**DBMS 🡪 UNIT-3**

**Embedded SQL**

* The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.
* A language to which SQL queries are embedded is referred to as a host language, and the SQL structures permitted in the host language comprise embedded SQL.
* The basic form of these languages follows that of the System R embedding of SQL into PL/I.
* EXEC SQL statement is used to identify embedded SQL request to the preprocessor

EXEC SQL <embedded SQL statement > END\_EXEC

Note: this varies by language (for example, the Java embedding uses # SQL { …. }; )

**Example Query**

From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable credit\_amount.

Specify the query in SQL and declare a *cursor* for it

EXEC SQL

**declare** *c* **cursor for   
 select** *ID, name* **from** *student* **where tot\_cred** *> :credit\_amount*

END\_EXEC

* The open statement causes the query to be evaluated

EXEC SQL open c END\_EXEC

* The fetch statement causes the values of one tuple in the query result to be placed on host language variables.

EXEC SQL fetch c into :si, :sn END\_EXEC  
Repeated calls to fetch get successive tuples in the query result

* A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to ‘02000’ to indicate no more data is available
* The close statement causes the database system to delete the temporary relation that holds the result of the query.

EXEC SQL close c END\_EXEC

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

**Updates Through Cursors**

* Can update tuples fetched by cursor by declaring that the cursor is for update

**declare** *c* **cursor for  
 select** \*   
 **from** *instructor* **where** *dept\_name* = ‘Music’  
 **for update**

* To update tuple at the current location of cursor *c*

**update** *instructor* **set** *salary = salary* + 100  
 **where current of** *c*

**JDBC**

* JDBC is a Java API for communicating with database systems supporting SQL.
* JDBC supports a variety of features for querying and updating data, and for retrieving query results.
* JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
* Model for communicating with the database:
* Open a connection
* Create a “statement” object
* Execute queries using the Statement object to send queries and fetch results
* Exception mechanism to handle errors

**Creating JDBC Application**

There are following six steps involved in building a JDBC application −

* **Import the packages:** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.\** will suffice.
* **Register the JDBC driver:** Requires that you initialize a driver so you can open a communication channel with the database.
* **Open a connection:** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
* **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.
* **Extract data from result set:** Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
* **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

**JDBC Sample Code**

public static void JDBCexample(String dbid, String userid, String passwd)

{

try {

Class.forName ("oracle.jdbc.driver.OracleDriver");

Connection conn = DriverManager.getConnection(   
 "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);

Statement stmt = conn.createStatement();

… Do Actual Work ….

stmt.close();

conn.close();

}

catch (SQLException sqle) {

System.out.println("SQLException : " + sqle);

}

}

**Update to database**

try {  
 stmt.executeUpdate(  
 "insert into instructor values(’77987’, ’Kim’, ’Physics’, 98000)");  
} catch (SQLException sqle)  
{  
 System.out.println("Could not insert tuple. " + sqle);  
}

**Execute query and fetch and print results**

ResultSet rset = stmt.executeQuery(  
 "select dept\_name, avg (salary) from instructor group by dept\_name");  
while (rset.next()) {  
 System.out.println(rset.getString("dept\_name") + " " +  
 rset.getFloat(2));  
}

**Getting result fields:**

rs.getString(“dept\_name”) and rs.getString(1) equivalent if dept\_name is the first argument of select result.

**Dealing with Null values**

int a = rs.getInt(“a”);

if (rs.wasNull()) Systems.out.println(“Got null value”);

### ****Dynamic SQL****

When the pattern of database access is known in advance then static SQL is very adequate to serve us. Sometimes, in many applications we may not know the pattern of database access in advance. For example, a report writer must be able to decide at run time that which SQL statements will be needed to access the database. Such a need can’t be fulfilled with static SQL and requires an advanced form of static SQL known as dynamic SQL.

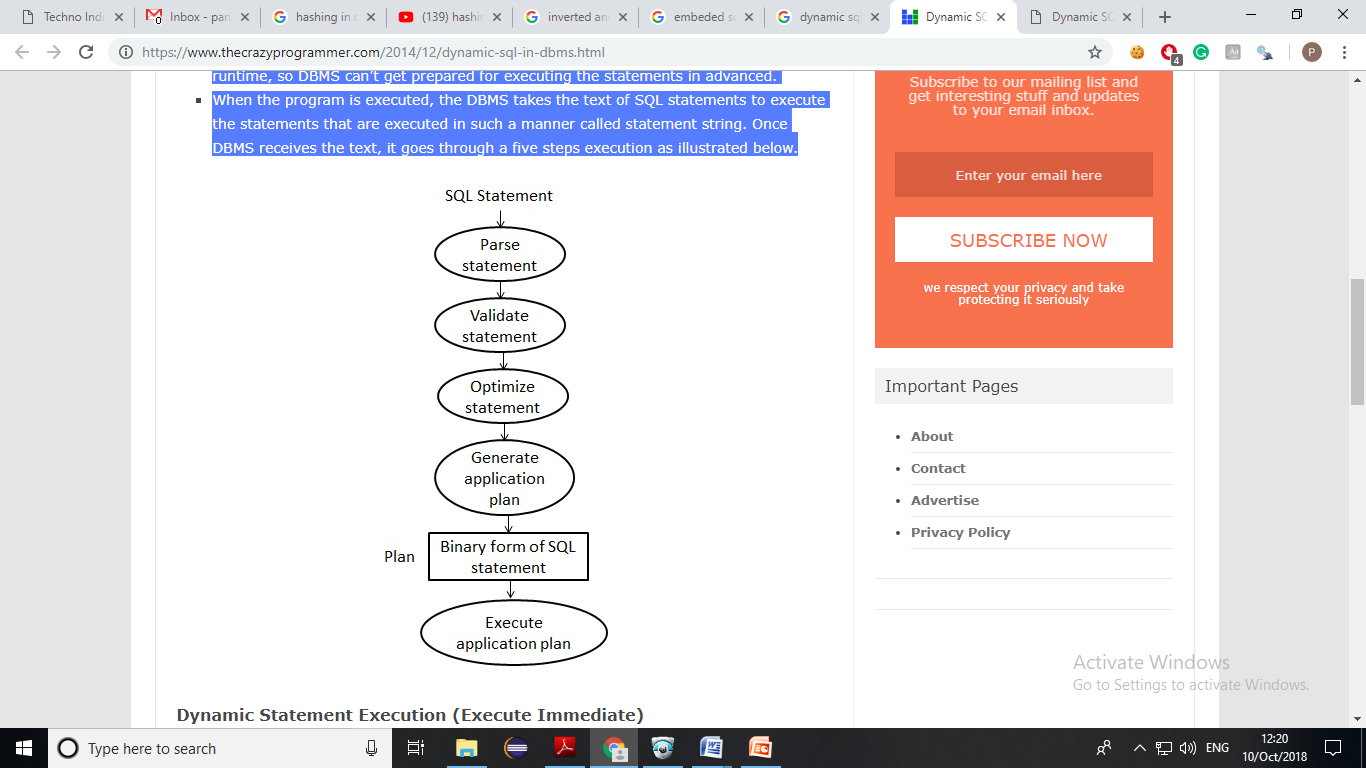
There are several limitations in static SQL. Although using the host variables (host variables allows us to input values for search condition at run time), we can achieve a little bit dynamicness, for e.g.,

**exec sql select tname, sex from teacher where salary > :sal;**

Here the salary will be asked on run time. But getting column name or table asked at run time not possible with embedded SQL. For having such a feature we need dynamic SQL.

