# Concurrency Control in DBMS

The fundamental goal of database concurrency control is to ensure that concurrent execution of transactions does not result in a loss of database consistency. The concept of serializability can be used to achieve this goal, since all serializable schedules preserve consistency of the database. However, not all schedules that preserve consistency of the database are serializable.

In general it is not possible to perform an automatic analysis of low-level operations by transactions and check their effect on database consistency constraints. However, there are simpler techniques. One is to use the database consistency constraints as the basis for a split of the database into subdatabases on which concurrency can be managed separately.

Another is to treat some operations besides read and write as fundamental low-level operations and to extend concurrency control to deal with them.

**Concurrency Control Problems**

There are several problems that arise when numerous transactions are executed simultaneously in a random manner. The database transaction consist of two major operations "Read" and "Write". It is very important to manage these operations in the concurrent execution of the transactions in order to maintain the consistency of the data.

**Dirty Read Problem(Write-Read conflict)**

Dirty read problem occurs when one transaction updates an item but due to some unconditional events that transaction fails but before the transaction performs rollback, some other transaction reads the updated value. Thus creates an inconsistency in the database. Dirty read problem comes under the scenario of Write-Read conflict between the transactions in the database

1. The lost update problem can be illustrated with the below scenario between two transactions T1 and T2.
2. Transaction T1 modifies a database record without committing the changes.
3. T2 reads the uncommitted data changed by T1
4. T1 performs rollback
5. T2 has already read the uncommitted data of T1 which is no longer valid, thus creating inconsistency in the database.

**Lost Update Problem**

Lost update problem occurs when two or more transactions modify the same data, resulting in the update being overwritten or lost by another transaction. The lost update problem can be illustrated with the below scenario between two transactions T1 and T2.

1. T1 reads the value of an item from the database.
2. T2 starts and reads the same database item.
3. T1 updates the value of that data and performs a commit.
4. T2 updates the same data item based on its initial read and performs commit.
5. This results in the modification of T1 gets lost by the T2's write which causes a lost update problem in the database.

**Concurrency Control Techniques**

Concurrency control is provided in a database to:

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

Various concurrency control techniques are:

**1.** locking Protocol
**2.** Time stamp ordering Protocol
**3.** Multi version concurrency control
**4.** Validation concurrency control
5. Multiple Granularity

**Lock Based Concurrency Control Protocol**

In a database management system (DBMS), lock-based concurrency control (BCC) is used to control the access of multiple transactions to the same data item. This protocol helps to maintain data consistency and integrity across multiple users.

In the protocol, transactions gain locks on data items to control their access and prevent conflicts between concurrent transactions. This article will look deep into the Lock Based Protocol in detail.

**What is a Lock?**

A Lock is a variable assigned to any data item to keep track of the status of that data item so that isolation and non-interference are ensured during concurrent transactions.

Lock-based concurrency control ensures that transactions in a database can proceed safely without causing conflicts. Mastering this topic is key for anyone studying DBMS.

**Lock Based Protocols**

A lock is a variable associated with a data item that describes the status of the data item to possible operations that can be applied to it. They synchronize the access by concurrent transactions to the database items. It is required in this protocol that all the data items must be accessed in a mutually exclusive manner. Let me introduce you to two common locks that are used and some terminology followed in this protocol.

1. **Shared Lock (S):** Shared Lock is also known as Read-only lock. As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.
2. **Exclusive Lock (X):** Data item can be both read as well as written.This is Exclusive and cannot be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

**Lock Compatibility Matrix**

A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions. Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive(X) on the item no other transaction may hold any lock on the item. If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. Then the lock is granted.

| | S | X |
|---|---|---|
| S | ✔ | ✗ |
| X | ✗ | ✗ |

Lock Compatibility Matrix

**Types of Lock-Based Protocols**

**1. Simplistic Lock Protocol**

It is the simplest method for locking data during a transaction. Simple lock-based protocols enable all transactions to obtain a lock on the data before inserting, deleting, or updating it. It will unlock the data item once the transaction is completed.

**2. Pre-Claiming Lock Protocol**

Pre-claiming Lock Protocols assess transactions to determine which data elements require locks. Before executing the transaction, it asks the DBMS for a lock on all of the data elements. If all locks are given, this protocol will allow the transaction to start. When the transaction is finished, it releases all locks. If all of the locks are not provided, this protocol allows the transaction to be reversed and waits until all of the locks are granted.

