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

Lock Based Protocols

A lock is a variable associated with a data item that describes a status of data item with respect to possible operation 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.

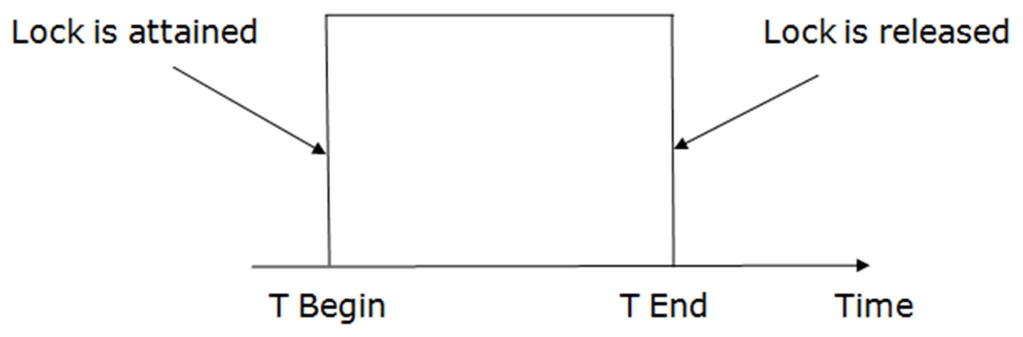
There are two types of lock:

1. Shared lock:

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

Exclusive lock:

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



Timestamp based Protocols

The timestamp-based algorithm uses a timestamp to serialize the execution of concurrent transactions. This protocol ensures that every conflicting read and write operations are executed in timestamp order. The protocol uses the **System Time or Logical Count as** a Timestamp.

The older transaction is always given priority in this method. It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.

Lock-based protocols help you to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

Example:

Suppose there are there transactions T1, T2, and T3.

T1 has entered the system at time 0010

T2 has entered the system at 0020

T3 has entered the system at 0030

Priority will be given to transaction T1, then transaction T2 and lastly Transaction T3.

Advantages:

- Schedules are serializable just like 2PL protocols
- No waiting for the transaction, which eliminates the possibility of deadlocks!

Disadvantages:

Starvation is possible if the same transaction is restarted and continually aborted

Validation based Protocols

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 transaction. At the end of transaction execution, while execution of 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:

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

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

3. Write Phase:
   On the success of 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 favorable 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 is included in rollbacks, cascading rollbacks are avoided. This method is not favorable for longer transactions because they are more likely to have conflicts and might be repeatedly rolled back due to conflicts with short transactions.

Deadlock Handling

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.

Deadlock Handling in Centralized Systems

There are three classical approaches for deadlock handling, namely –

- Deadlock prevention.
- Deadlock avoidance.
- Deadlock detection and removal.

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.

Deadlock Avoidance

Deadlock can be avoided if resources are allocated in such a way that it avoids the deadlock occurrence. There are two algorithms for deadlock avoidance.

- Wait/Die
- Wound/Wait

Here is the table representation of resource allocation for each algorithm. Both of these algorithms take process age into consideration while determining the best possible way of resource allocation for deadlock avoidance.

	Wait/Die	Wound/Wait
Older process needs a resource held by younger process	**Older process** waits	**Younger process** dies
Younger process needs a resource held by older process	**Younger process** dies	**Younger process** waits

Once of the famous deadlock avoidance algorithm is **Banker's algorithm**

Deadlock Detection –
When a transaction waits indefinately to obtain a lock, The database managememt system should detect whether the transaction is involved in a deadlock or not.

Wait-for-graph is one of the methods for detecting the deadlock situation. This method is suitable for smaller database. In this method a graph is drawn based on the transaction and their lock on the resource. If the graph created has a closed loop or a cycle, then there is a deadlock.

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

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

Reasons for a transaction failure could be -

1. **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.
2. **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. **For example,** The system aborts an active transaction, in case of deadlock or resource unavailability.

System Crash

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

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

Disk Failure

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

Storage Structure

Database tables and indexes may be stored on disk in one of a number of forms, including ordered/unordered flat files, ISAM, heap files, hash buckets, or B+ trees. Each form has its own particular advantages and disadvantages. The most commonly used forms are B+ trees and ISAM. Such forms or structures are one aspect of the overall schema used by a database engine to store information.

Unordered Flat Files

Unordered storage typically stores the records in the order they are inserted. Such storage offers good insertion efficiency O(1), but inefficient retrieval times O(n). Typically these retrieval times are better, however, as most databases use indexes on the primary keys, resulting in retrieval times of O(log n)or O(1) for keys that are the same as the database row offsets within the storage system.

