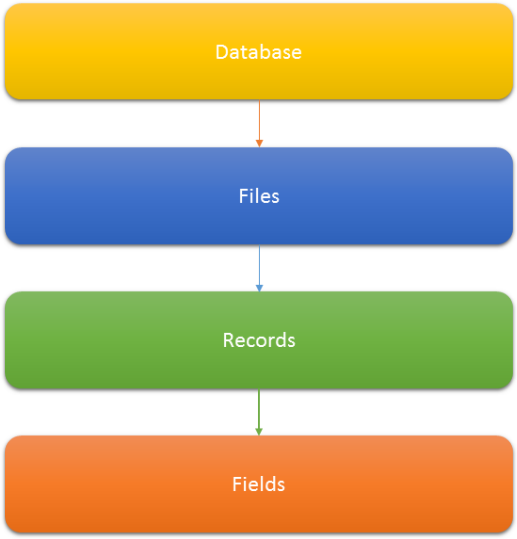
File Organization

The database is stored as a collection of files. Each file is a sequence of records. A record is a sequence of fields. Data is usually stored in the form of records. Records usually describe entities and their attributes



.

If every record in the file has exactly the same size (in bytes), the file is said to be made up of fixed-length records.

If different records in the file have different sizes, the file is said to be made up of variable length records

Blocking Factor: It is the number of records per block.

And there are two types of strategies to store records in the blocks.

- **Spanned Strategy**: We can store the remaining part of the records into some other block.

- Advantages: We do not have wastage of memory
- Disadvantage: Block Access is increased.

UnSpanned Strategy: No record can be stored in more than 1 block.

- Advantage: Block access is reduced.
- Disadvantage: Wastage of memory. And suitable only for fixed length records.

Organization of Records in a File

Ordered File Organization

- All records of the files are ordered on some key value. That's the name is ordered.
- Advantage is that we can apply binary sort on the data so the searching will become efficient

Un Ordered File Organization

So as the name suggests that wherever we have the place we insert the records so the insertion time will reduce but the searching time will increase

We will apply linear search

Advantage: Insertion is efficient.

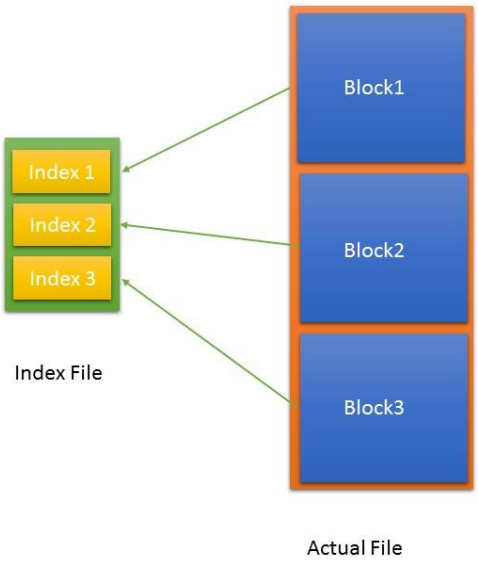
Disadvantage: searching is inefficient.

Indexing

In this process we make a file which will contain pointers which will point to actual data records.
It has two columns which has the following structure

	Key_name	Pointer

In the following diagram we can see that how the records are arranges



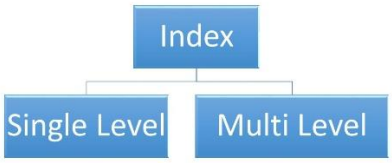
Index File size will be less as the index file will have only two fields.

- Key Name
- Pointer to the actual records

Whereas the actual file could have various fields and the size will be more as compared to the index file.

Types of index file

There are two types of classification of index files.



One is based on levels.

Single Level: We have following subtypes in this one.