**Dynamic SQL Concepts**

* In dynamic SQL, the SQL statements are not hard coded in the programming language. The text of the SQL statement is asked at the run time to the user.
* In dynamic SQL, the SQL statements that are to be executed are not known until runtime, so DBMS can’t get prepared for executing the statements in advanced.
* When the program is executed, the DBMS takes the text of SQL statements to execute the statements that are executed in such a manner called statement string. Once DBMS receives the text, it goes through a five steps execution as illustrated below.



### Dynamic Statement Execution (Execute Immediate)

### The Execute Immediate statement provides the simplest form of dynamic SQL. This statement passes the text of SQL statements to DBMS and asks the DBMS to execute the SQL statements immediately.

For using the statement our program goes through the following steps.

1. The program constructs a SQL statement as a string of text in one of its data areas (called a buffer).
2. The program passes the SQL statements to the DBMS with the EXECUTE IMMEDIATE statement.
3. The DBMS executes the statement and sets the SQL CODE/SQL STATE values to flag the finishing status same like if the statement had been hard coded using static SQL.

# Form

In a database context, a form is a window or screen that contains numerous fields, or spaces to enter data. Each field holds a field label so that any user who views the form gets an idea of its contents. A form is more user friendly than generating queries to create tables and insert data into fields.

Databases like SQL and Oracle do not use built-in forms, employing the query option for data creation and manipulation instead. This makes querying knowledge essential to handling these databases. Microsoft Access, however, uses forms for data entry, making it more user friendly than its counterparts. The fields and field labels are organized in a logical manner for easy form access and manipulation.

While making an entry into fields of a form, it is important to be careful about the field types, which are generally, set when the form is created. As such, an attempt to enter values that do not satisfy the field constraints will fail. For instance, a field with a field type "not null" does not take null values and cannot be left blank. Some fields may also have table relationships connected by a foreign key; any alteration of such a field in a form needs special care.

Some online databases also have built-in form templates included in them. These databases can easily be used by users who are not even aware of scripting, as all entries and changes can be made in the form by a single click. A user is also free to customize the form layout by selecting suitable form templates.

# Database Report

A database report is the formatted result of database queries and contains useful data for decision-making and analysis.

Most good business applications contain a built-in reporting tool; this is simply a front-end interface that calls or runs back-end database queries that are formatted for easy application usage. For example, a banking software application may contain specifically defined reports on all customers with large deposits or reports on monthly loan summaries for all customers.

To extract data, a query must be run with various tools that call at least one query language. Structured Query Language (SQL) is the most popular and well-known query language. Other query languages include:

* Hyper Text Structured Query Language (HTSQL): This language translates hypertext transfer protocol (HTTP) queries to SQL.
* Poliqarp Query Language: This language searches annotated text.
* SPARQL (a recursive acronym for SPARQL Protocol and RDF Query Language): This language is for graphing applications.

Another standard reporting feature is the ability to create output parameters or restrictions. For example, when a user runs a monthly loans summary report, the user first enters the specific month or account type associated with the requested report. Specialized reporting tools not typically available via simple queries may be connected to a database to facilitate additional reporting capabilities. These often offer greater data insight and highlight trends and patterns, and are frequently labelled under the buzzword "business intelligence" (BI).

**DBMS 🡪 Unit -4**

**DBMS | File Organization**

 A database consists of a huge amount of data. The data is grouped within a table in RDBMS, and each table have related records. A user can see that the data is stored in form of tables, but in actual this huge amount of data is stored in physical memory in form of files.

**File –** A file is named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tables and optical disks.

**What is File Organization?**

File Organization refers to the logical relationships among various records that constitute the file, particularly with respect to the means of identification and access to any specific record. In simple terms, Storing the files in certain order is called file Organization. **File Structure** refers to the format of the label and data blocks and of any logical control record.

**Types of File Organizations –**

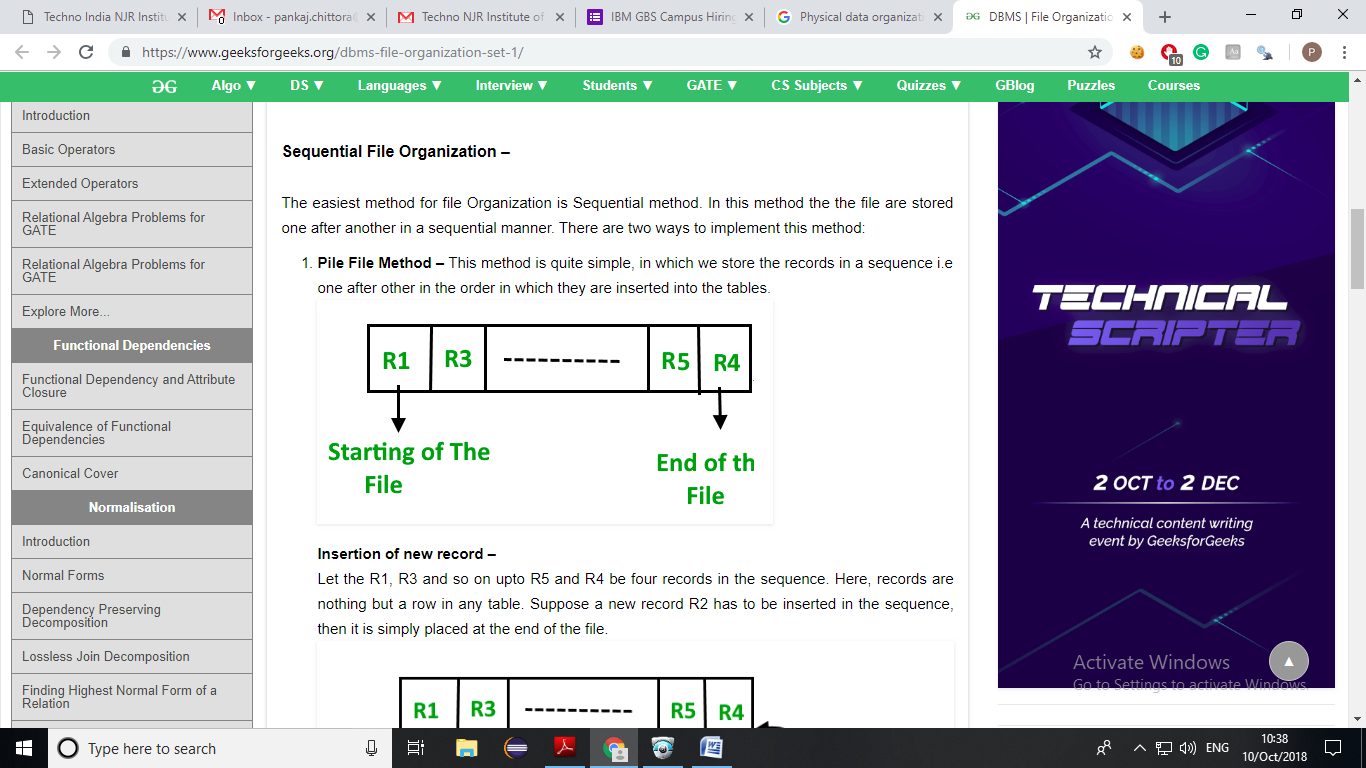
Various methods have been introduced to Organize files. These particular methods have advantages and disadvantages on the basis of access or selection . Thus it is all upon the programmer to decide the best suited file Organization method according to his requirements.  
Some types of File Organizations are :