**3. Two-phase locking (2PL)**

A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases

- **Growing Phase:** New locks on data items may be acquired but none can be released.
- **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.

For more detail refer the published article Two-phase locking (2PL).

**4. Strict Two-Phase Locking Protocol**

Strict Two-Phase Locking requires that in addition to the 2-PL **all Exclusive(X) locks** held by the transaction be released until after the Transaction Commits. For more details refer the published article Strict Two-Phase Locking Protocol.

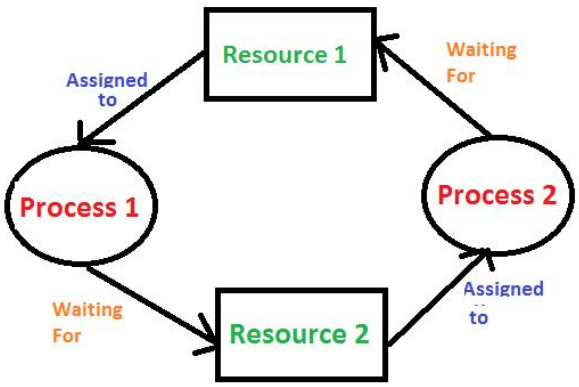
**Problem With Simple Locking**

Consider the Partial Schedule:

| S.No | $T_1$ | $T_2$ |
|---|---|---|
| 1 | lock-X(B) | |
| 2 | read(B) | |
| 3 | B:=B-50 | |
| 4 | write(B) | |
| 5 | | lock-S(A) |
| 6 | | read(A) |
| 7 | | lock-S(B) |
| 8 | lock-X(A) | |
| 9 | …… | …… |

**1. Deadlock**

In deadlock consider the above execution phase. Now, $T_1$ holds an Exclusive lock over B, and $T_2$ holds a Shared lock over A. Consider Statement 7, $T_2$ requests for lock on B, while in Statement 8 $T_1$ requests lock on A. This as you may notice imposes a deadlock as none can proceed with their execution.




Deadlock

**2. Starvation**

Starvation is also possible if concurrency control manager is badly designed. For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. This may be avoided if the concurrency control manager is properly designed.

**Two-Phase Locking Protocol:**
Locking is an operation which secures: permission to read, OR permission to write a data item. Two phase locking is a process used to gain ownership of shared resources without creating the possibility of deadlock. The 3 activities taking place in the two phase update algorithm are:
**(i).** Lock Acquisition
**(ii).** Modification of Data
**(iii).** Release Lock
Two phase locking prevents deadlock from occurring in distributed systems by releasing all the resources it has acquired, if it is not possible to acquire all the resources required without waiting for another process to finish using a lock. This means that no process is ever in a state where it is holding some shared resources, and waiting for another process to release a shared resource which it requires. This means that deadlock cannot occur due to resource contention. A transaction in the Two Phase Locking Protocol can assume one of the 2 phases:

**(i) Growing Phase:** In this phase a transaction can only acquire locks but cannot release any lock. The point when a transaction acquires all the locks it needs is called the Lock Point.

**(ii) Shrinking Phase:** In this phase a transaction can only release locks but cannot acquire any.

Now, recalling where we last left off, there are two types of Locks available Shared S(a) and Exclusive X(a). Implementing this lock system without any restrictions gives us the Simple Lock-based protocol (or Binary Locking), but it has its own disadvantages, they do not guarantee Serializability. Schedules may follow the preceding rules but a non-serializable schedule may result.
To guarantee serializability, we must follow some additional protocols concerning the positioning of locking and unlocking operations in every transaction. This is where the concept of Two-Phase Locking(2-PL) comes into the picture, 2-PL ensures serializability. Now, let's dig deep!

**Two Phase Locking**
A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases.

- **Growing Phase:** New locks on data items may be acquired but none can be released.
- **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.

