**Unit 6 Concurrency Control**

Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another. Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical database, would have a mix of reading and WRITE operations and hence the concurrency is a challenge.

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions. Concurrency control protocols can be broadly divided into two categories −

- Lock based protocols
- Time stamp based protocols
- Validation Protocol

# 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/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.

  Read locks are shared because no data value is being changed.

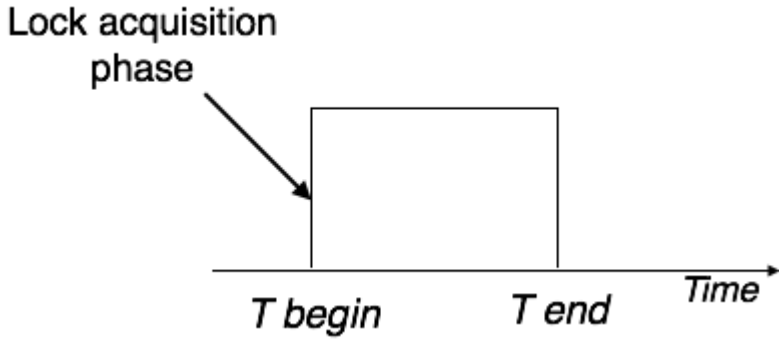
There are four types of lock protocols available −

**Simplistic Lock Protocol**

Simplistic lock-based protocols allow transactions to obtain a lock on every object before a 'write' operation is performed. Transactions may unlock the data item after completing the ‘write’ operation.

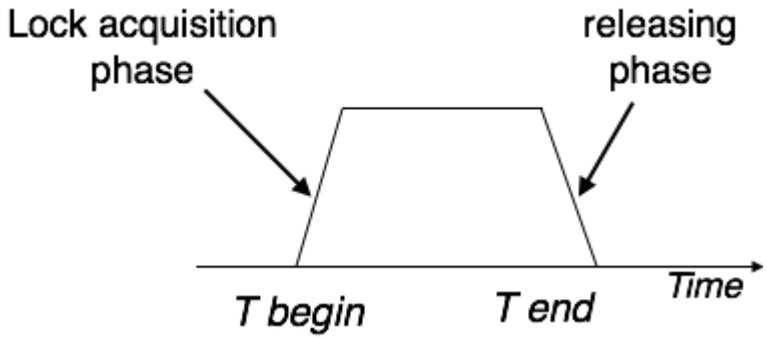
**Pre-claiming Lock Protocol**

Pre-claiming protocols evaluate their operations and create a list of data items on which they need locks. Before initiating an execution, the transaction requests the system for all the locks it needs beforehand. If all the locks are granted, the transaction executes and releases all the locks when all its operations are over. If all the locks are not granted, the transaction rolls back and waits until all the locks are granted.




**Two-Phase Locking 2PL**

This locking protocol divides the execution phase of a transaction into three parts. In the first part, when the transaction starts executing, it seeks permission for the locks it requires. The second part is where the transaction acquires all the locks. As soon as the transaction releases its first lock, the third phase starts. In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.

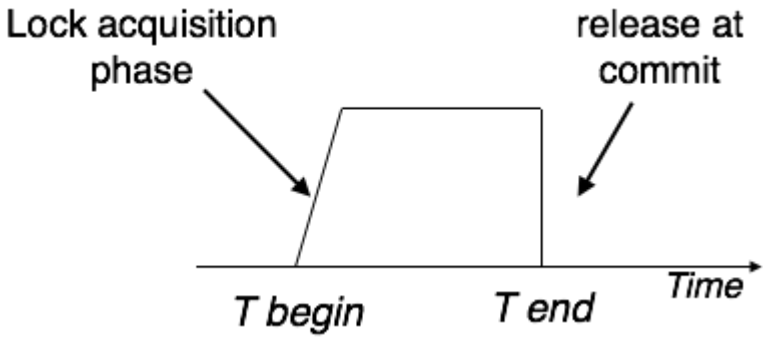



Two-phase locking has two phases, one is **growing**, where all the locks are being acquired by the transaction; and the second phase is shrinking, where the locks held by the transaction are being released.

To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

**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 lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.



# Timestamp-based Protocols

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses—

i). System time

OR

ii). Logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution.

In timestamp-based protocols start working as soon as a transaction is created.

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.