* Sequential File Organization
* Heap File Organization
* Hash File Organization
* B+ Tree File Organization
* Clustered File Organization

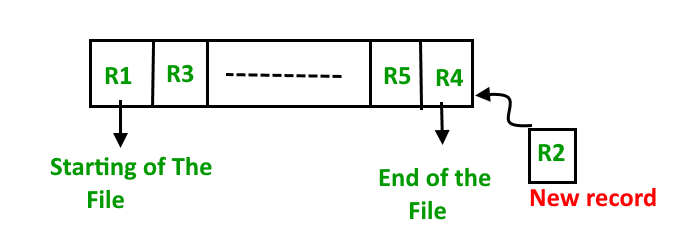
**Sequential File Organization –**

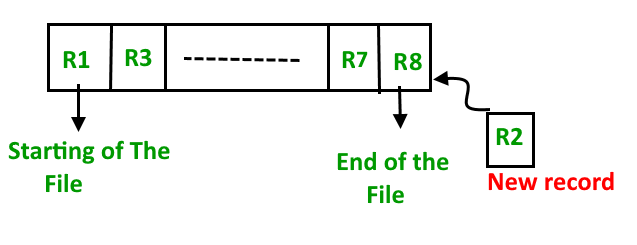
The easiest method for file Organization is Sequential method. In this method the the file are stored one after another in a sequential manner. There are two ways to implement this method:

1. Pile File Method – This method is quite simple, in which we store the records in a sequence i.e one after other in the order in which they are inserted into the tables.



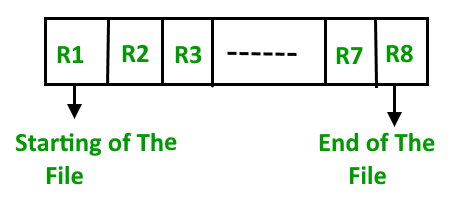
**Insertion of new record –**  
Let the R1, R3 and so on up to R5 and R4 be four records in the sequence. Here, records are nothing but a row in any table. Suppose a new record R2 has to be inserted in the sequence, and then it is simply placed at the end of the file.



1. **Sorted File Method –**In this method, As the name itself suggest whenever a new record has to be inserted, it is always inserted in a sorted (ascending or descending) manner. Sorting of records may be based on any primary key or any other key.  
   

**Insertion of new record –**

Let us assume that there is a pre-existing sorted sequence of four records R1, R3, and so on upto R7 and R8. Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file and then it will sort the sequence.



**Pros and Cons of Sequential File Organization –**  
**Pros –**

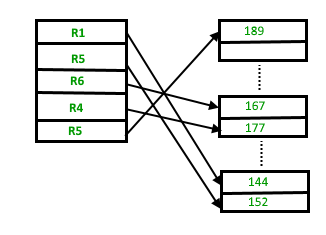
* Fast and efficient method for huge amount of data.
* Simple design.
* Files can be easily stored in magnetic tapes i.e cheaper storage mechanism.

**Cons –**

* Time wastage as we cannot jump on a particular record that is required, but we have to move in a sequential manner which takes our time.
* Sorted file method is inefficient as it takes time and space for sorting records.

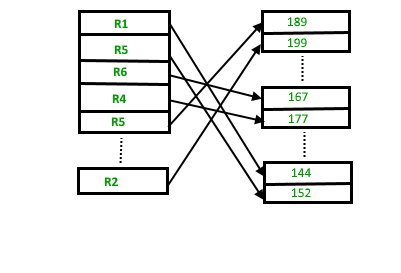
**Heap File Organization –**

Heap File Organization works with data blocks. In this method records are inserted at the end of the file, into the data blocks. No Sorting or Ordering is required in this method. If a data block is full, the new record is stored in some other block, Here the other data block need not be the very next data block, but it can be any block in the memory. It is the responsibility of DBMS to store and manage the new records.



**Insertion of new record –**

Suppose we have four records in the heap R1, R5, R6, R4 and R3 and suppose a new record R2 has to be inserted in the heap then, since the last data block i.e data block 3 is full it will be inserted in any of the database selected by the DBMS, lets say data block 1.



If we want to search, delete or update data in heap file Organization then we will traverse the data from the beginning of the file till we get the requested record. Thus if the database is very huge, searching, deleting or updating the record will take a lot of time.

**Pros and Cons of Heap File Organization –**

**Pros –**

* Fetching and retrieving records is faster than sequential record but only in case of small databases.
* When there is a huge number of data needs to be loaded into the database at a time, then this method of file Organization is best suited.

**Cons –**

* Problem of unused memory blocks.
* Inefficient for larger databases.

**Indexing**

We know that data is stored in the form of records. Every record has a key field, which helps it to be recognized uniquely.

Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to what we see in books.

Indexing is defined based on its indexing attributes. Indexing can be of the following types −

* **Primary Index** − Primary index is defined on an ordered data file. The data file is ordered on a **key field**. The key field is generally the primary key of the relation.
* **Secondary Index** − Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
* **Clustering Index** − Clustering index is defined on an ordered data file. The data file is ordered on a non-key field.

Ordered Indexing is of two types –

* Dense Index
* Sparse Index

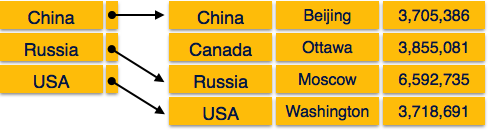
## Dense Index

In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index records contain search key value and a pointer to the actual record on the disk.