**Note:** If lock conversion is allowed, then upgrading of lock( from S(a) to X(a) ) is allowed in the Growing Phase, and downgrading of lock (from X(a) to S(a)) must be done in the shrinking phase.
Let's see a transaction implementing 2-PL.

| | **T1** | **T2** |
|---|---|---|
| 1 | lock-S(A) | |
| 2 | | lock-S(A) |
| 3 | lock-X(B) | |
| 4 | ……. | ……. |
| 5 | Unlock(A) | |
| 6 | | Lock-X(C) |
| 7 | Unlock(B) | |
| 8 | | Unlock(A) |
| 9 | | Unlock(C) |
| 10 | ……. | ……. |

This is just a skeleton transaction that shows how unlocking and locking work with 2-PL. Note for:
**Transaction $T_1$**

- The growing Phase is from steps 1-3
- The shrinking Phase is from steps 5-7
- Lock Point at 3

**Transaction $T_2$**

- The growing Phase is from steps 2-6
- The shrinking Phase is from steps 8-9
- Lock Point at 6

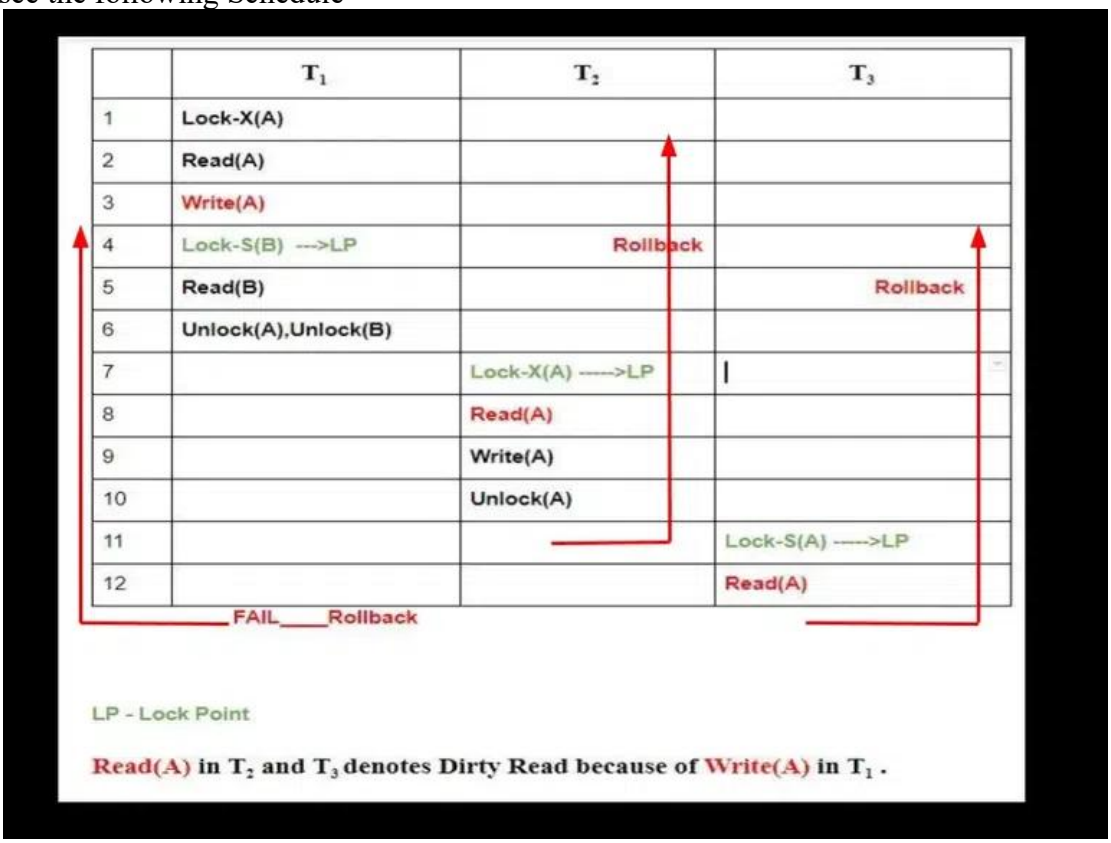
**Lock Point**

The Point at which the growing phase ends, i.e., when a transaction takes the final lock it needs to carry on its work. Now look at the schedule, you'll surely understand. I have said that 2-PL ensures serializability, but there are still some drawbacks of 2-PL. Let's glance at the drawbacks.

- Cascading Rollback is possible under 2-PL.
- Deadlocks and Starvation are possible.

**Cascading Rollbacks in 2-PL**

Let's see the following Schedule




Cascading Roll-Back

Take a moment to analyze the schedule. Yes, you're correct, because of Dirty Read in $T_2$ and $T_3$ in lines 8 and 12 respectively, when $T_1$ failed we have to roll back others also. Hence, Cascading Rollbacks are possible in 2-PL. I have taken skeleton schedules as examples because it's easy to understand when it's kept simple. When explained with real-time transaction problems with many variables, it becomes very complex.