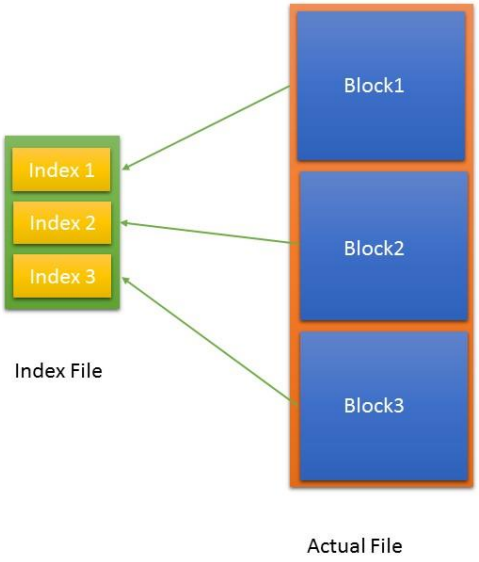
- Primary Index: Primary Key and the data will be ordered.
- Clustered Index: It will be applied on non-key and Ordered
- Secondary Index: It will be applied on non-key and unordered

Classification of Indexing

In the previous page we have seen one classification. Here we are going to discuss one more classification.

- Dense Index: For every records we are having one entry in the index file.
- Sparse Index: Only one record for each block we will have it in the index file.

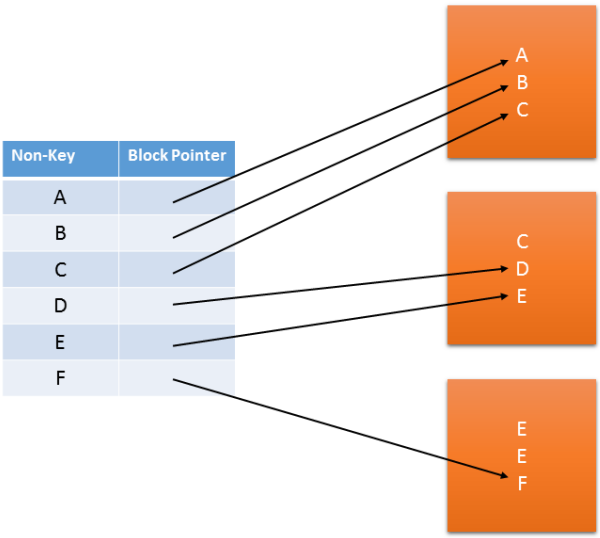
Primary Index (Primary Key + Ordered Data) A primary index is an ordered file whose records are fixed length size with 2 fields. First field is same as primary key and the second field is pointer to the data block.



- Here index entry is created for first record of each block, called 'block anchor'.
- Number of index entries thus are equal to number of blocks

Clustered Indexing

Clustered Index is an ordered file with two fields, non-key and block pointer.
Clustering Index is created on data file whose file records are physically ordered on a non-key field which does not have distinct value for each record, that field is called clustering field.



Explanation

- Index Entry is created for each distinct value of clustering field.
- Block pointer points to first block where key is available
- Type of index is dense and sparse both.

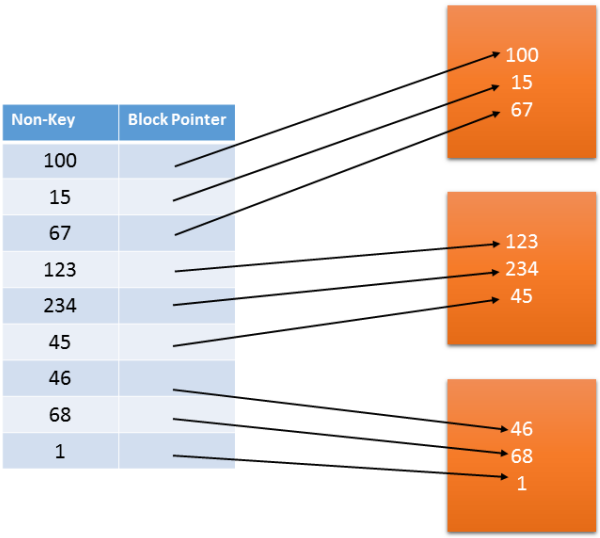
Average Number of block accesses required >= logB + 1

Secondary Index

This a type of dense Index.