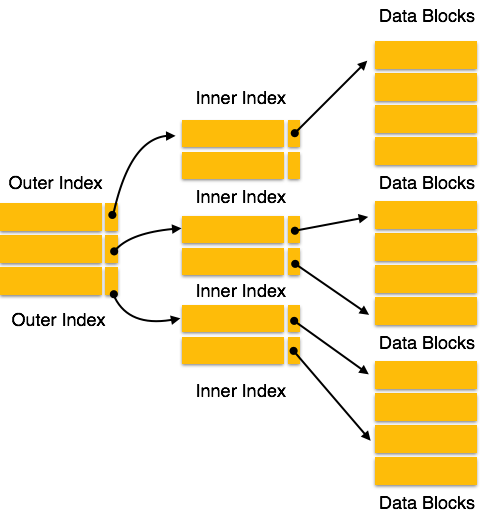
## Sparse Index

In sparse index, index records are not created for every search key. An index record here contains a search key and an actual pointer to the data on the disk. To search a record, we first proceed by index record and reach at the actual location of the data. If the data we are looking for is not where we directly reach by following the index, then the system starts sequential search until the desired data is found.



## Multilevel Index

Index records comprise search-key values and data pointers. Multilevel index is stored on the disk along with the actual database files. As the size of the database grows, so does the size of the indices. There is an immense need to keep the index records in the main memory so as to speed up the search operations. If single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.



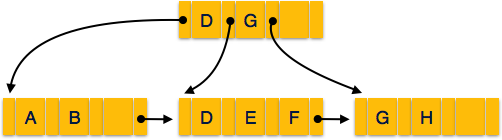
Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.

## B+ Tree

A B+ tree is a balanced binary search tree that follows a multi-level index format. The leaf nodes of a B+ tree denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height, thus balanced. Additionally, the leaf nodes are linked using a link list; therefore, a B+ tree can support random access as well as sequential access.

### Structure of B+ Tree

Every leaf node is at equal distance from the root node. A B+ tree is of the order **n** where **n** is fixed for every B+ tree.



**Internal nodes** −

* Internal (non-leaf) nodes contain at least ⌈n/2⌉ pointers, except the root node.
* At most, an internal node can contain **n** pointers.

**Leaf nodes** −

* Leaf nodes contain at least ⌈n/2⌉ record pointers and ⌈n/2⌉ key values.
* At most, a leaf node can contain **n** record pointers and **n** key values.
* Every leaf node contains one block pointer **P** to point to next leaf node and forms a linked list.

### B+ Tree Insertion

* B+ trees are filled from bottom and each entry is done at the leaf node.
* If a leaf node overflows −
  + Split node into two parts.
  + Partition at **i = ⌊(m+1)/2⌋.**
  + First **i** entries are stored in one node.
  + Rest of the entries (i+1 onwards) are moved to a new node.
  + **i*th*** key is duplicated at the parent of the leaf.
* If a non-leaf node overflows −
  + Split node into two parts.
  + Partition the node at **i = ⌈(m+1)/2⌉**.
  + Entries up to **i** are kept in one node.
  + Rest of the entries are moved to a new node.

### B+ Tree Deletion

* B+ tree entries are deleted at the leaf nodes.
* The target entry is searched and deleted.
  + If it is an internal node, delete and replace with the entry from the left position.
* After deletion, underflow is tested,
  + If underflow occurs, distribute the entries from the nodes left to it.
* If distribution is not possible from left, then
  + Distribute from the nodes right to it.
* If distribution is not possible from left or from right, then
  + Merge the node with left and right to it.

# Hashing

For a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.

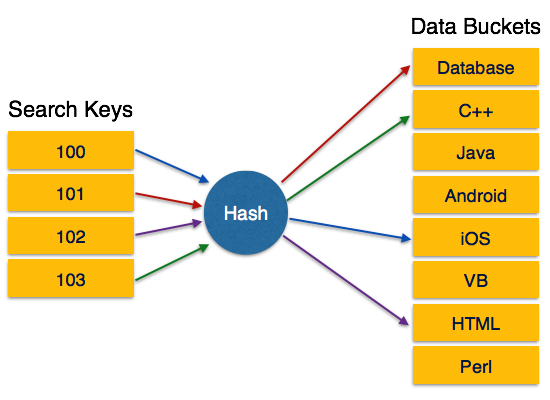
Hashing uses hash functions with search keys as parameters to generate the address of a data record.

## Hash Organization

* **Bucket** − A hash file stores data in **bucket** format. Bucket is considered a **unit of storage**. A bucket typically stores one complete disk block, which in turn can store one or more records.
* **Hash Function** − A hash function, **h,** is a mapping function that maps all the set of search-keys **K** to the address where actual records are placed. It is a function from search keys to bucket addresses.

## Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.



### Operation

* **Insertion** − When a record is required to be entered using static hash, the hash function **h** computes the bucket address for search key **K**, where the record will be stored.

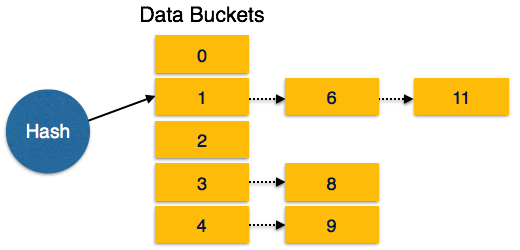
Bucket address = h (K)

* **Search** − When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.
* **Delete** − this is simply a search followed by a deletion operation.

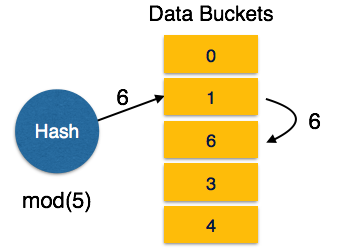
## Bucket Overflow

The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

* **Overflow Chaining** − When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.



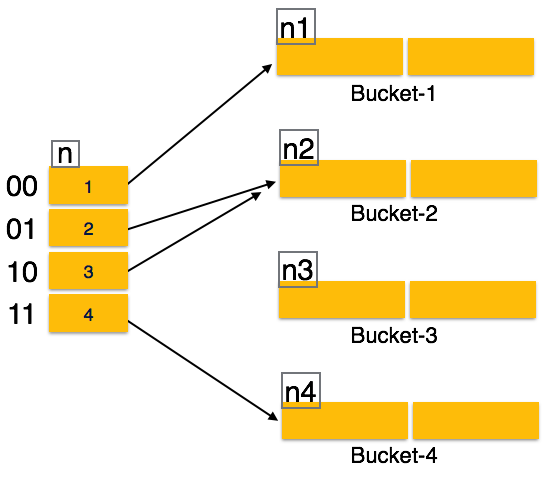
* **Linear Probing** − When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



## Dynamic Hashing

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as **extended hashing**.

Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



## Organization

The prefix of an entire hash value is taken as a hash index. Only a portion of the hash value is used for computing bucket addresses. Every hash index has a depth value to signify how many bits are used for computing a hash function. These bits can address 2n buckets. When all these bits are consumed − that is, when all the buckets are full − then the depth value is increased linearly and twice the buckets are allocated.

## Operation

* **Querying** − Look at the depth value of the hash index and use those bits to compute the bucket address.
* **Update** − Perform a query as above and update the data.
* **Deletion** − Perform a query to locate the desired data and delete the same.
* **Insertion** − Compute the address of the bucket
  + If the bucket is already full.
    - Add more buckets.
    - Add additional bits to the hash value.
    - Re-compute the hash function.
  + Else
    - Add data to the bucket,
  + If all the buckets are full, perform the remedies of static hashing.

Hashing is not favourable when the data is organized in some ordering and the queries require a range of data. When data is discrete and random, hash performs the best.

Hashing algorithms have high complexity than indexing. All hash operations are done in constant time.

**Multilist File Organisation**

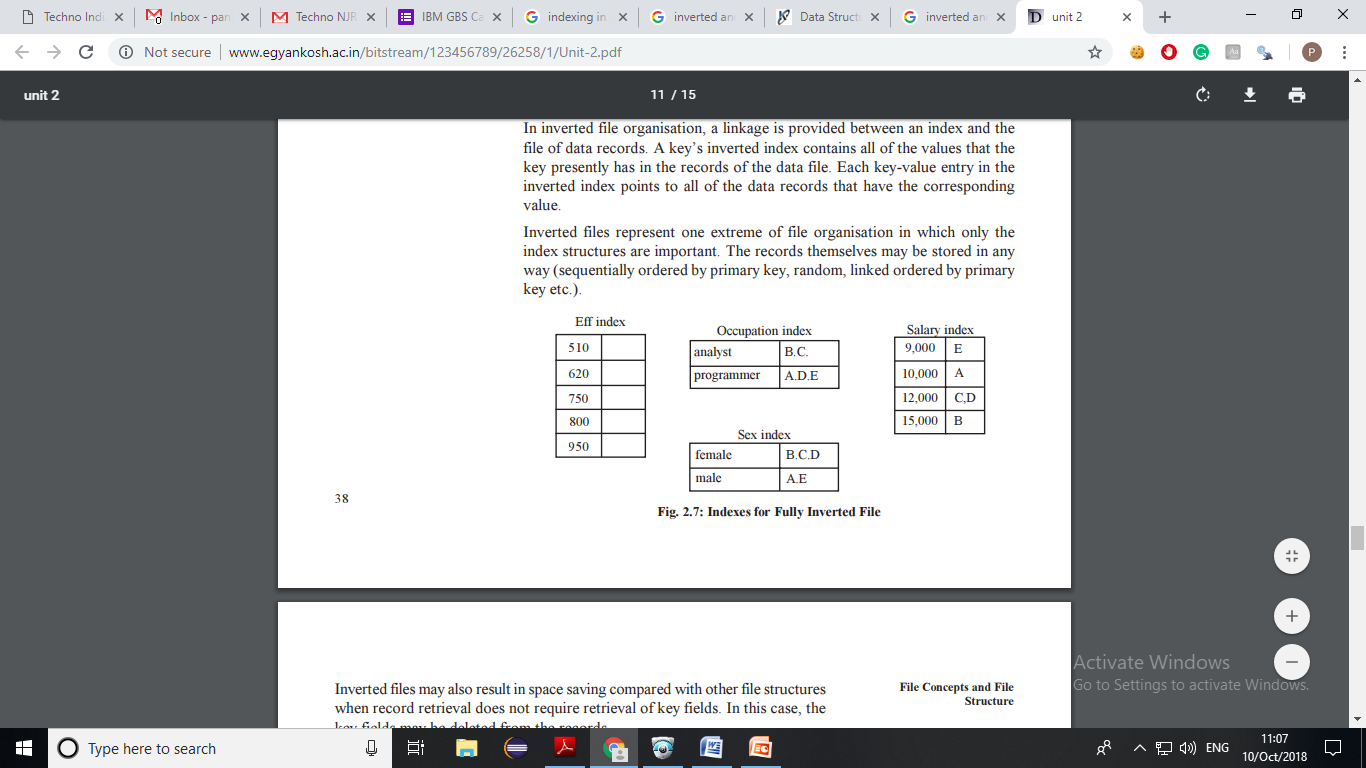
The basic approach to providing the linkage between an index and the file of data records is called multilist organisation. A multilist file maintains an index for each secondary key. The index for secondary key contains, instead of a list of primary keys related to that secondary key, only one primary key value related to that secondary key. That record will be linked to other records containing the same secondary key in the data file. The multi-list organisation differs from inverted file in that while the entry in the inverted file index for a key value has a pointer to each data record with that key value, the entry in the multi-list index for a key value has just one pointer to the first data record with that key value.

Linking records together in order of increasing primary key value facilitates easy insertion and deletion once the place at which the insertion or deletion to be made is known. Searching for a record with a given primary key value is difficult when no index is available, since the only search possible is a sequential search. To facilitate searching on the primary key as well as on secondary keys, it is customary to maintain several indexes, one for each key. Using an index in this way reduces the length of the lists and thus the search time. This idea is very easily generalised to allow for easy secondary key retrieval. We just set up indexes for each key and allow records to be in more than one list. This leads to the multilist structure for file representation.

**Inverted File Organisation**

In inverted file organisation, a linkage is provided between an index and the file of data records. A key’s inverted index contains all of the values that the key presently has in the records of the data file. Each key-value entry in the inverted index points to all of the data records that have the corresponding value.

Inverted files represent one extreme of file organisation in which only the index structures are important. The records themselves may be stored in any way (sequentially ordered by primary key, random, linked ordered by primary key etc.).



Inverted files may also result in space saving compared with other file structures when record retrieval does not require retrieval of key fields. In this case, the key fields may be deleted from the records.

Both inverted files and multilist files have:

* An index for each secondary key.
* An index entry for each distinct value of the secondary key.
* The index may be tabular or tree-structured.
* The entries in an index may or may not be sorted.
* The pointers to data records may be direct or indirect.

The indexes differ in that

* An entry in an inverted index has a pointer to each data record with that value.
* An entry in a multilist index has a pointer to the first data record with that value.

Thus an inverted index may have variable-length entries whereas a multilist index has fixed-length entries.

Some of the implications of these differences are the following:

* Index management is easier in the multilist approach because entries are fixed in length.
* The inverted file approach tends to exhibit better inquiry performance. Many types of queries can be answered by accessing inversion indexes without necessitating access to data records, thereby reducing I/O-access requirements.
* Inversion of a file can be transparent to a programmer who accesses that file but does not use the inversion indexes, while a multilist structure affects the file’s record layout. The multilist pointers can be made transparent to a programmer if the data manager does not make them available for programmer use and stores them at the end of each record.

