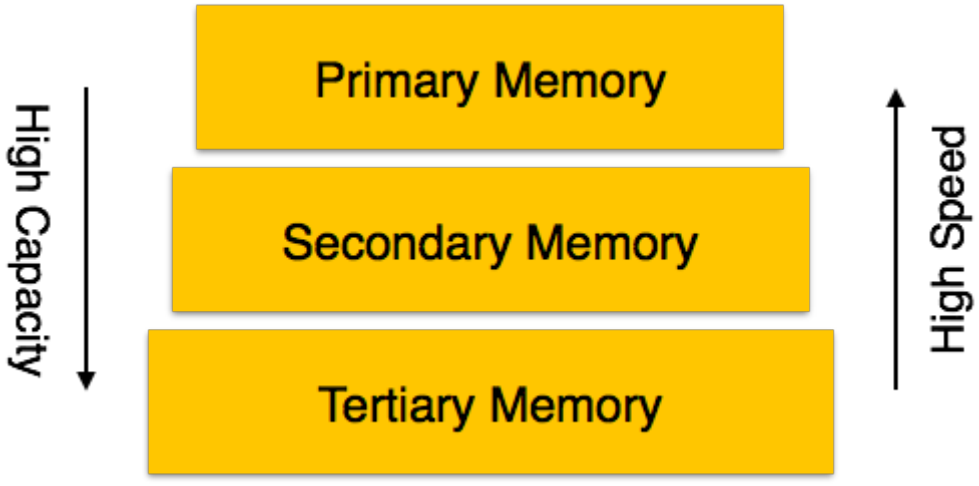
Disk Failure

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.

6.5 Storage Structure

Databases are stored in file formats, which contain records. These storage devices can be broadly categorized into three types −



- **Primary Storage/Volatile storage** − The memory storage that is directly accessible to the CPU comes under this category. CPU's internal memory (registers), fast memory (cache), and main memory (RAM) are directly accessible to the CPU, as they are all placed on the motherboard or CPU chipset. This storage is typically very small, ultra-fast, and volatile. Primary storage requires continuous power supply in order to maintain its state. In case of a power failure, all its data is lost.
- **Secondary Storage/NonVolatile storage** − Secondary storage devices are used to store data for future use or as backup. Secondary storage includes memory devices that are not a part of the CPU chipset or motherboard, for example, magnetic disks, optical disks (DVD, CD, etc.), hard disks, flash drives, and magnetic tapes.
- **Tertiary Storage** − Tertiary storage is used to store huge volumes of data. Since such storage devices are external to the computer system, they are the slowest in speed. These storage devices are mostly used to take the back up of an entire system. Optical disks and magnetic tapes are widely used as tertiary storage.

6.6 Recovery and Atomicity:

When a system crashes, it should have many transactions being executed and numerous files opened for them to switch the information items. Transactions are a product of numerous operations that are atomic in nature. However consistent with ACID properties of a database, atomicity of transactions as an entire should be maintained, that is, either all the operations are executed or none.

6.6.1 Log-based recovery Or Manual Recovery):

Log could be a sequence of records, which maintains the records of actions performed by dealing. It’s necessary that the logs area unit written before the particular modification and hold on a stable storage media, that is failsafe. Log-based recovery works as follows:

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

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.

$$<T_n, \text{Start}>$$

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

$$<W, T_n, X, OV_1, NV_2>$$

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

$$<R, T_n, X>$$

- When the transaction finishes, it logs −

$$<T_n, \text{commit}>$$

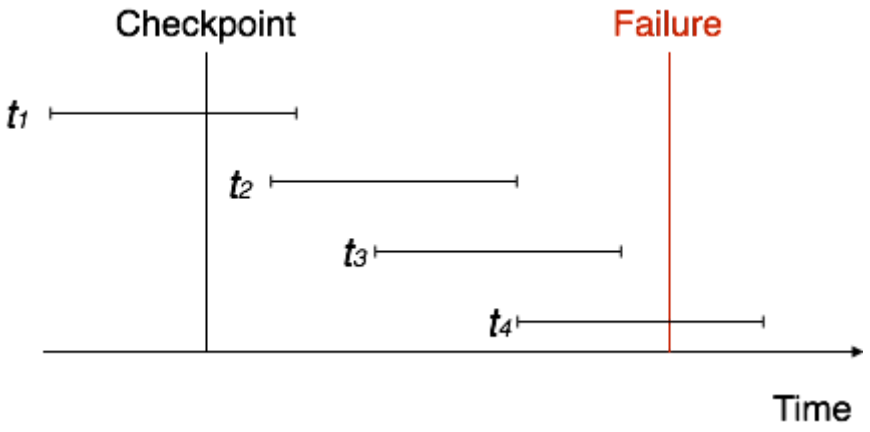
6.6.2 Recovery with Concurrent Transactions

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.