- Secondary Index provides a secondary means of accessing a file for which some primary access already exists.
- Secondary Index may be a key or a non-key
- Index entry is created for each record in data file
- Type of secondary index is Dense

Index fields are key (unordered) and block pointer



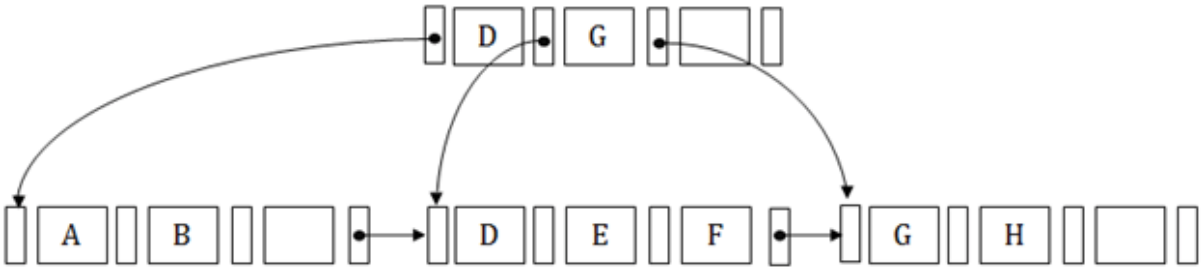
There are two multilevel indexes: B TREES AND B+ TREES

B+ Tree

- The B+ tree is a balanced binary search tree. It follows a multi-level index format.
- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, 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

- In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.
- It contains an internal node and leaf node.



Internal node

- An internal node of the B+ tree can contain at least n/2 record pointers except the root node.
- At most, an internal node of the tree contains n pointers.

Leaf node

- The leaf node of the B+ tree can contain at least n/2 record pointers and n/2 key values.
- At most, a leaf node contains n record pointer and n key values.
- Every leaf node of the B+ tree contains one block pointer P to point to next leaf node.

B+ Tree Construction

The method for constructing B+tree is similar to the building of B tree but the only difference here is that, the parent nodes also appear in the leaf nodes.

**********PLS REFER MY CLASS NOTES**********

UNIT-V

Recovery and Atomicity

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 −

- **Log Based Recovery**: Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- **Shadow paging**: Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

LOG BASED RECOVERY

It is used to recover from a system crash or transaction failure using a log.

It is the most commonly used structure for recording database modifications.

- The log is a small file that contains a sequence of records where the records depict the operations performed by Transaction.
- Log of each transaction is maintained in some stable storage (RAM/ shared memory) So that if any failure occurs, then it can be recovered from there.
- Before any operation is performed on the database, it will be recorded in the log.
- The process of storing the logs should be done before the actual transaction is applied in the database.

A log contains:

1. Transaction Identifier
2. Data item identifier
3. Old value
4. New value

   Example: <T_i, X_j,V_1,V_2>

   T_i: Transaction identifier used to uniquely identify a Transaction.

   X_j: Data item on which the operations are performed.

   V_1: Old value of data item

   V_2: New value of data item

Let's assume there is a transaction to modify the City of a student. The following logs are written for this transaction.

- When the transaction is initiated, then it writes 'start' log.

  <Tn, Start>

- When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.

  <Tn, City, 'Noida', 'Bangalore' >

- When the transaction is finished, then it writes another log to indicate the end of the transaction.

  <Tn, Commit>

There are two approaches to modify the database using log based recovery:

1. **Deferred database modification**
2. **Immediate database modification.**

1. Deferred database modification:

In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

A Deffered database modification is also called No Undo/ Redo method.

Undo: Restores the old values

Redo: Updates the new values

In this log contains a new value of the data item.

TRANSACTION	LOG	DATABASE
T1		A=3000, B=2000
R(A)	<T1,START>	
A=A-500		
W(A)	<T1,A,2500>	
R(B)		
B=B+500		
W(B)	<T1,B,2500>	
COMMIT	<T1,COMMIT>	A=2500,B=2500
SYSTEM CRASH		
		REDO
		A=2500,B=2500

Example: A=3000, B=2000

The deferred modification technique occurs if the transaction does not modify the database until it has committed.