# Timestamp Ordering Protocol

The timestamp-ordering protocol ensures serializability among transactions in their conflicting read and writes operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

- The timestamp of transaction $T_i$ is denoted as $TS(T_i)$.
- Read time-stamp of data-item X is denoted by R-timestamp(X).
- Write time-stamp of data-item X is denoted by W-timestamp(X).

Timestamp ordering protocol works as follows −

- **If a transaction Ti issues a read(X) operation −**
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected.
  - If TS(Ti) >= W-timestamp(X)
    - Operation executed.
  - All data-item timestamps updated.
- **If a transaction Ti issues a write(X) operation −**
  - If TS(Ti) < R-timestamp(X)
    - Operation rejected.
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected and Ti rolled back.
  - Otherwise, operation executed.

# Validation Protocol

• Execution of transaction Ti is done in three phases.

1. Read and execution phase: Transaction Ti writes only to temporary local variables 2. Validation phase: Transaction Ti performs a “validation test” to determine if local variables can be written without violating serializability.

3. Write phase: If Ti is validated, the updates are applied to the database; otherwise, Ti is rolled back.

• The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.

• Each transaction Ti has 3 timestamps –

**Start (Ti)**: the time when Ti started its execution –

**Validation (Ti)**: the time when Ti entered its validation phase –

**Finish (Ti):** the time when Ti finished its write phase

• Serializability order is determined by timestamp given at validation time, to increase concurrency. Thus TS(Ti) is given the value of Validation(Ti).

• This protocol is useful and gives greater degree of concurrency if probability of conflicts is low. That is because the serializability order is not pre-decided and relatively less transactions will have to be rolled back.

Validation Test for Transaction Tj

• If for all Ti with TS (Ti) < TS (Tj)

either one of the following condition holds: –

finish(Ti) < start(Tj)

– start(Tj) < finish(Ti) < validation(Tj)

The set of data items written by Ti does not intersect with the set of data items read by Tj.

Then validation succeeds and Tj can be committed. Otherwise, validation fails and Tj is aborted.

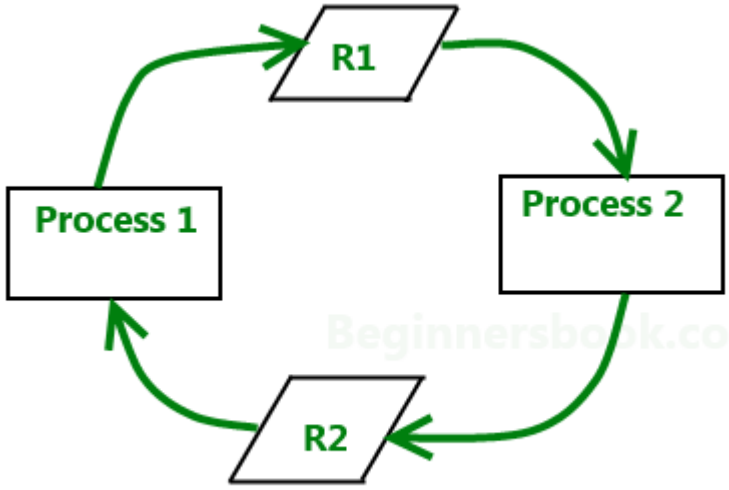
• Justification: Either first condition is satisfied, and there is no overlapped execution, or second condition is satisfied and

1. The writes of Tj do not affect reads of Ti since they occur after Ti has finished its reads.

2. The writes of Ti do not affect reads of Tj since Tj does not read any item written by Ti.

## Deadlock

A **deadlock** is a condition wherein two or more tasks are waiting for each other in order to be finished but none of the task is willing to give up the resources that other task needs. In this situation no task ever gets finished and is in waiting state forever.




**Process P1 holds resource R2 and requires R1 while Process P2 holds resource R1 and requires R2.**

## Cause of Deadlock (Coffman conditions)

Coffman stated four conditions for a deadlock occurrence. A deadlock may occur if all the following conditions holds true.

- **Mutual exclusion condition**: There must be at least one resource that cannot be used by more than one process at a time.
- **Hold and wait condition**: A process that is holding a resource can request for additional resources that are being held by other processes in the system.
- **No pre-emption condition**: A resource cannot be forcibly taken from a process. Only the process can release a resource that is being held by it.
- **Circular wait condition**: A condition where one process is waiting for a resource that is being held by second process and second process is waiting for third process ….so on and the last process is waiting for the first process. Thus making a circular chain of waiting.

