Module 5

Physical Data Organization: index structures, primary, secondary and clustering indices, Single level and Multi-level indexing, B+Trees (basic structure only, algorithms not needed), Query Optimization:heuristics-based query optimization,

5.1 Physical Data Organization

There are two types of storage devices used with computers: a primary storage device,, and a secondary storage device

- Primary storage devices: Generally smaller in size, these are designed to hold data temporarily and are internal to the computer. They have the fastest data access speed, and include RAM and cache memory.
- Secondary storage devices: These usually have large storage capacity, and they store data permanently. They can be either internal or external to the computer, and they include the hard disk, optical disk drive and USB storage device.

Relative data and information is stored collectively in file formats. The **File** is a collection of records. A file can contain:

- **Fixed-length records** - all the records are exactly the same length
- **Variable-length records** - the length of each record varies

Using variable-length records might enable you to save disk space. When you use Fixed-length records, you need to make the record length equal to the length of the longest record.

A file is a sequence of records stored in binary format. A disk drive is formatted into several blocks that can store records. File records are mapped onto those disk blocks. **Blocking factor** is the number of records in a block at a particular time.

Blocking factor = floored((Block size)/(Record size))

Block size is larger than the record size.

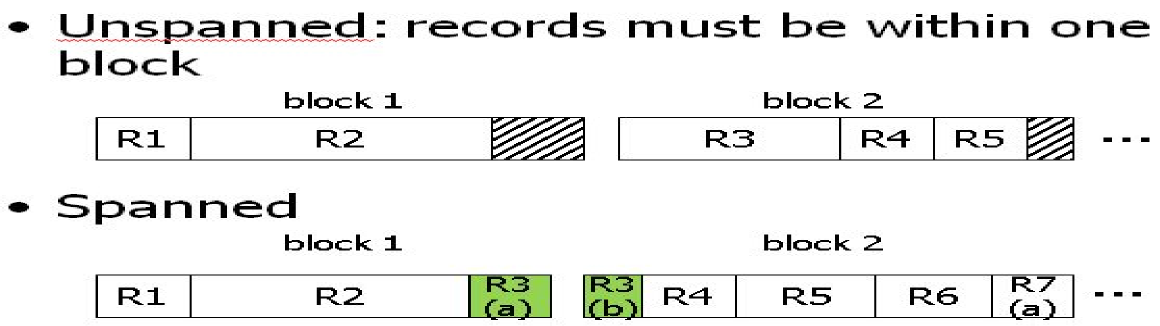
Assume We have 10 000 000 records. Each record is 80 bytes long and block size is 5000 0 then find number of blocks need to store all records,

Blocking factor= Floor((5000/80))
= Floor(62.5)
= 62 ie. 62 records

And we have 10000000 records, so we need `ceiled(10000000/62)=ceiled(161290.32)=161291` blocks to store all that data.

Spanned record and unspanned records

A part of record will get stored on one block and remaining on some other disk block. In such a case there will be a pointer at the end of the disk block that will point to the next block. Such a organization is called **Spanned**. If Records are not allowed to cross block boundaries we call them **unspanned**.



- Unspanned: records must be within one block
- Spanned

File Allocation Methods

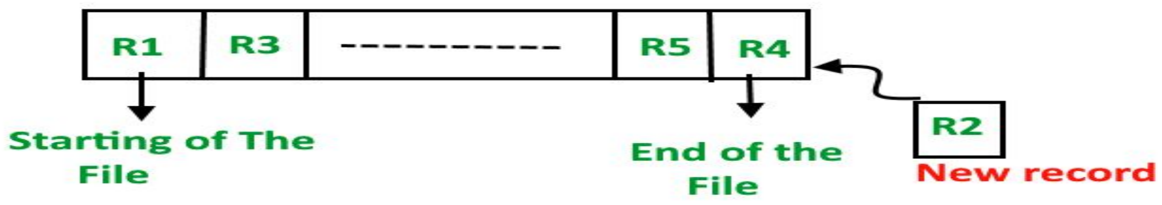
The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

- Contiguous Allocation: file occupies a contiguous set of blocks on the disk.This method suffers from both internal and external fragmentation.
- Linked Allocation: ach file is a linked list of disk blocks which **need not be** contiguous. The disk blocks can be scattered anywhere on the disk. Each block contains a pointer to the next block occupied by the file.
- Indexed Allocation: a special block known as the **Index block** contains the pointers to all the blocks occupied by a file. Each file has its own index block.