**Deadlock in 2-PL**

Consider this simple example, it will be easy to understand. Say we have two transactions $T_1$ and $T_2$.

**Schedule:** Lock-$X_1$(A) Lock-$X_2$(B) Lock-$X_1$(B) Lock-$X_2$(A)

Drawing the precedence graph, you may detect the loop. So Deadlock is also possible in 2-PL.

Two-phase locking may also limit the amount of concurrency that occurs in a schedule because a Transaction may not be able to release an item after it has used it. This may be because of the protocols and other restrictions we may put on the schedule to ensure serializability, deadlock freedom, and other factors. This is the price we have to pay to ensure serializability and other

factors, hence it can be considered as a bargain between concurrency and maintaining the ACID properties.

The above-mentioned type of 2-PL is called **Basic 2PL**. To sum it up it ensures Conflict Serializability but **does not** prevent Cascading Rollback and Deadlock. Further, we will study three other types of 2PL, **Strict 2PL**, **Conservative 2PL**, and **Rigorous 2PL**.

**Problem with Two-Phase Locking**

- It does not insure recoverability which can be solved by strict two-phase locking and rigorous two-phase locking.
- It does not ensure a cascade-less schedule which can be solved by strict two-phase locking and rigorous two-phase locking.
- It may suffer from deadlock which can be solved by conservative two-phase locking.

**Timestamp based Concurrency Control**

Timestamp-based concurrency control is a method used in database systems to ensure that transactions are executed safely and consistently without conflicts, even when multiple transactions are being processed simultaneously. This approach relies on timestamps to manage and coordinate the execution order of transactions. Refer to the timestamp of a transaction *T* as ***TS(T)***.

The main idea for this protocol is to order the transactions based on their Timestamps. A schedule in which the transactions participate is then serializable and the only *equivalent serial schedule permitted* has the transactions in the order of their Timestamp Values. Stating simply, the schedule is equivalent to the particular *Serial Order* corresponding to the *order of the Transaction timestamps*. An algorithm must ensure that, for each item accessed by *Conflicting Operations* in the schedule, the order in which the item is accessed does not violate the ordering. To ensure this, use two Timestamp Values relating to each database item **X**.

- **W_TS(X)** is the largest timestamp of any transaction that executed **write(X)** successfully.
- **R_TS(X)** is the largest timestamp of any transaction that executed **read(X)** successfully.

**Basic Timestamp Ordering**

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 $\mathbf{TS(T_i) < TS(T_j)}$. 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 describes the Basic TO protocol in the following two cases.

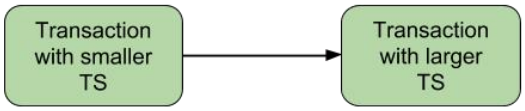
- Whenever a Transaction *T* issues a **W_item(X)** operation, check the following conditions:
  - If ***R_TS(X) > TS(T)*** and if ***W_TS(X) > TS(T)***, then abort and rollback T and reject the operation. else,
  - Execute W_item(X) operation of T and set W_TS(X) to TS(T).
- Whenever a Transaction *T* 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 set R_TS(X) to the larger of TS(T) and current R_TS(X).

Whenever the Basic TO algorithm detects two conflicting operations that occur in an incorrect order, it rejects the latter of the two operations by aborting the Transaction that issued it. Schedules produced by Basic TO are guaranteed to be *conflict serializable*. Already discussed that using Timestamp can ensure that our schedule will be *deadlock free*.

One drawback of the Basic TO protocol is that Cascading Rollback is still possible. Suppose we have a Transaction $T_1$ and $T_2$ has used a value written by $T_1$. If $T_1$ is aborted and resubmitted to

the system then, $T_2$ must also be aborted and rolled back. So the problem of Cascading aborts still prevails. Let's gist the Advantages and Disadvantages of Basic TO protocol:

- Timestamp Ordering protocol ensures serializability since the precedence graph will be of the form:




Precedence Graph for TS ordering

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may *not be cascade free*, and may not even be recoverable.

**Strict Timestamp Ordering**

A variation of Basic TO is called **Strict TO** ensures that the schedules are both Strict and Conflict Serializable. In this variation, a Transaction T that issues a R_item(X) or W_item(X) such that TS(T) > W_TS(X) has its read or write operation delayed until the Transaction **T'** that wrote the values of X has committed or aborted.