## Deadlock prevention

We have learnt that if all the four Coffman conditions hold true then a deadlock occurs so preventing one or more of them could prevent the deadlock.

- **Removing mutual exclusion**: All resources must be sharable that means at a time more than one processes can get a hold of the resources. That approach is practically impossible.
- **Removing hold and wait condition**: This can be removed if the process acquires all the resources that are needed before starting out. Another way to remove this to enforce a rule of requesting resource when there are none in held by the process.
- **Preemption of resources**: Preemption of resources from a process can result in rollback and thus this needs to be avoided in order to maintain the consistency and stability of the system.
- **Avoid circular wait condition**: This can be avoided if the resources are maintained in a hierarchy and process can hold the resources in increasing order of precedence. This avoid circular wait. Another way of doing this to force one resource per process rule – A process can request for a resource once it releases the resource currently being held by it. This avoids the circular wait.

**There are two technique**:

**Wait-Die Scheme**

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur −

- If TS(T1) < TS(T2) − that is T1, which is requesting a conflicting lock, is older than T2 − then T1 is allowed to wait until the data-item is available.
- If TS(T1) > TS(T2) − that is T1 is younger than T2 − then T1 dies. T1 is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

**Wound-Wait Scheme**

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

- If TS(T1) < TS(T2), then T2 forces T2 to be rolled back − that is T1 wounds T2. T2 is restarted later with a random delay but with the same timestamp.
- If TS(T1) > TS(T2), then T1 is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

## Ignore the deadlock (Ostrich algorithm)

Did that made you laugh? You may be wondering how ignoring a deadlock can come under deadlock handling. But to let you know that the windows you are using on your PC, uses this approach of deadlock handling and that is reason sometimes it hangs up and you have to reboot it to get it working. Not only Windows but UNIX also uses this approach.

Well! Let me answer the second question first, This is known as Ostrich algorithm because in this approach we ignore the deadlock and pretends that it would never occur, just like Ostrich behavior “to stick one’s head in the sand and pretend there is no problem.”

**Let’s discuss why we ignore it**: When it is believed that deadlocks are very rare and cost of deadlock handling is higher, in that case ignoring is better solution than handling it. For example: Let’s take the operating system example – If the time requires handling the deadlock is higher than the time requires rebooting the windows then rebooting would be a preferred choice considering that deadlocks are very rare in windows.

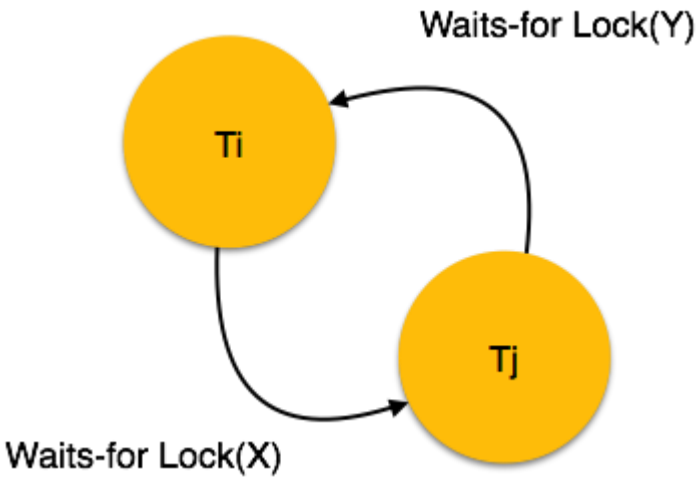
# Deadlock Avoidance

Aborting a transaction is not always a practical approach. Instead, deadlock avoidance mechanisms can be used to detect any deadlock situation in advance. Methods like "wait-for graph" are available but they are suitable for only those systems where transactions are lightweight having fewer instances of resource. In a bulky system, deadlock prevention techniques may work well.

Wait-for Graph

This is a simple method available to track if any deadlock situation may arise. For each transaction entering into the system, a node is created. When a transaction $T_i$ requests for a lock on an item, say X, which is held by some other transaction $T_j$, a directed edge is created from $T_i$ to $T_j$. If $T_j$ releases item X, the edge between them is dropped and $T_i$ locks the data item.

The system maintains this wait-for graph for every transaction waiting for some data items held by others. The system keeps checking if there's any cycle in the graph.