5.1.1 File Organization: The DB is stored as a collection of files. Each files is a sequence of records. A record is a sequence of field. If every record in a file has same size, then file is said to be fixed length record. If every record in a file has different size, then file is said to be variable length record. File Organization are based on sorted and unsorted records.

5.1.1.1 Files of Unordered Records (Heap Files)

In this simplest and most basic type of organization, records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. Such an organization is called a heap or pile file. There is no sorting or ordering of the records. Once the data block is full, the next record is stored in the new block. This new block need not be the very next block. This method can select any block in the memory to store the new records. When a record has to be retrieved from the database, in this method, we need to traverse from the beginning of the file till we get the requested record (linear search). To delete a record, a program must first find its block, copy the block into a buffer, delete the record from the buffer, and finally rewrite the block back to the disk. This leaves unused space in the disk block. Deleting a large number of records in this way results in wasted storage space. The deletion techniques require periodic reorganization of the file to reclaim the unused space of deleted records.



5.1.1.2 Files of Ordered Records (Sorted Files)

Sorting is based on a key field. Insertion and deletion operation is expensive. To insert and delete we have to find the exact block of records. First the record is placed in the overflow file which is unordered. Then at the time of reorganisation, records are sorted and copy to the master file. For selection, both linear search and binary search are used. Linear search in overflow file and binary search in master file.

5.1.2 Hashing

Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure. Data is stored at the data blocks whose address is generated by using hash function.

5.2 Index structures

An index is a data structure which is used to quickly locate and access the data in a database table.

Type of Index

- Single level index
- Multi level index

5.2.1 Single level index:

An index is a small table having only two columns. The first column contains search key of a table and the second column contains a set of pointers holding the address of the disk block where that particular key value can be found.

The values in the index are ordered, so we can do a binary search on the index. The index file is much smaller than the data file, so the binary search is much faster.

Search Key	Pointer

There are different types of indexes, these include:

- Primary Indexes
- Clustering index
- Secondary index

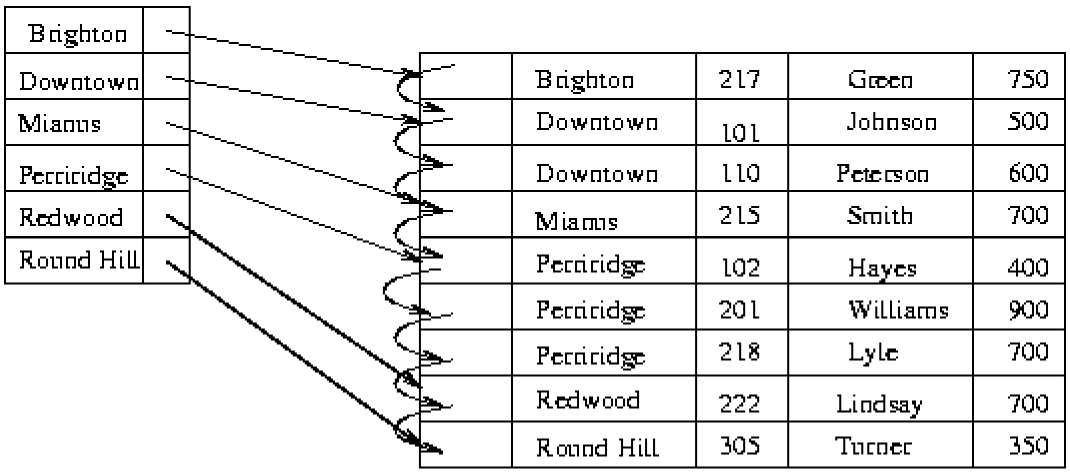
Primary Index: Primary index is defined on an ordered data file. The data file is ordered on a key field. The key field is generally the primary key of the relation. Primary index is created automatically when the table is created in the DB.

There are two types of primary index,

- 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.