**DBMS 🡪 UNIT - 5**

**CONCURRENCY CONTROL**

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
* Timestamp-based protocols.

**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 Locks:** 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.

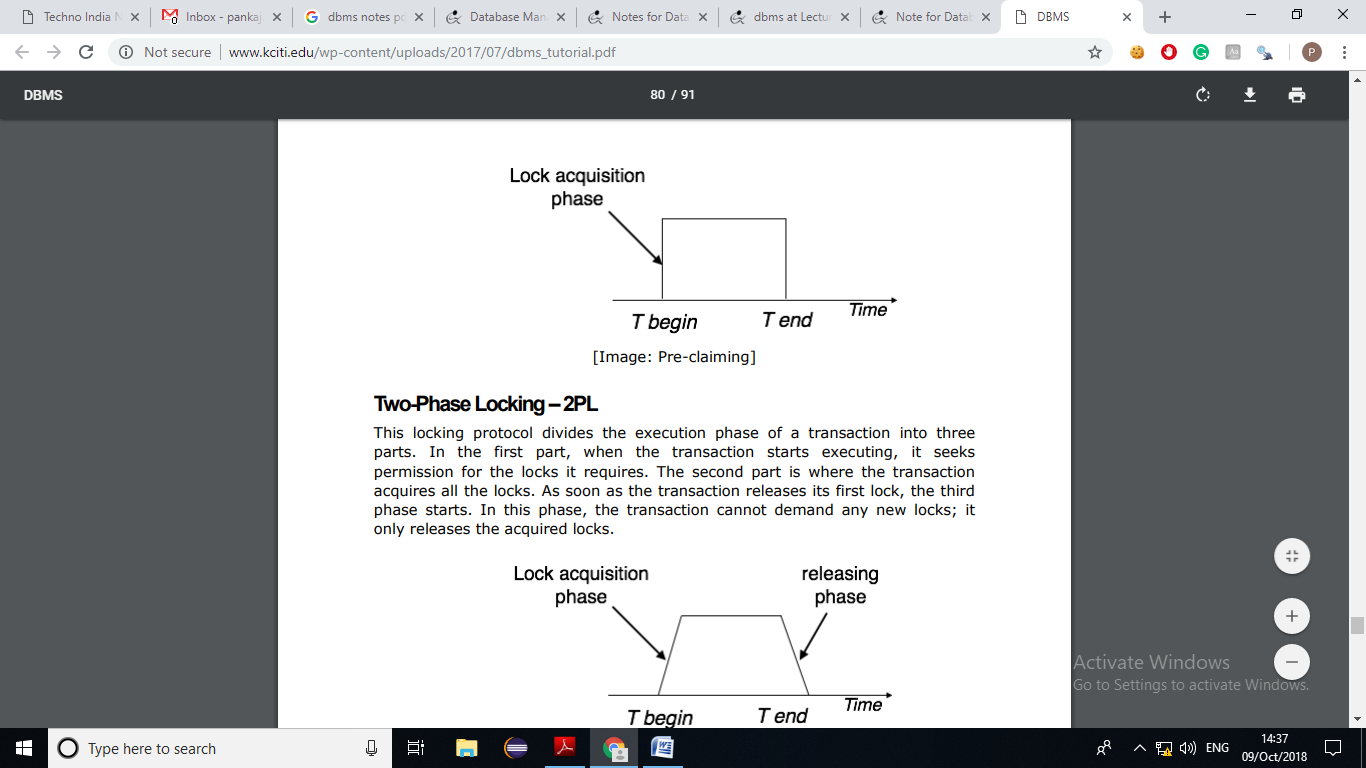
**There are four types of lock protocols available:**

**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.

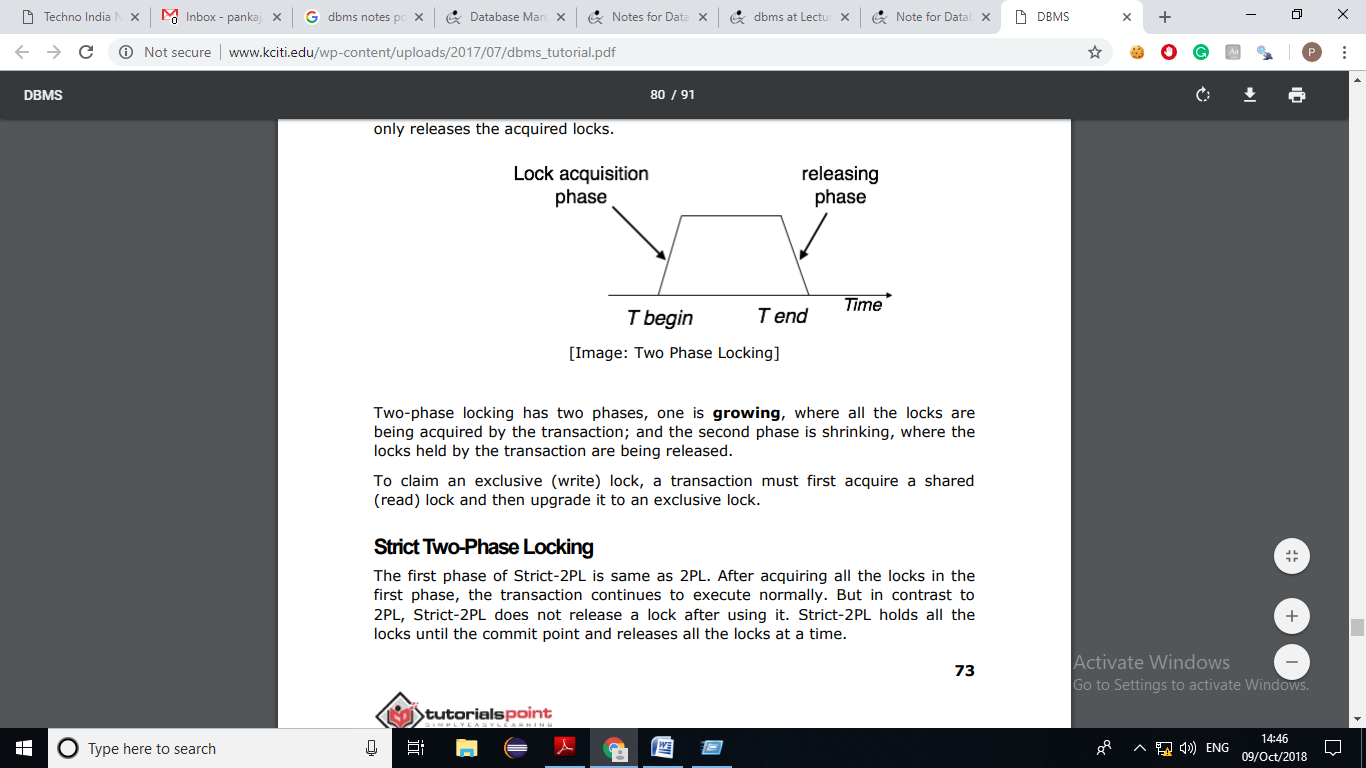
**Pre-claiming Lock Protocol**

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.