Ordered Flat Files

Ordered storage typically stores the records in order and may have to rearrange or increase the file size when a new record is inserted, resulting in lower insertion efficiency. However, ordered storage provides more efficient retrieval as the records are pre-sorted, resulting in a complexity of O (log n)

Heap files

Heap files are lists of unordered records of variable size. Although sharing a similar name, heap files are widely different from in-memory heaps. In-memory heaps are ordered, as opposed to heap files.

- Simplest and most basic method
  - insert efficient, with new records added at the end of the file, providing chronological order
  - retrieval efficient when the handle to the memory is the address of the memory
  - search inefficient, as searching has to be linear
  - deletion is accomplished by marking selected records as "deleted"
  - requires periodic reorganization if file is very volatile (changed frequently)
- Advantages
  - efficient for bulk loading data
  - efficient for relatively small relations as indexing overheads are avoided
  - efficient when retrievals involve large proportion of stored records

- Disadvantages
  - not efficient for selective retrieval using key values, especially if large
  - sorting may be time-consuming
  - not suitable for volatile tables

Hash buckets

- Hash functions calculate the address of the page in which the record is to be stored based on one or more fields in the record
  - hashing functions chosen to ensure that addresses are spread evenly across the address space
  - 'occupancy' is generally 40% to 60% of the total file size
  - unique address not guaranteed so collision detection and collision resolution mechanisms are required
- Open addressing
- Chained/unchained overflow
- Pros and cons
  - efficient for exact matches on key field
  - not suitable for range retrieval, which requires sequential storage
  - calculates where the record is stored based on fields in the record
  - hash functions ensure even spread of data
  - collisions are possible, so collision detection and restoration is required

B+ trees

These are the most commonly used in practice.

- Time taken to access any record is the same because the same number of nodes is searched
- Index is a full index so data file does not have to be ordered
- Pros and cons
  - versatile data structure – sequential as well as random access
  - access is fast
  - supports exact, range, part key and pattern matches efficiently.
  - volatile files are handled efficiently because index is dynamic – expands and contracts as table grows and shrinks
  - less well suited to relatively stable files – in this case, ISAM is more efficient

ISAM

ISAM (an acronym for **indexed sequential access method**) is a method for creating, maintaining, and manipulating computer files of data so that records can be retrieved sequentially or randomly by one or more keys. Indexes of key fields are maintained to achieve fast retrieval of required file records in Indexed files. IBM originally developed ISAM for mainframe computers, but implementations are available for most computer systems. The term *ISAM* is used for several related concepts:

- The IBM ISAM product and the algorithm it employs
- A database system where an application developer directly uses an application programming interface to search indexes in order to locate records in data files. In contrast, a relational database uses a query optimizer which automatically selects indexes
- An indexing algorithm that allows both sequential and keyed access to data. Most databases use some variation of the B-tree for this purpose, although the original IBM ISAM and VSAM implementations did not do so.
- Most generally, any index for a database. Indexes are used by almost all databases.

Recovery and Atomicity

The process of restoring the database to a correct state in the event of a failure is known as database recovery. It is a service to be provided by DBMS to ensure the database reliability. Reliability here means both, the resilience of DBMS to various types of failure and its capability to recover from them.

The database has to be restored back to consistent state from its inconsistent state that has been caused due to failure.

CATEGORIZATION OF RECOVERY ALGORITHMS

Conceptually, there are two main techniques for recovery from non-catastrophic transaction failures:

1. Deferred update (or NO UNDO/REDO) algorithm.

2. Immediate update (or UNDO/REDO) algorithm

Deferred update : These techniques do not physically update on disk until after a transaction reaches its commit point. Then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction work space or buffers. During commit, the updates are first recorded persistently in the log and then written to the database. 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 of a committed transaction from the log, because their effect yet have been recorded in the database. Hence, deferred update is also known as the NOUNDO/REDO algorithm.

Immediate update : The database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are typically recorded in the log on disk by force writing before they are applied to the database , making recovery still possible. If a transaction fails after recording some changes in the database but before reaching its commit point, the effect of its operations on the database must be undone i.e. the transaction must be rolled back. Here both undo and redo may be required during recovery. This technique is known as UNDO/REDO algorithm.

Log based Recovery

A **log** is the most widely used recording database modification technique. The log is a structure used for recording database modification. It is a sequence of log record recording all the update activities in the database. We can use this log to recover from failure and to bring back the database to the consistent state. An update log record contains various fields:

- **Transaction identifier:** It is the identifier which uniquely identifies the transaction.
- **Data item identifier:** It is the identifier which uniquely identifies the data to be used.
- **Old value:** It is the value of the data item before the write operation.
- **New value:** It is the value of the data item after performing the write operation. There are many types of log record we denote the various types of log records:

    1. <Ti start> transaction Ti has started.
    2. <Ti, Xi, V1, V2> transaction Ti has performed a write on data item Xi, Xj has value v1 before the write and will have value V2 after the write operation.
    3. <Ti commit> transaction Ti has committed.
    4. <Ti abort> transaction Ti ha aborted.