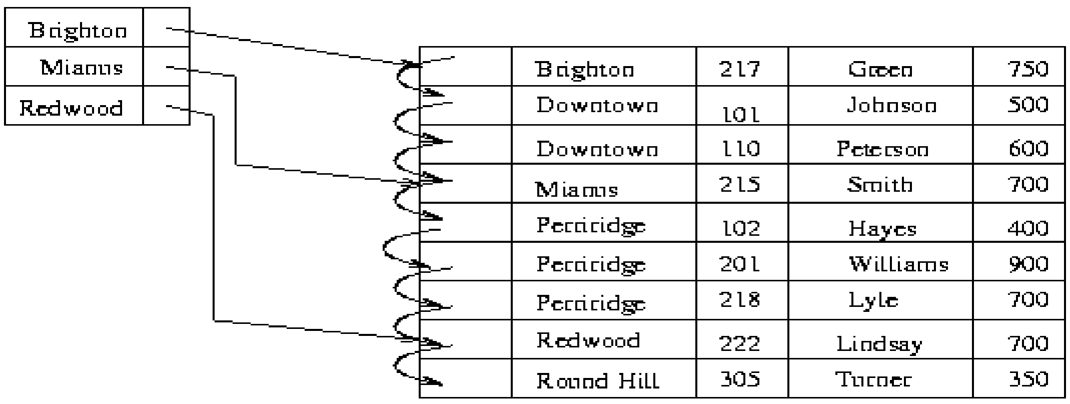
Brighton
Downtown
Mianus
Perriridge
Redwood
Round Hill

Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perriridge	102	Hayes	400
Perriridge	201	Williams	900
Perriridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

Sparse Index

Index records are created only for some of the search key.

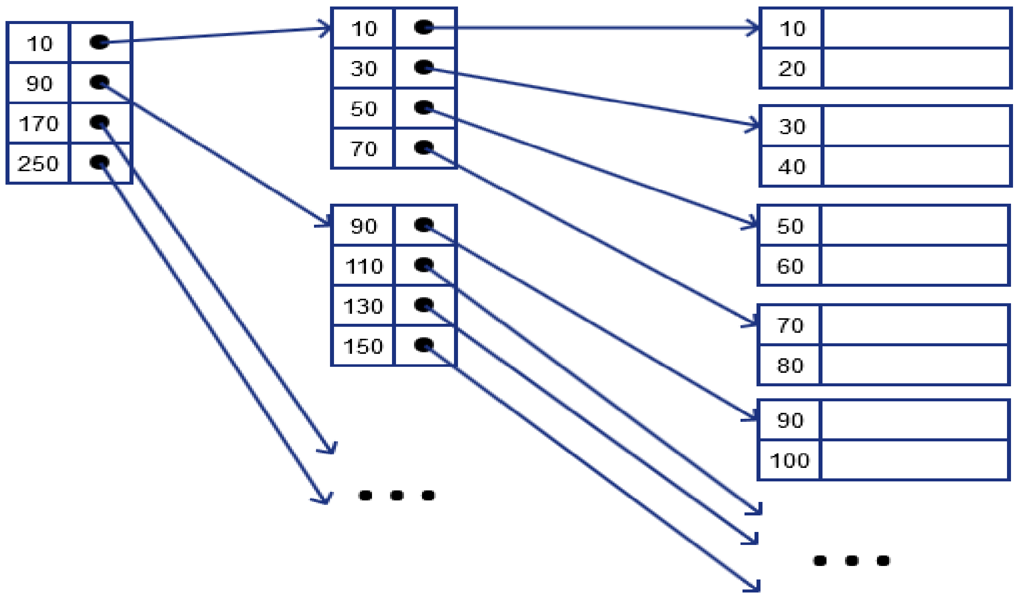
- To locate a record, we find the index record with search key value less than or equal to the search key value we are looking for.
- We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.