**Advantages of Timestamp Ordering Protocol**

- **High Concurrency:** Timestamp-based concurrency control allows for a high degree of concurrency by ensuring that transactions do not interfere with each other.
- **Efficient:** The technique is efficient and scalable, as it does not require locking and can handle a large number of transactions.
- **No Deadlocks:** Since there are no locks involved, there is no possibility of deadlocks occurring.
- **Improved Performance:** By allowing transactions to execute concurrently, the overall performance of the database system can be improved.

**Disadvantages of Timestamp Ordering Protocol**

- **Limited Granularity:** The granularity of timestamp-based concurrency control is limited to the precision of the timestamp. This can lead to situations where transactions are unnecessarily blocked, even if they do not conflict with each other.
- **Timestamp Ordering:** In order to ensure that transactions are executed in the correct order, the timestamps need to be carefully managed. If not managed properly, it can lead to inconsistencies in the database.
- **Timestamp Synchronization:** Timestamp-based concurrency control requires that all transactions have synchronized clocks. If the clocks are not synchronized, it can lead to incorrect ordering of transactions.
- **Timestamp Allocation:** Allocating unique timestamps for each transaction can be challenging, especially in distributed systems where transactions may be initiated at different locations.

**Multi-Version Concurrency Control (MVCC)**

Multi-Version Concurrency Control is a technology, utilized to enhance databases by resolving concurrency problems and also data locking by preserving older database versions. When many tasks attempt to update the same piece of data simultaneously, MVCC causes a conflict and necessitates a retry from one or more of the processes.

**Types of Multi-Version Concurrency Control (MVCC) in DBMS**

Below are some types of Multi-Version Concurrency Control (MVCC) in DBMS

- **Timestamp-based MVCC:** The data visibility to transactions is defined by the unique timestamp assigned to each transaction that creates a new version of a record.
- **Snapshot-based MVCC:** This utilizes the database snapshot that is created at the beginning of a transaction to supply the information that is needed for the transaction.
- **History-based MVCC**: This Keeps track of every modification made to a record, making transaction rollbacks simple.
- **Hybrid MVCC: This** coordinates data flexibility and performance by combining two or more MVCC approaches.

**How Does Multi-Version Concurrency Control (MVCC) Works?**

- In the database, every tuple has a version number. The tuple with the greatest version number can have a read operation done on it simultaneously.
- Only a copy of the record may be used for writing operations.
- While the copy is being updated concurrently, the user may still view the previous version.
- The version number is increased upon successful completion of the writing process.
- The upgraded version is now used for every new record operation and every time there is an update, this cycle is repeated.

**Implementation of Multi-Version Concurrency Control (MVCC) in DBMS**

- MVCC operates time stamps (TS) and increases transaction IDs to assure transactional consistency. MVCC manage many copies of the object, ensuring that a transaction (T) never has to wait to read a database object (P).
- A specific transaction Ti can read the most recent version of the object, which comes before the transaction's Read Timestamp RTS(Ti) since each version of object P contains both a Read Timestamp and a Write Timestamp.
- If there are other pending transactions to the same object that have an earlier Read Timestamp (RTS), then a Write cannot be completed.
- You cannot finish your checkout transaction until the person in front of you has finished theirs, much as when you are waiting in a queue at the shop.
- To reiterate, each object (P) has a Timestamp (TS). Should transaction Ti attempt to Write to an object and its Timestamp (TS) exceeds the object's current Read Timestamp, TS(Ti) < RTS(P), the transaction will be cancelled and retried.
- Ti makes a new copy of object P and sets its read/write timestamp (TS) to the transaction timestamp (TS ← TS(Ti)).

**Advantages of Multi-Version Concurrency Control (MVCC) in DBMS**

Below are some advantages of Multi-Version Concurrency Control in DBMS

- **The reduced read-and-write necessity for database locks:** The database can support many read-and-write transactions without locking the entire system thanks to MVCC.
- **Increased Concurrency:** This Enables several users to use the system at once.
- **Minimize read operation delays:** By enabling concurrent read operations, MVCC helps to cut down on read times for data.
- **Accuracy and consistency:** Preserves data integrity over several concurrent transactions.

**Disadvantages of Multi-Version Concurrency Control (MVCC) in DBMS**

Below are some disadvantages of Multi-Version Concurrency Control in DBMS

- **Overhead:** Keeping track of many data versions might result in overhead.
- **Garbage collecting:** To get rid of outdated data versions, MVCC needs effective garbage collecting systems.
- **Increase the size of the database:** Expand the capacity of the database since MVCC generates numerous copies of the records and/or tuples.
- **Complexity:** Compared to more straightforward locking systems, MVCC usage might be more difficult.