- 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 before <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.

- **Deferred update –** This technique does not physically update the database on disk until a transaction has reached its commit point. Before reaching commit, all transaction updates are recorded in the local transaction workspace. If a transaction fails before reaching its commit point, it will not have changed the database in any way so UNDO is not needed. It may be necessary to REDO the effect of the operations that are recorded in the local transaction workspace, because their effect may not yet have been written in the database. Hence, a deferred update is also known as the **No-undo/redo algorithm**

- **Immediate update –** In the immediate update, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are recorded in a log on disk before they are applied to the database, making recovery still possible. If a transaction fails to reach its commit point, the effect of its operation must be undone i.e. the transaction must be rolled back hence we require both undo and redo. This technique is known as **undo/redo algorithm.**

6.7 Semantic Web and RDF

The Semantic Web is a vision about an extension of the existing World Wide Web, which provides software programs with machine-interpretable metadata of the published information and data. RDF is used to develop semantic web.

- RDF stands for Resource Description Framework
- RDF is a framework for describing resources on the web
- RDF is designed to be read and understood by computers
- RDF is not designed for being displayed to people
- RDF is written in XML
- RDF is a part of the W3C's Semantic Web Activity

RDF - Examples of Use

- Describing properties for shopping items, such as price and availability
- Describing time schedules for web events
- Describing information about web pages (content, author, created and modified date)
- Describing content and rating for web pictures
- Describing content for search engines

- Describing electronic libraries

6.8 GIS

Geographic Information Systems (GIS) Connects Geography with Data Every day, millions of decisions are being powered by Geographic Information Systems (GIS). Geographic Information Systems is a computer-based tool that analyzes, stores, manipulates and visualizes geographic information, usually in a map. GIS data can be separated into two categories:data which is represented by vector and raster forms

vector data model: [data models] A representation of the world using points, lines, and polygons. Vector models are useful for storing data that has discrete boundaries, such as country borders, land parcels, and streets.
raster data model: [data models] A representation of the world as a surface divided into a regular grid of cells. Raster models are useful for storing data that varies continuously, as in an aerial photograph, a satellite image, a surface of chemical concentrations, or an **e**levation surface.

6.9 Biological database

Biological databases are libraries of life sciences information, collected from scientific experiments, published literature, high-throughput experiment technology, and computational analysis. One of the first databases to emerge was GenBank, which is a collection of all available protein and DNA sequences. It is maintained by the National Institutes of Health (NIH) and the National Center for Biotechnology Information (NCBI). GenBank paved the way for the Human Genome Project (HGP). The HGP allowed complete sequencing and reading of the genetic blueprint. The data stored in biological databases is organized for optimal analysis and consists of two types: raw and curated (or annotated). Biological databases are complex, heterogeneous, dynamic, and yet inconsistent. The inconsistency is due to the lack of standards at the ontological level. Data is submitted directly to biological databases for indexing, organization, and data

optimization. Biological databases can be further classified as primary, secondary, and composite databases.

Primary databases contain information for sequence or structure only. Examples of primary biological databases include:

- Swiss-Prot and PIR for protein sequences
- GenBank and DDBJ for genome sequences
- Protein Databank for protein structures

Secondary databases contain information derived from primary databases. Secondary databases store information such as conserved sequences, active site residues, and signature sequences. Protein Databank data is stored in secondary databases. Composite databases contain a variety of primary databases, which eliminates the need to search each one separately. Each composite database has different search algorithms and data structures.

6.10 Big Data

Big Data is also data but with a huge size. Big Data is a term used to describe a collection of data that is huge in size and yet growing exponentially with time. In short such data is so large and complex that none of the traditional data management tools are able to store it or process it efficiently.

Examples Of Big Data: Social Media

The statistic shows that *500+terabytes* of new data get ingested into the databases of social media site Facebook, every day. This data is mainly generated in terms of photo and video uploads, message exchanges, putting comments etc.

Types Of Big Data

BigData' could be found in three forms:

1. Structured
2. Unstructured
3. Semi-structured

Structured

Any data that can be stored, accessed and processed in the form of fixed format is termed as a 'structured' data.

Unstructured

Any data with unknown form or the structure is classified as unstructured data. In addition to the size being huge, un-structured data poses multiple challenges in terms of its processing for deriving value out of it.

Semi-structured

Semi-structured data can contain both the forms of data. We can see semi-structured data as a structured in form but it is actually not defined with e.g. a table definition in relational DBMS