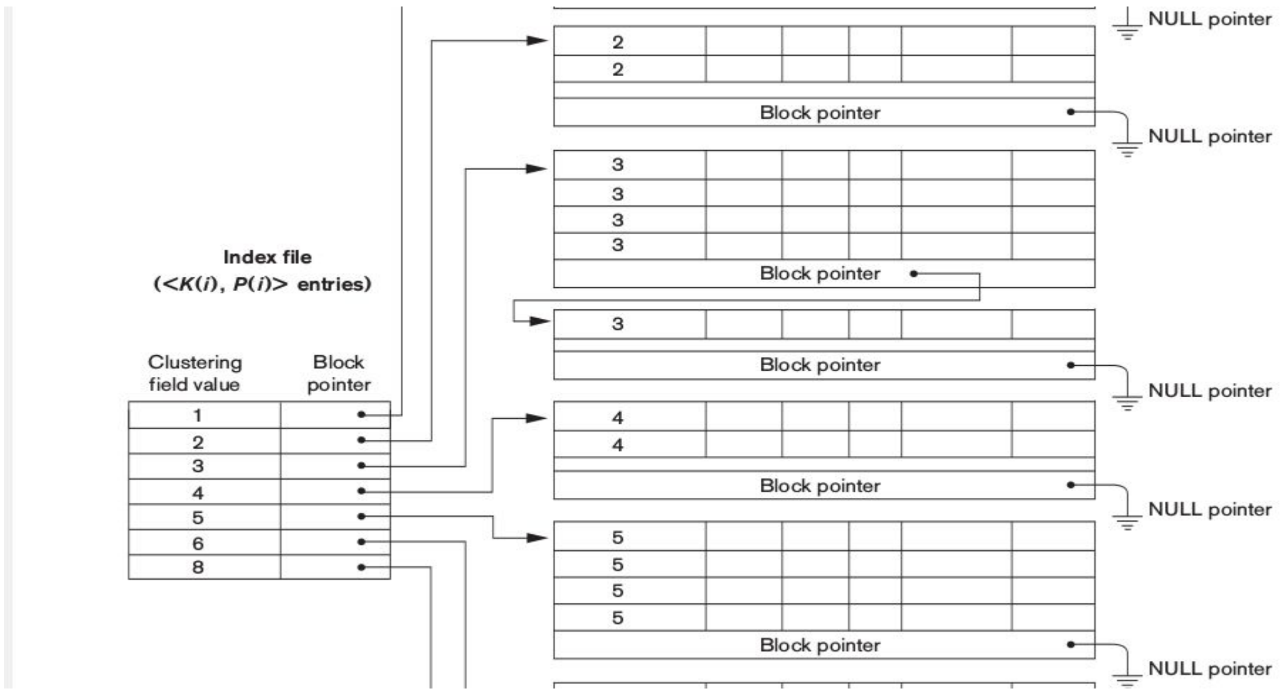
Brighton
Mianus
Redwood

Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perriridge	102	Hayes	400
Perriridge	201	Williams	900
Perriridge	218	Lyle	700
Redwood	222	Lindsay	700
Round Hill	305	Turner	350

Secondary Index: Index table is generally kept in primary memory and its entire contents are placed in secondary memory. Primary memory contain sparse index and secondary memory contain two indexes (1 sparse index, 1 actual table)



Clustered Index: Clustered indexes are efficient on columns that are searched for a range of values. Primary memory contain index of clustered group which point to actual data in secondary. Here 2 in primary memory pointed to the secondary memory. If cluster two is ended in same block then the block pointer set to be null. In the case of cluster 3, it spanned to the next block then the block pointer point to the next block.



MULTILEVEL INDEX

● If the primary index does not fit in the memory access become expensive.

● To reduce the number of disk accesses to index records, treat primary index kept on the disk as sequential file and construct a sparse index on it.

▪ Outer index – A sparse index of primary index.

▪ Inner index – The primary index file.

● If even outer index is too large to fit in main memory, yet another level of index can be created.

● Indices from all levels must be updated on insertion or deletion from the file.

● Insertion and deletion are complicated because all index files at different levels are ordered files.

● One solution to this problem is to keep some space reserved in each block for new entries.

● The outer index is the first level sparse index.

● The inner index is the second level sparse index.

● In order to search for an index, one can first apply binary search on the second level index to find the largest value which is less than or equal to the search key.

● The pointer corresponding to this value points to the block of the first level index that contains an entry for the search record.

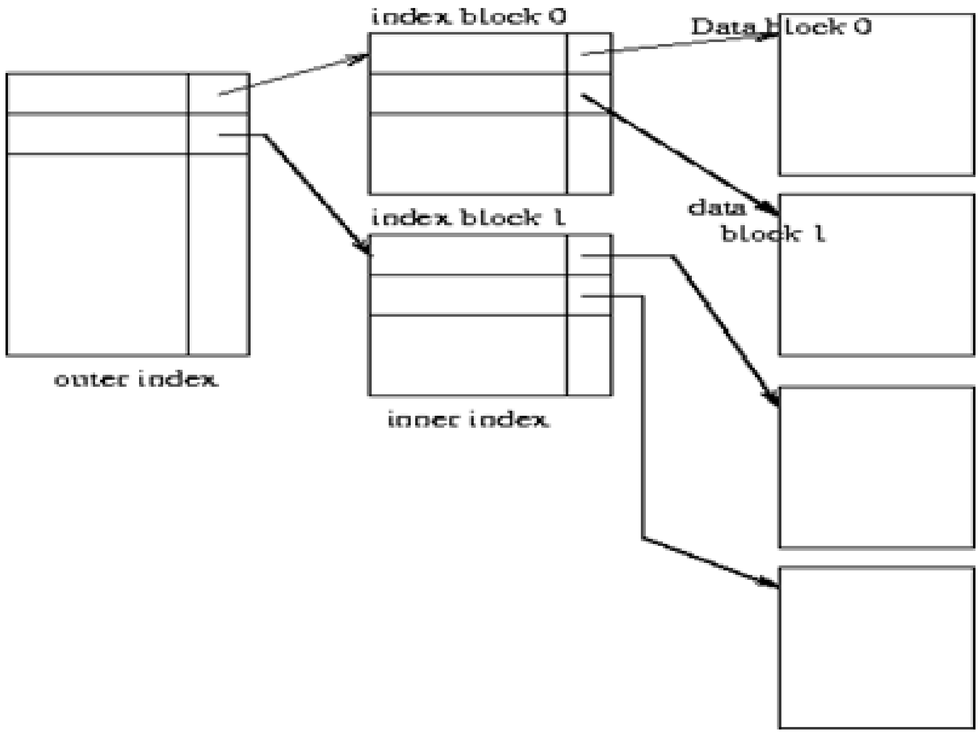
● This block level index is searched to locate the largest value which is less than or equal to the search key.