**Multiple Granularity Locking in DBMS**

The various Concurrency Control schemes have used different methods and every individual Data item is the unit on which synchronization is performed. A certain drawback of this technique is if a transaction $T_i$ needs to access the entire database, and a locking protocol is used, then $T_i$ must lock each item in the database. It is less efficient, it would be simpler if $T_i$ could use a single lock to lock the entire database. But, if it considers the second proposal, this should not in fact overlook certain flaws in the proposed method. Suppose another transaction just needs to access a few data items from a database, so locking the entire database seems to be unnecessary moreover it may cost us a loss of Concurrency, which was our primary goal in the first place.

Let's start by understanding what is meant by Granularity.

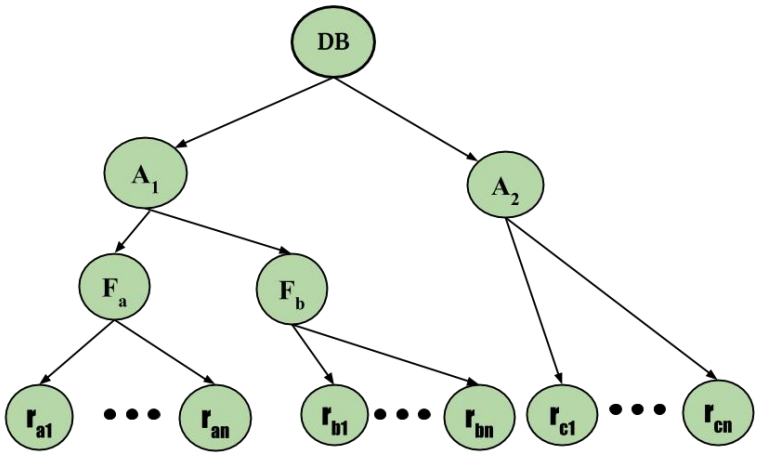
**Granularity**

It is the size of the data item allowed to lock. Now *Multiple Granularity* means hierarchically breaking up the database into blocks that can be locked and can be tracked what needs to lock and in what fashion. Such a hierarchy can be represented graphically as a tree.

**For example,** consider the tree, which consists of four levels of nodes. The highest level represents the entire database. Below it is nodes of type area; the database consists of exactly these areas. The area has children nodes which are called files. Every area has those files that are its child nodes. No file can span more than one area.

Finally, each file has child nodes called records. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file. Hence, the levels starting from the top level are:

- database
- area
- file
- record




Multi Granularity tree Hiererchy

Consider the above diagram for the example given, each node in the tree can be locked individually. As in the 2-phase locking protocol, it shall use shared and exclusive lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also implicitly locks all the descendants of that node in the same lock mode. For example, if transaction $T_i$ gets an explicit lock on file $F_c$ in exclusive mode, then it has an implicit lock in exclusive mode on all the records belonging to that file. It does not need to lock the individual

records of $F_c$ explicitly. this is the main difference between Tree-Based Locking and Hierarchical locking for multiple granularities.

Now, with locks on files and records made simple, how does the system determine if the root node can be locked? One possibility is for it to search the entire tree but the solution nullifies the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new lock mode, called *Intention lock mode*.

**Intention Mode Lock**

In addition to **S** and **X** lock modes, there are three additional lock modes with multiple granularities:

- **Intention-Shared (IS):** explicit locking at a lower level of the tree but only with shared locks.
- **Intention-Exclusive (IX):** explicit locking at a lower level with exclusive or shared locks.
- **Shared & Intention-Exclusive (SIX):** the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive mode locks.