Whenever a transaction performs write operation database is modified only when log record for that write is created. And logs may or may not contain the record of reading of data item. It is so because reading the data item does not affect the consistency of the database and is nowhere helpful in recovery mechanism.

In the **recovery method**, we use two operations:

1. **Undo(Ti):** Restores the values of all data item updated by transaction Ti to the old values.
2. **Redo(Ti):** Sets the value of all data item updated by transaction Ti to the new values. We need to undo a transaction T only when log contains the record <T start> only when log contains the record <start> and <T commit> both.

Transaction modification technique

There are mainly two techniques to ensure the transaction atomicity despite failure are as follow:

Immediate database modification

The immediate update technique allows database modification to output the database while the transaction is still in the active state. Data modification written by an active transaction is called uncommitted modification.

When a system crashes or a transaction fails, the old value of the data item should be used for bringing the database into the consistent state. This can be done by undoing operation. Before a transaction, Ti starts its execution the record <Ti start> is written to the log. During its execution and write(X) operation by Ti is preceded by the writing of the appropriate new update record to the log. When Ti partially commits, the record <Ti commit> is written to the log.

Example

```
<To start>
<To A, 1000, 950>
<To B, 2000, 2050>
<To commit>
<T1 start>
<T1 C, 700, 600>
<Ti commit>
```

We consider an example of banking system taken earlier for transaction To and T1 such that To is followed by T1. If the system crash occurs just after the log record and during recovery we do redo (To) and undo (T1) as we have both < To start > and <To commit> in the log record. But we do not have <T1 commit> with <T1 start> in log record. Undo(T1) should be done first then redo (To) should be done.

Deferred modification technique

Deferred database modification technique ensures transaction atomicity by recording all database modification in the log. In this technique, all the write statements of the transaction are applied to the database only when the transaction is partially committed. A transaction is said to be partially committed once the final action of the transaction has been executed.

When a transaction partially commits, then the information on the log associated with the transaction is used in executing the deferred writes. If the system crashes before the transaction completes its execution or if the transaction abort then the information on the log is ignored. Using the log the system can handle that result in the loss of information on volatile storage. The recovery scheme uses the procedure.

Redo(Ti) sets the value of all data item updated by transaction Ti to the new values. The set of data item updated by Ti and their respective new values can be found in the log. The redo transaction must be idempotent. That an executing it several times must be equivalent to executing it once.

After a failure, the recovery subsystem consults the log to determine which transaction need to to be redone. Transaction Ti is redone if and only if the log record contains both **<Ti start>** and **<Ti commit>** statements.

Example

```
(A)            (B)
<To start>     <To start>
<To A, 950>    <To A , 950>
<To B, 205>    <To B, 2050>
               <To commit>
               <T1 start>
               <T1 C, 600>
```

If a system fails just after the log record for the step write(B) of transaction To. The during recovery no redo operation will be done as we have only <To start> in log record but not <To commit>.

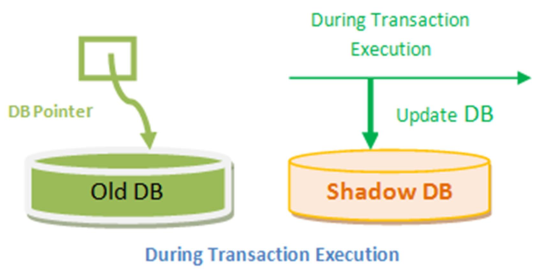
If a system crash occurs just after the log record write C. the during recovery only redo (To) is done as we have only <To start> and <To commit> in log disk. At the same time we have <T1 start> in log disk but not <T1 commit> so redo (T1) will not be done.