● The pointer corresponding to this block directs us to the block that contains the required content.

If the second level index is too small to fit in main memory at once, fewer blocks of index file needs to be accessed from the disk to locate a particular record.

However if the second level is too large to fit in the main memory at once, another level of index can be created.

This process of creating index on index can be repeated until the size of the index become small enough to fit in the main memory.This type of index with multiple levels of index is called “MULTILEVEL INDEX”.



5.3 B+Tree

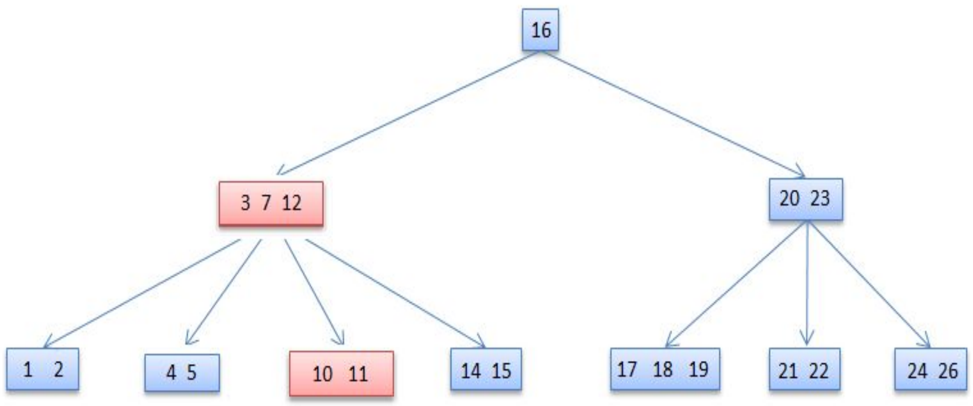
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations. In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values. The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient. B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory. The internal nodes of B+ tree are often called index nodes.

For example, B+Tree of Order 4 contains a maximum of 3 key values in a node, minimum key 1 and maximum of 4 children for a node.

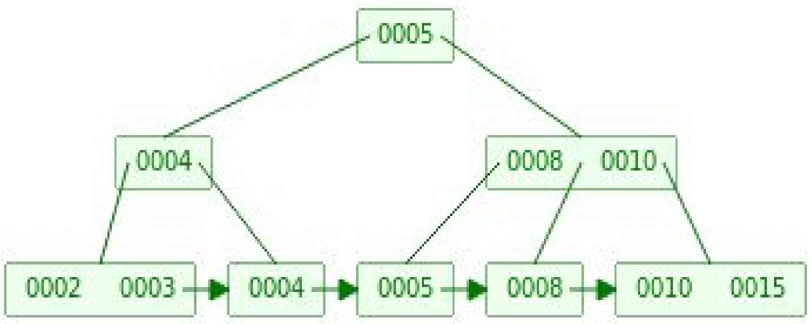
- **Property #1** - All **leaf nodes** must be **at same level**.
- **Property #2** - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys in the case of order.
- Incase of degree t minimum key is t-1 and maximum key is 2t-1
- **Property #3** - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
- **Property #4** - If the root node is a non leaf node, then it must have **atleast 2** children.

- **Property #5** - A non leaf node with **n-1** keys must have **n** number of children.
- **Property #6** - All the **key values in a node** must be in **Ascending Order**.