The compatibility matrix for these lock modes are described below:

| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | ✔ | ✔ | ✔ | ✔ | ✗ |
| IX | ✔ | ✔ | ✗ | ✗ | ✗ |
| S | ✔ | ✗ | ✔ | ✗ | ✗ |
| SIX | ✔ | ✗ | ✗ | ✗ | ✗ |
| X | ✗ | ✗ | ✗ | ✗ | ✗ |

**IS** : Intention Shared
**IX** : Intention Exclusive
**S** : Shared
**X** : Exclusive
**SIX** : Shared & Intention Exclusive

Multi Granularity tree Hierarchy

The multiple-granularity locking protocol uses the intention lock modes to ensure serializability. It requires that a transaction $T_i$ that attempts to lock a node must follow these protocols:

- Transaction $T_i$ must follow the lock-compatibility matrix.
- Transaction $T_i$ must lock the root of the tree first, and it can lock it in any mode.
- Transaction $T_i$ can lock a node in S or IS mode only if $T_i$ currently has the parent of the node-locked in either IX or IS mode.
- Transaction $T_i$ can lock a node in X, SIX, or IX mode only if $T_i$ currently has the parent of the node-locked in either IX or SIX modes.
- Transaction $T_i$ can lock a node only if $T_i$ has not previously unlocked any node (i.e., $T_i$ is two-phase).
- Transaction $T_i$ can unlock a node only if $T_i$ currently has none of the children of the node-locked.

Observe that the multiple-granularity protocol requires that locks be acquired in top-down (root-to-leaf) order, whereas locks must be released in bottom-up (leaf to-root) order.

As an illustration of the protocol, consider the tree given above and the transactions:

- Say transaction $T_1$ reads record $R_{a2}$ in file $F_a$. Then, $T_1$ needs to lock the database, area $A_1$, and $F_a$ in IS mode (and in that order), and finally to lock $R_{a2}$ in S mode.
- Say transaction $T_2$ modifies record $R_{a9}$ in file $F_a$ . Then, $T_2$ needs to lock the database, area $A_1$, and file $F_a$ (and in that order) in IX mode, and at last to lock $R_{a9}$ in X mode.
- Say transaction $T_3$ reads all the records in file $F_a$. Then, $T_3$ needs to lock the database and area $A_1$ (and in that order) in IS mode, and at last to lock $F_a$ in S mode.
- Say transaction $T_4$ reads the entire database. It can do so after locking the database in S mode.

**Note** that transactions $T_1$, $T_3$, and $T_4$ can access the database concurrently. Transaction $T_2$ can execute concurrently with $T_1$, but not with either $T_3$ or $T_4$.

This protocol enhances concurrency and reduces lock overhead. Deadlock is still possible in the multiple-granularity protocol, as it is in the two-phase locking protocol. These can be eliminated by using certain deadlock elimination techniques.

**Validation Based Protocol in DBMS**

Validation Based Protocol is also called Optimistic Concurrency Control Technique. This protocol is used in DBMS (Database Management System) for avoiding concurrency in transactions. It is called optimistic because of the assumption it makes, i.e. very less interference occurs, therefore, there is no need for checking while the transaction is executed.

In this technique, no checking is done while the transaction is been executed. Until the transaction end is reached updates in the transaction are not applied directly to the database. All updates are applied to local copies of data items kept for the transaction. At the end of transaction execution, while execution of the transaction, a **validation phase** checks whether any of transaction updates violate serializability. If there is no violation of serializability the transaction is committed and the database is updated; or else, the transaction is updated and then restarted.

Optimistic Concurrency Control is a three-phase protocol. The three phases for validation based protocol:

**Read Phase:**

Values of committed data items from the database can be read by a transaction. Updates are only applied to local data versions.

**Validation Phase:**

Checking is performed to make sure that there is no violation of serializability when the transaction updates are applied to the database.

**Write Phase:**

On the success of the validation phase, the transaction updates are applied to the database, otherwise, the updates are discarded and the transaction is slowed down.

The idea behind optimistic concurrency is to do all the checks at once; hence transaction execution proceeds with a minimum of overhead until the validation phase is reached. If there is not much interference among transactions most of them will have successful validation, otherwise, results will be discarded and restarted later. These circumstances are not much favourable for optimization techniques, since, the assumption of less interference is not satisfied.

Validation based protocol is useful for rare conflicts. Since only local copies of data are included in rollbacks, cascading rollbacks are avoided. This method is not favourable for longer transactions because they are more likely to have conflicts and might be repeatedly rolled back due to conflicts with short transactions.