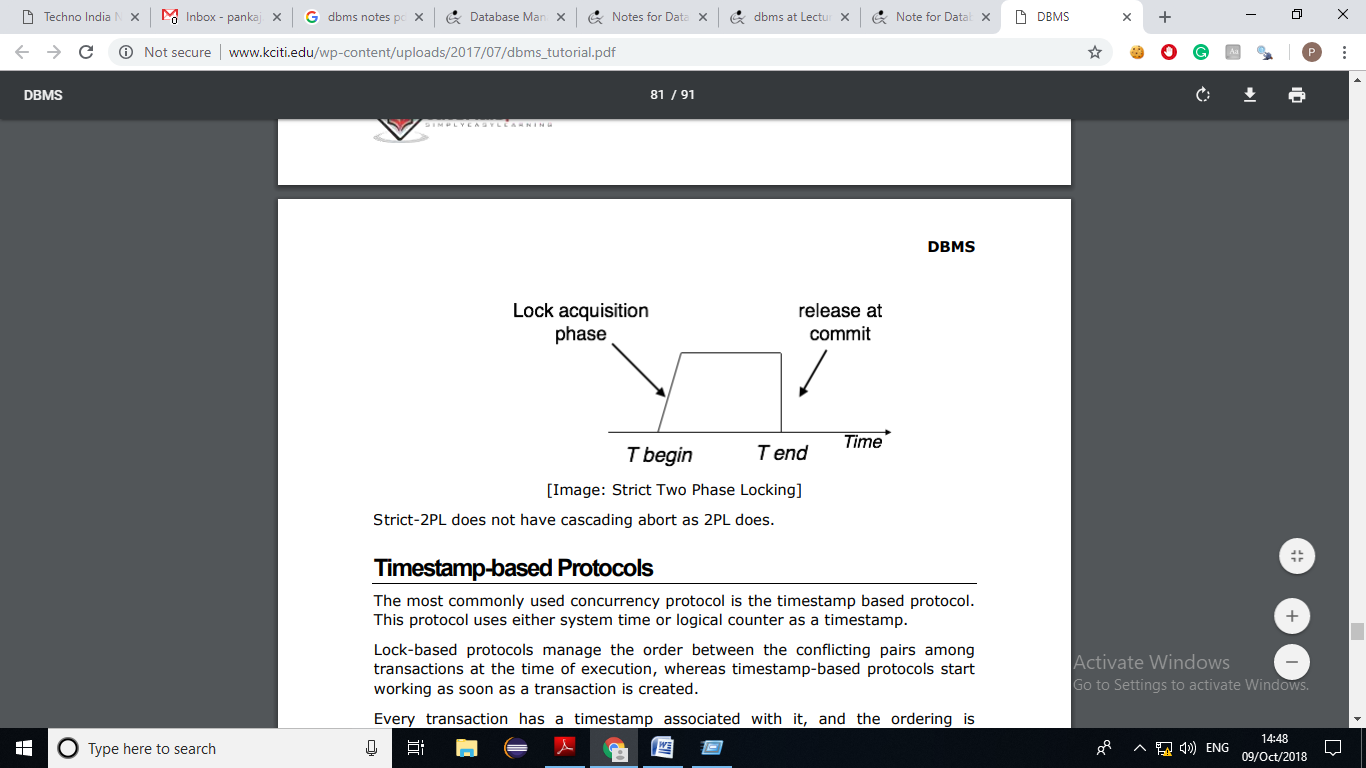
****

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.

Strict-2PL does not have cascading abort as 2PL does.



**Deadlock in DBMS**

In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.

For example, assume a set of transactions {T0, T1, T2, ...,Tn}. T0 needs a resource X to complete its task. Resource X is held by T1, and T1 is waiting for a resource Y, which is held by T2. T2 is waiting for resource Z, which is held by T0. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

Deadlock Prevention

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

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(Ti) < TS(Tj) − that is Ti, which is requesting a conflicting lock, is older than Tj − then Ti is allowed to wait until the data-item is available.
* If TS(Ti) > TS(tj) − that is Ti is younger than Tj − then Ti dies. Ti 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(Ti) < TS(Tj), then Ti forces Tj to be rolled back − that is Ti wounds Tj. Tj is restarted later with a random delay but with the same timestamp.
* If TS(Ti) > TS(Tj), then Ti 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.

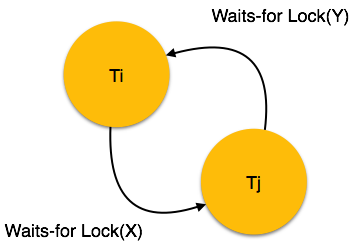
Deadlock Avoidance

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 Ti requests for a lock on an item, say X, which is held by some other transaction Tj, a directed edge is created from Ti to Tj. If Tj releases item X, the edge between them is dropped and Ti 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**.

**Log based recovery**

Atomicity property of DBMS states that either all the operations of transactions must be performed or none. The modifications done by an aborted transaction should not be visible to database and the modifications done by committed transaction should be visible.

To achieve our goal of atomicity, user must first output to stable storage information describing the modifications, without modifying the database itself. This information can help us ensure that all modifications performed by committed transactions are reflected in the database. This information can also help us ensure that no modifications made by an aborted transaction persist in the database.

**Log and log records –**

The log is a sequence of log records, recording all the update activities in the database. In a stable storage, logs for each transaction are maintained. Any operation which is performed on the database is recorded is on the log. Prior to performing any modification to database, an update log record is created to reflect that modification.

An update log record represented as: <Ti, Xj, V1, V2> has these fields:

1. Transaction identifier: Unique Identifier of the transaction that performed the write operation.
2. Data item: Unique identifier of the data item written.
3. Old value: Value of data item prior to write.
4. New value: Value of data item after write operation.

**Other type of log records are:**

1. <Ti start>: It contains information about when a transaction Ti starts.
2. <Ti commit>: It contains information about when a transaction Ti commits.
3. <Ti abort>: It contains information about when a transaction Ti aborts.

**Undo and Redo Operations –**

Because all database modifications must be preceded by creation of log record, the system has available both the old value prior to modification of data item and new value that is to be written for data item. This allows system to perform redo and undo operations as appropriate:

Undo: using a log record sets the data item specified in log record to old value.

Redo: using a log record sets the data item specified in log record to new value.