Btree



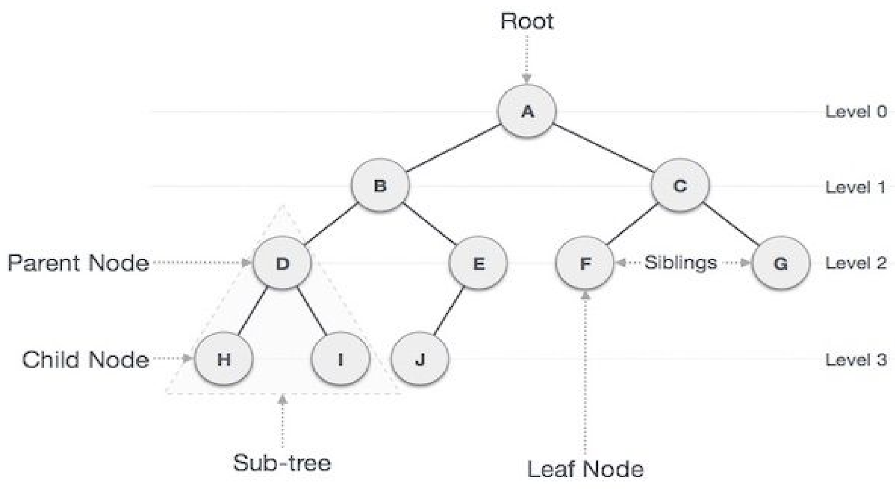
B+ Tree



B Tree VS B+ Tree

B Tree	B+ Tree
Search keys can not be repeatedly stored.	Redundant search keys can be present.
Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
Searching for some data is a slower process	Searching is comparatively faster
Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

This image is not a btree or either b+ tree because its all nodes aren’t in same level



5.4 Query Optimization

Query: A query is a request for information from a database.

Query Plans: A query plan (or query execution plan) is an ordered set of steps used to access data in a SQL relational database management system.

Query Processing: Query Processing is a translation of high-level queries into low-level expression.

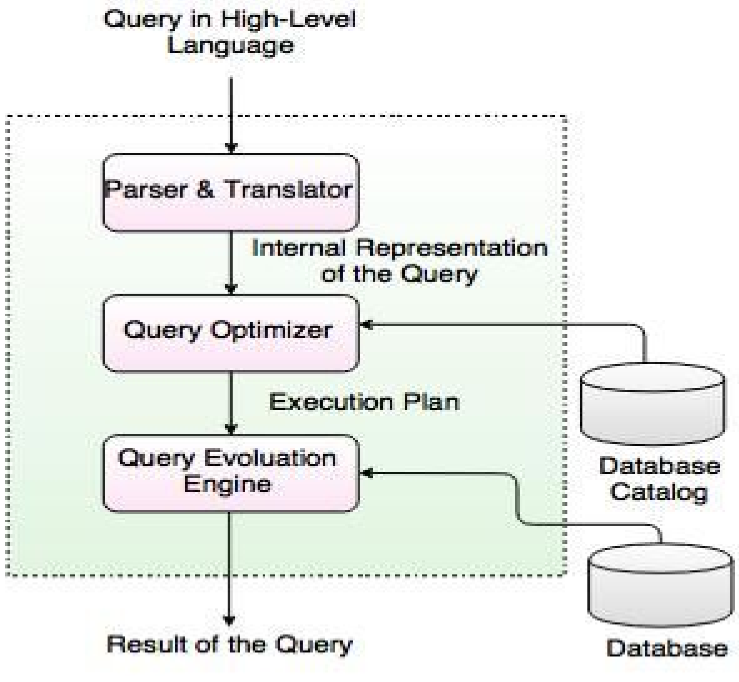


Fig. Query Processing

The user typically writes his requests in SQL language. In order to process and execute this request, DBMS has to convert it into low level – machine understandable language.

Step 1: Any query issued to the database is first picked by query processor. It scans and parses the query into individual tokens and examines for the correctness of query. It checks for the validity of tables / views used and the syntax of the query.

Step 2: convert Sql to relational algebra in different forms and structures.

Step 3: Create query tree or execution plan. In order to transform a given query into a query tree, the query is decomposed into query block (nested query)

Step 4: Query execution

Step 5: Result

query processing is done with the following aim −

- Minimization of response time of query
- Maximize system throughput
- Reduce the amount of memory and storage required for processing.
- Increase parallelism.

Query Optimization: A single query can be executed through different algorithms or re-written in different forms and structures. The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans.

Importance: The goal of query optimization is to reduce the system resources required to fulfill a query, and ultimately provide the user with the correct result set faster.

In Query Optimization, first translate given query to relational algebra and create a query tree.