In order to perform the Validation test, each transaction should go through the various phases as described above. Then, we must know about the following three time-stamps that we assigned to transaction $T_i$, to check its validity:

**1. Start($T_i$):** It is the time when $T_i$ started its execution.
**2. Validation($T_i$):** It is the time when $T_i$ just finished its read phase and begin its validation phase.
**3. Finish($T_i$):** the time when $T_i$ end it's all writing operations in the database under write-phase.
Two more terms that we need to know are:
**1. Write_set:** of a transaction contains all the write operations that $T_i$ performs.
**2. Read_set:** of a transaction contains all the read operations that $T_i$ performs.
In the Validation phase for transaction $T_i$ the protocol inspect that $T_i$ doesn't overlap or intervene with any other transactions currently in their validation phase or in committed. The validation phase for $T_i$ checks that for all transaction $T_j$ one of the following below conditions must hold to being validated or pass validation phase:
**1. Finish($T_j$)<Starts($T_i$)**, since $T_j$ finishes its execution means completes its write-phase before $T_i$ started its execution(read-phase). Then the serializability indeed maintained.
**2.** $T_i$ begins its write phase after $T_j$ completes its write phase, and the read_set of $T_i$ should be disjoint with write_set of $T_j$.
**3.** $T_j$ completes its read phase before $T_i$ completes its read phase and both read_set and write_set of $T_i$ are disjoint with the write_set of $T_j$.
**Ex: Here two Transactions $T_i$ and $T_j$ are given, since TS($T_j$)<TS($T_i$)** so the validation phase succeeds in the Schedule-A. It's noteworthy that the final write operations to the database are performed only after the validation of both Ti and $T_j$. Since $T_i$ reads the old values of **x(12)** and **y(15)** while **print(x+y)** operation unless final write operation take place.

**Schedule-A**

| $T_j$ | $T_i$ |
|---|---|
| **r(x) // x=12** | |
| | **r(x)** |
| | **x=x-10** |
| | **r(y) //y=15** |
| | **y=y+10** |
| | **r(x)** |
| **<validate>** | |
| **print(x+y)** | |
| | **<validate>** |
| | **w(x)** |
| | **w(y)** |

**Schedule-A is a validated schedule**

**Advantages:**
**1. Avoid Cascading-rollbacks:** This validation based scheme avoid cascading rollbacks since the final write operations to the database are performed only after the transaction passes the validation phase. If the transaction fails then no updation operation is performed in the database. So no dirty read will happen hence possibilities cascading-rollback would be null.
**2. Avoid deadlock:** Since a strict time-stamping based technique is used to maintain the specific order of transactions. Hence deadlock isn't possible in this scheme.
**Disadvantages:**
**1. Starvation:** There might be a possibility of starvation for long-term transactions, due to a sequence of conflicting short-term transactions that cause the repeated sequence of restarts of the long-term transactions so on and so forth. To avoid starvation, conflicting transactions must be temporarily blocked for some time, to let the long-term transactions to finish.

**Recovery With Concurrent Transactions**

Concurrency control means that multiple transactions can be executed at the same time and then the interleaved logs occur. But there may be changes in transaction results so maintain the order of execution of those transactions.

During recovery, it would be very difficult for the recovery system to backtrack all the logs and then start recovering.

Recovery with concurrent transactions can be done in the following four ways.

1. Interaction with concurrency control
2. Transaction rollback
3. Checkpoints
4. Restart recovery

**Interaction with concurrency control :**

In this scheme, the recovery scheme depends greatly on the concurrency control scheme that is used. So, to rollback a failed transaction, we must undo the updates performed by the transaction.

**Transaction rollback :**

- In this scheme, we rollback a failed transaction by using the log.
- The system scans the log backward a failed transaction, for every log record found in the log the system restores the data item.

**Checkpoints :**

- Checkpoints is a process of saving a snapshot of the applications state so that it can restart from that point in case of failure.
- Checkpoint is a point of time at which a record is written onto the database form the buffers.
- Checkpoint shortens the recovery process.
- When it reaches 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 the next checkpoint and so on.
- The checkpoint is used to declare the point before which the DBMS was in the consistent state, and all the transactions were committed.

To ease this situation, 'Checkpoints' Concept is used by the most DBMS.

- In this scheme, we used checkpoints to reduce the number of log records that the system must scan when it recovers from a crash.
- In a concurrent transaction processing system, we require that the checkpoint log record be of the form <checkpoint L>, where **'L'** is a list of transactions active at the time of the checkpoint.
- A fuzzy checkpoint is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out.

**Restart recovery :**

- When the system recovers from a crash, it constructs two lists.
- The undo-list consists of transactions to be undone, and the redo-list consists of transaction to be redone.
- The system constructs the two lists as follows: Initially, they are both empty. The system scans the log backward, examining each record, until it finds the first <checkpoint> record.