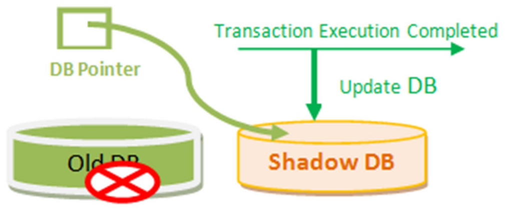
```
(C)
   <To start>
   <To A , 950>
   <To B, 2050>
   <To commit>
   <T1 start>
   <T1 C, 600>
   <T1 commit>
```

If a system crash occurs just after the log record <T1 commit> the during the recovery we will perform both redo (T0) and redo (TI) as we have both <To start> <To commit> and <T1 start>, <T1 commit> in log disk

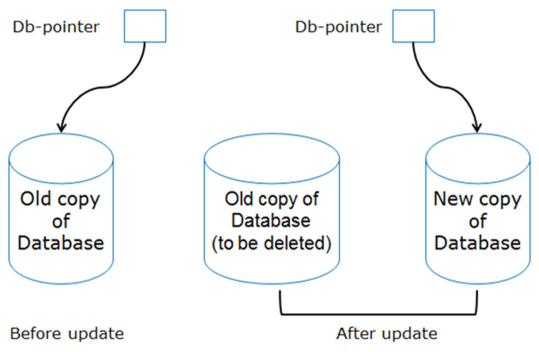
Shadow Paging

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.



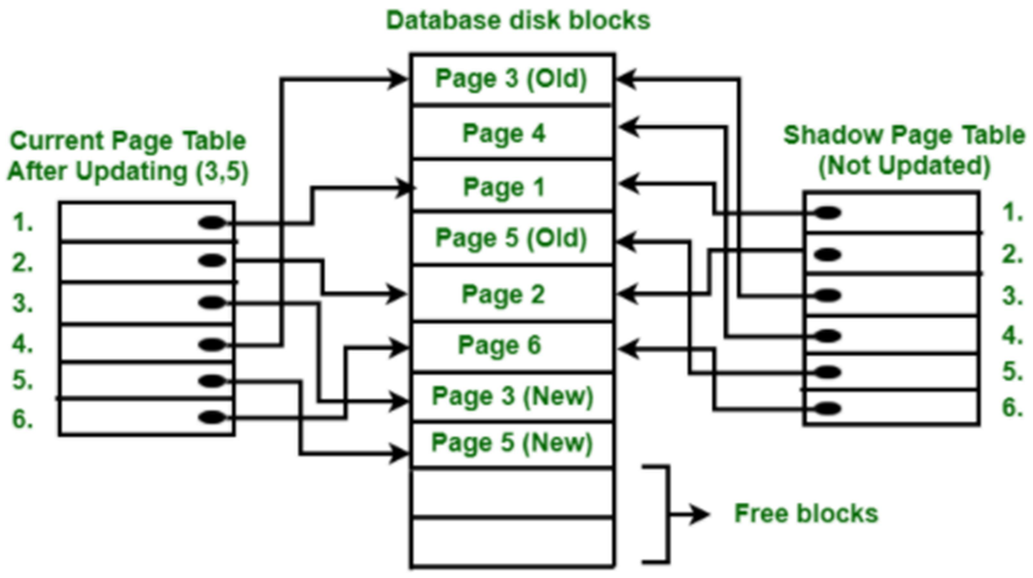
Shadow-copy technique for atomicity and durability

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.

Shadow Paging is recovery technique that is used to recover database. In this recovery technique, database is considered as made up of fixed size of logical units of storage which are referred as **pages.** pages are mapped into physical blocks of storage, with help of the **page table** which allow one entry for each logical page of database. This method uses two page tables named **current page table** and **shadow page table**.

The entries which are present in current page table are used to point to most recent database pages on disk. Another table i.e., Shadow page table is used when the transaction starts which is copying current page table. After this, shadow page table gets saved on disk and current page table is going to be used for transaction. Entries present in current page table may be changed during execution but in shadow page table it never get changed. After transaction, both tables become identical.



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

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

4. The changes are now propagated to Page 3 (New) which is pointed by current page table.

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

1. All the modifications which are done by transaction which are present in buffers are transferred to physical database.

2. Output current page table to disk.

3. Disk address of current page table output to fixed location which is in stable storage containing address of shadow page table. This operation overwrites address of old shadow page table. With this current page table becomes same as shadow page table and transaction is committed.

Failure :
If system crashes during execution of transaction but before commit operation, With this, it is sufficient only to free modified database pages and discard current page table. Before execution of transaction, state of database get recovered by reinstalling shadow page table.

If the crash of system occur after last write operation then it does not affect propagation of changes that are made by transaction. These changes are preserved and there is no need to perform redo operation.

Advantages :

- This method require fewer disk accesses to perform operation.
- In this method, recovery from crash is inexpensive and quite fast.
- There is no need of operations like- Undo and Redo.

Disadvantages :

- Due to location change on disk due to update database it is quite difficult to keep related pages in database closer on disk.
- During commit operation, changed blocks are going to be pointed by shadow page table which have to be returned to collection of free blocks otherwise they become accessible.
- The commit of single transaction requires multiple blocks which decreases execution speed.
- To allow this technique to multiple transactions concurrently it is difficult.