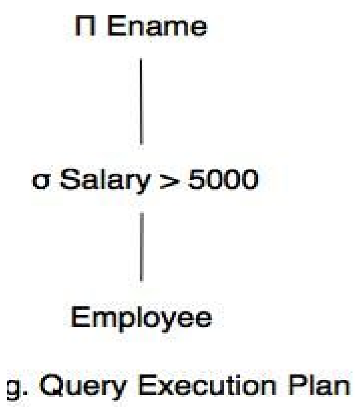
Q1. SELECT Ename FROM Employee WHERE Salary > 5000;

Translated into Relational Algebra Expression

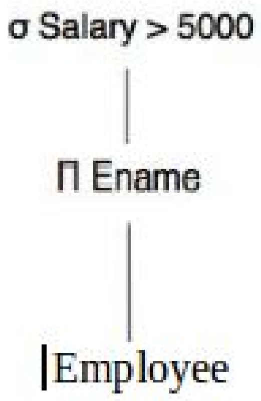
σ Salary > 5000 (π Ename (Employee))

OR

π Ename (σ Salary > 5000 (Employee))



g. Query Execution Plan

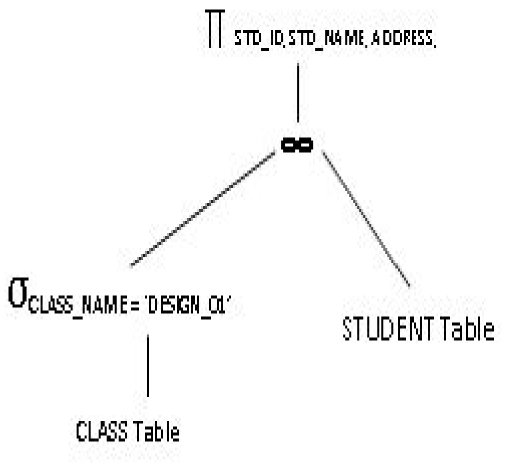


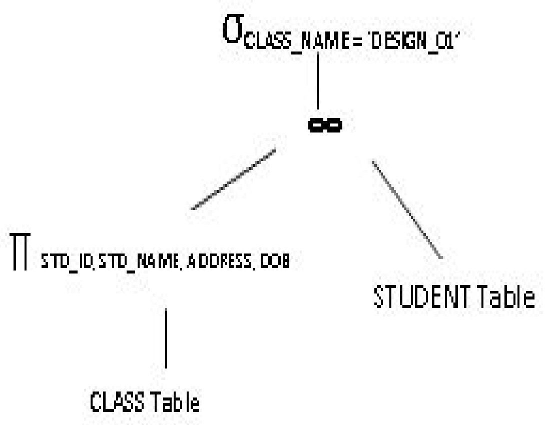
Q2. SELECT STD_ID, STD_NAME, ADDRESS, DOB FROM STUDENT s, CLASS c WHERE s.CLASS_ID = c.CLASS_ID AND c.CLASS_NAME = 'DESIGN_01';

∏ STD_ID, STD_NAME, ADDRESS, DOB (σ CLASS_NAME = 'DESIGN_01' (STUDENT ⋈ CLASS))