The database can be modified using two approaches –

1. Deferred Modification Technique: If the transaction does not modify the database until it has partially committed, it is said to use deferred modification technique.
2. Immediate Modification Technique: If database modification occur while transaction is still active, it is said to use immediate modification technique.

**Recovery using Log records –**

After a system crash has occurred, the system consults the log to determine which transactions need to be redone and which need to be undone.

1. Transaction Ti needs to be undone if the log contains the record <Ti start> but does not contain either the record <Ti commit> or the record <Ti abort>.
2. Transaction Ti needs to be redone if log contains record <Ti start> and either the record <Ti commit> or the record <Ti abort>.

**Use of Checkpoints –**

When a system crash occurs, user must consult the log. In principle, that need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time-consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will cause recovery to take longer.

To reduce these types of overhead, user introduce checkpoints. A log record of the form <checkpoint L> is used to represent a checkpoint in log where L is a list of transactions active at the time of the checkpoint. When a checkpoint log record is added to log all the transactions that have committed before this checkpoint have <Ti commit> log record before the checkpoint record. Any database modifications made by Ti are written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on Ti.

After a system crash has occurred, the system examines the log to find the last <checkpoint L> record. The redo or undo operations need to be applied only to transactions in L, and to all transactions that started execution after the record was written to the log. Let us denote this set of transactions as T. Same rules of undo and redo are applicable on T as mentioned in Recovery using Log records part.

Note that user need to only examine the part of the log starting with the last checkpoint log record to find the set of transactions T, and to find out whether a commit or abort record occurs in the log for each transaction in T. For example, consider the set of transactions {T0, T1, . . ., T100}. Suppose that the most recent checkpoint took place during the execution of transaction T67 and T69, while T68 and all transactions with subscripts lower than 67 completed before the checkpoint. Thus, only transactions T67, T69, . . ., T100 need to be considered during the recovery scheme. Each of them needs to be redone if it has completed (that is, either committed or aborted); otherwise, it was incomplete, and needs to be undone.

Shadow Paging:

• Shadow paging is a technique for providing atomicity and durability in database systems.

• Shadow paging is a copy-on-write technique for avoiding in-place updates of pages. Instead, when a page is to be modified, a shadow page is allocated.

• Since the shadow page has no references (from other pages on disk), it can be modified liberally, without concern for consistency constraints, etc. When the page is ready to become durable, all pages that referred to the original are updated to refer to the new replacement page instead. Because the page is "activated" only when it is ready, it is atomic.

• This increases performance significantly by avoiding many writes on hotspots high up in the referential hierarchy (e.g.: a file system superblock) at the cost of high commit latency.

Shadow paging considers:

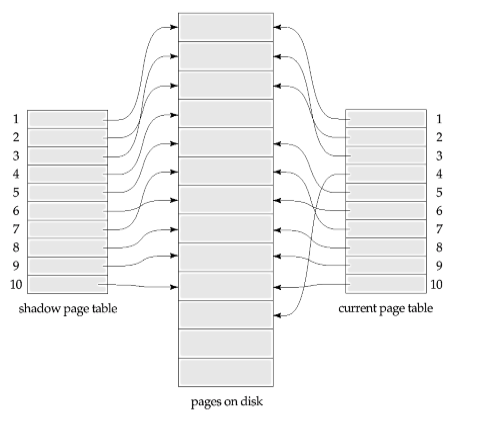
1. The database is partitioned into fixed-length blocks referred to as PAGES.
2. Page table has n entries – one for each database page.
3. Each contain pointer to a page on disk (1 to 1st page on database and so on…).

The idea is to maintain 2 pages tables during the life of transaction.

1. The current page table
2. The shadow page table

When transaction starts, both page tables are identical

1. The shadow page table is never changed over the duration of the transaction.
2. The current page table may be changed when a transaction performs a write operation.
3. All input and output operations use the current page table to locate database pages on disk.



**Advantages:**

• No Overhead for writing log records.

• No Undo / No Redo algorithm.

• Recovery is faster.

**Disadvantages:**

• Data gets fragmented or scattered.

• After every transaction completion database pages containing old version of modified data need to be garbage collected.

• Hard to extend algorithm to allow transaction to run concurrently.

View Serializability

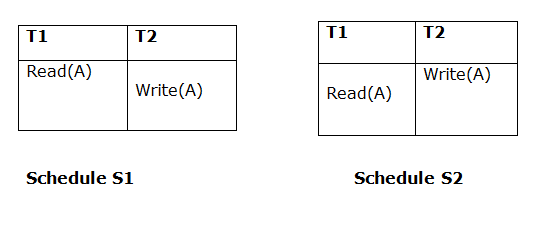
* A schedule will view serializable if it is view equivalent to a serial schedule.
* If a schedule is conflict serializable, then it will be view serializable.
* The view serializable which does not conflict serializable contains blind writes.

View Equivalent

Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

1. Initial Read

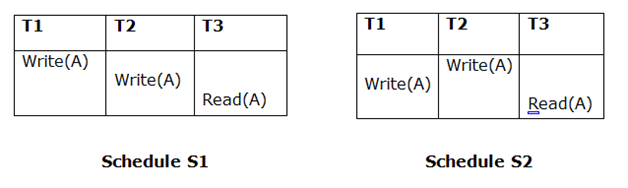
An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.



Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

2. Updated Read

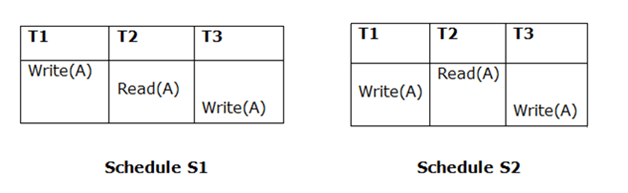
In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.



Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

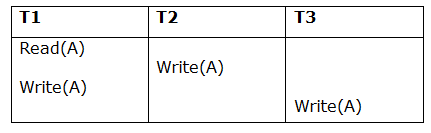
3. Final Write

A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.



Above two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

**Example:**

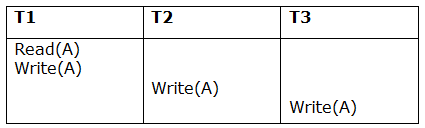


**Schedule S**

With 3 transactions, the total number of possible schedule

1. = 3! = 6
2. S1 = <T1 T2 T3>
3. S2 = <T1 T3 T2>
4. S3 = <T2 T3 T1>
5. S4 = <T2 T1 T3>
6. S5 = <T3 T1 T2>
7. S6 = <T3 T2 T1>

**Taking first schedule S1:**



**Schedule S1**

**Step 1:** final updation on data items

In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.

**Step 2:** Initial Read

The initial read operation in S is done by T1 and in S1, it is also done by T1.

**Step 3:** Final Write

The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.

**Hence, view equivalent serial schedule is:**

1. T1    →      T2    →    T3