TRANSACTION	LOG	DATABASE
T1		A=3000, B=2000
R(A)	<T1,START>	
A=A-500		
W(A)	<T1,A,2500>	
R(B)		
B=B+500		
W(B)	<T1,B,2500>	
SYSTEM CRASH		
COMMIT		A=3000, B =2000

2. Immediate database modification:

- The Immediate modification technique occurs if database modification occurs while the transaction is still active.

- In this technique, the database is modified immediately after every operation. It follows an actual database modification.

TRANSACTION	LOG	DATABASE
T1		A=3000, B=2000
R(A)	<T1,START>	
A=A-500		
W(A)	<T1,A,3000,2500>	A=2500, B=2000
R(B)		
B=B+500		
W(B)	<T1,B,2000,2500>	A=2500, B=2500
COMMIT	<T1, COMMIT>	A=2500,B=2500
SYSTEM CRASH		REDO
		A=2500,B=2500

When the system is crashed, then the system consults the log to find which transactions need to be undone and which need to be redone.

Immediate database modification is called Undo/ Redo strategy

UNDO & REDO:

1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.
2. If log contains record<T_n, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

Shadow paging in DBMS

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.

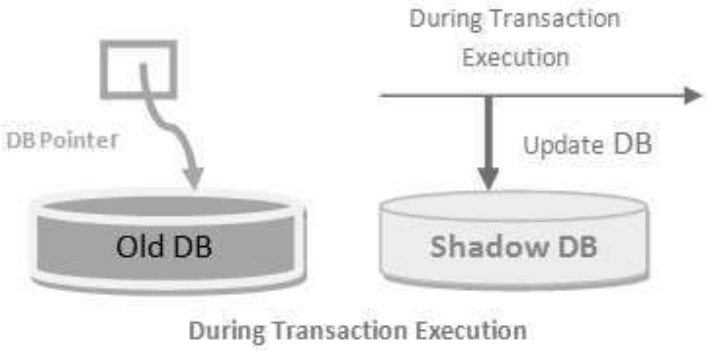
Database will be updated after all the transaction is complete.

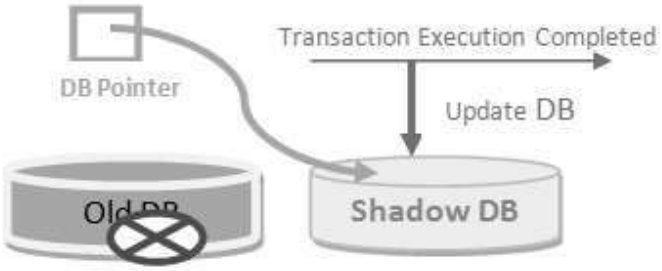
A database pointer will be always pointing to the consistent copy of the database, and copy of the database is used by transactions to update.

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.

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.

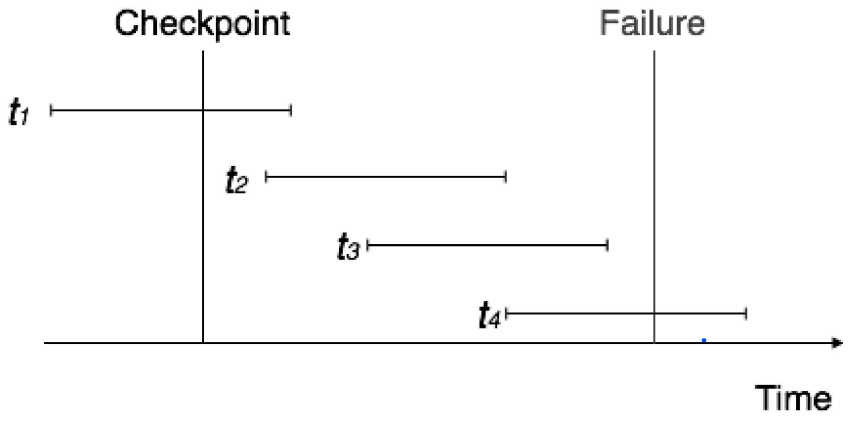
As time passes, the log file may grow too big to be handled at all.

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 log files from the end to start. It reads log files from T4 to T1.
- Recovery system maintains two lists, a redo-list, and an undo-list.
- The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
- **For example:** In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.
- The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.
- **For example:** Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.