Here, we can use any of the two following approaches −

- First, do not allow any request for an item, which is already locked by another transaction. This is not always feasible and may cause starvation, where a transaction indefinitely waits for a data item and can never acquire it.
- The second option is to roll back one of the transactions. It is not always feasible to roll back the younger transaction, as it may be important than the older one. With the help of some relative algorithm, a transaction is chosen, which is to be aborted. This transaction is known as the **victim** and the process is known as **victim selection**.

**Database Failure and Recovery**

**Database Failure:**

# Failure Classification

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

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.

# Storage Structure

We have already described the storage system. In brief, 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

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 fail safe.

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

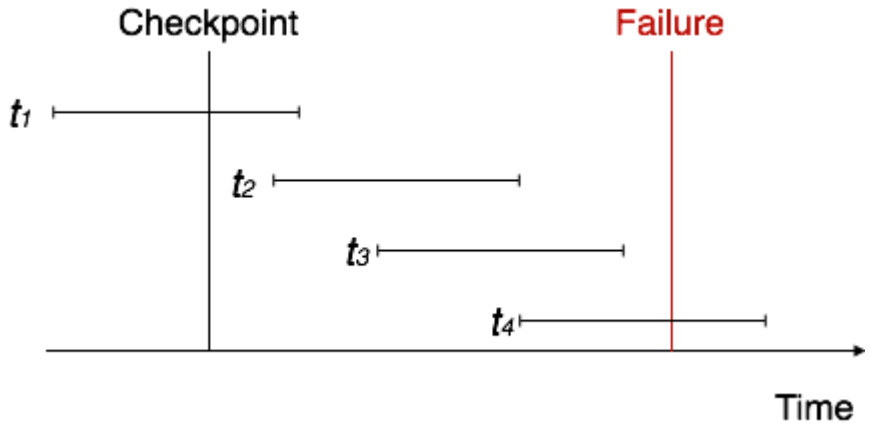
When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

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 −

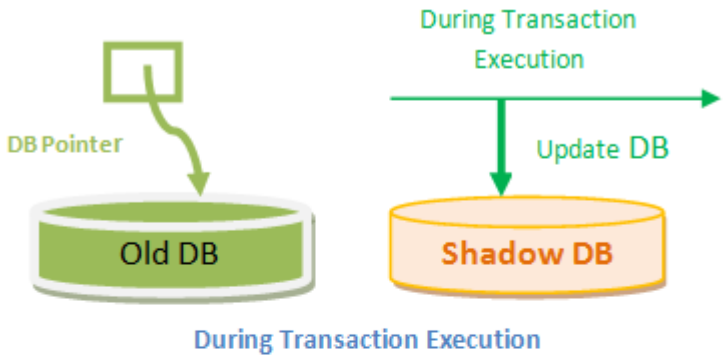


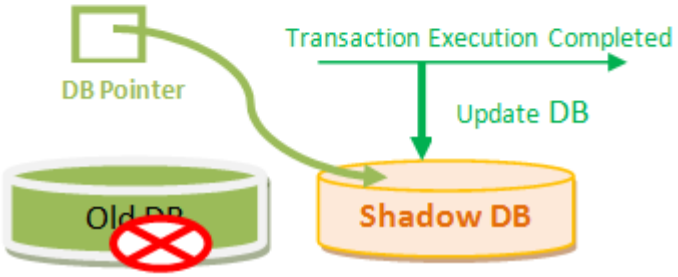

- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with <$T_n$, Start> and <$T_n$, Commit> or just <$T_n$, Commit>, it puts the transaction in the redo-list.
- If the recovery 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.

**Shadow paging**

This is the method where all the transactions are executed in the primary memory or the shadow copy of database. Once all the transactions completely executed, it will be updated to the database. Hence, if there is any failure in the middle of transaction, it will not be reflected in the database. Database will be updated after all the transaction is complete.




A database pointer will be always pointing to the consistent copy of the database, and copy of the database is used by transactions to update. Once all the transactions are complete, the DB pointer is modified to point to new copy of DB, and old copy is deleted. If there is any failure during the transaction, the pointer will be still pointing to old copy of database, and shadow database will be deleted. If the transactions are complete then the pointer is changed to point to shadow DB, and old DB is deleted.

As we can see in above diagram, the DB pointer is always pointing to consistent and stable database. This mechanism assumes that there will not be any disk failure and only one transaction executing at a time so that the shadow DB can hold the data for that transaction. It is useful if the DB is comparatively small because shadow DB consumes same memory space as the actual DB. Hence it is not efficient for huge DBs. In addition, it cannot handle concurrent execution of transactions. It is suitable for one transaction at a time.