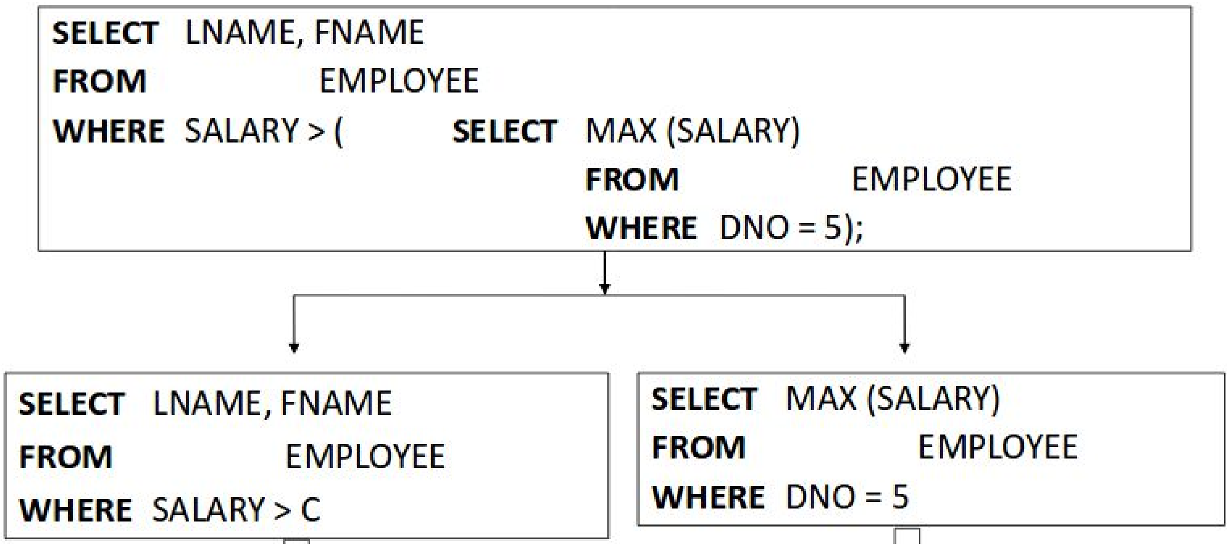
OR

σ CLASS_NAME = 'DESIGN_01' (∏ STD_ID, STD_NAME, ADDRESS, DOB (STUDENT ⋈CLASS))





Q3. SELECT LNAME, FNAME FROM EMPLOYEE WHERE SALARY > (SELECT MAX (SALARY) FROM EMPLOYEE WHERE DNO = 5)



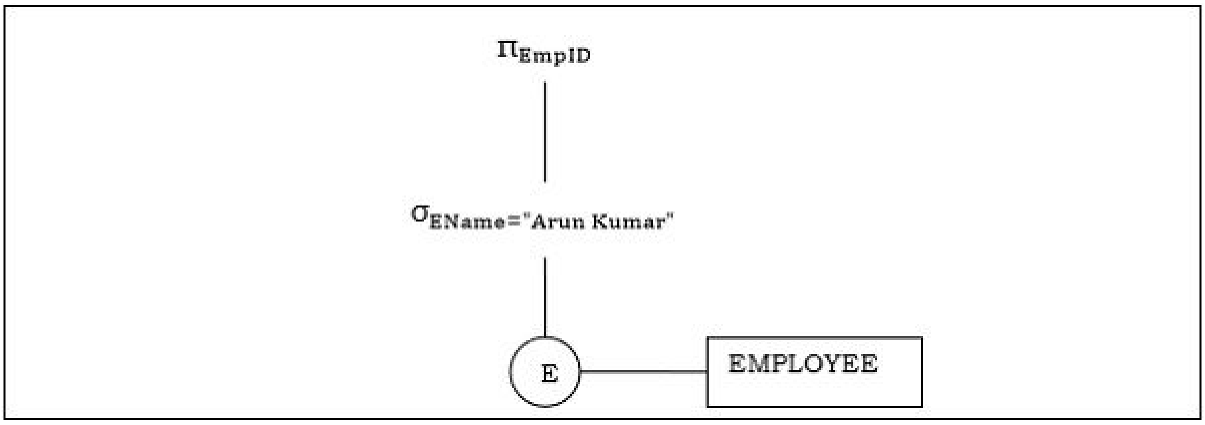
```
SELECT  LNAME, FNAME
FROM                 EMPLOYEE
WHERE  SALARY > (          SELECT   MAX (SALARY)
                                    FROM              EMPLOYEE
                                    WHERE   DNO = 5);
```

```
SELECT   LNAME, FNAME
FROM                EMPLOYEE
WHERE  SALARY > C
```

```
SELECT   MAX (SALARY)
FROM               EMPLOYEE
WHERE  DNO = 5
```

∏ LNAME, FNAME(σ SALARY>C (EMPLOYEE))   ∏ MAX(SALARY)(σ DNO=5 (EMPLOYEE))

Q4. $\pi_{EmpID}(\sigma_{EName="ArunKumar"}(EMPLOYEE))$



Approaches to Query Optimization

Among the approaches for query optimization, exhaustive search and heuristics-based algorithms are mostly used.

Heuristics-based algorithms

This method is also known as rule based optimization. This is based on the equivalence rule on relational expressions; hence the number of combination of queries get reduces here. Hence the cost of the query too reduces. This method creates relational tree for the given query based on the equivalence rules. These equivalence rules by providing an alternative way of writing and evaluating the query, gives the better path to evaluate the query.

- Perform all the selection operation as early as possible in the query.
- Perform all the projection as early as possible in the query.
- Next step is to perform most restrictive joins and selection operations.

Suppose we have a query to retrieve the students with age 18 and studying in class DESIGN_01. We can get all the student details from STUDENT table, and class details from CLASS table. We can write this query in two different ways.

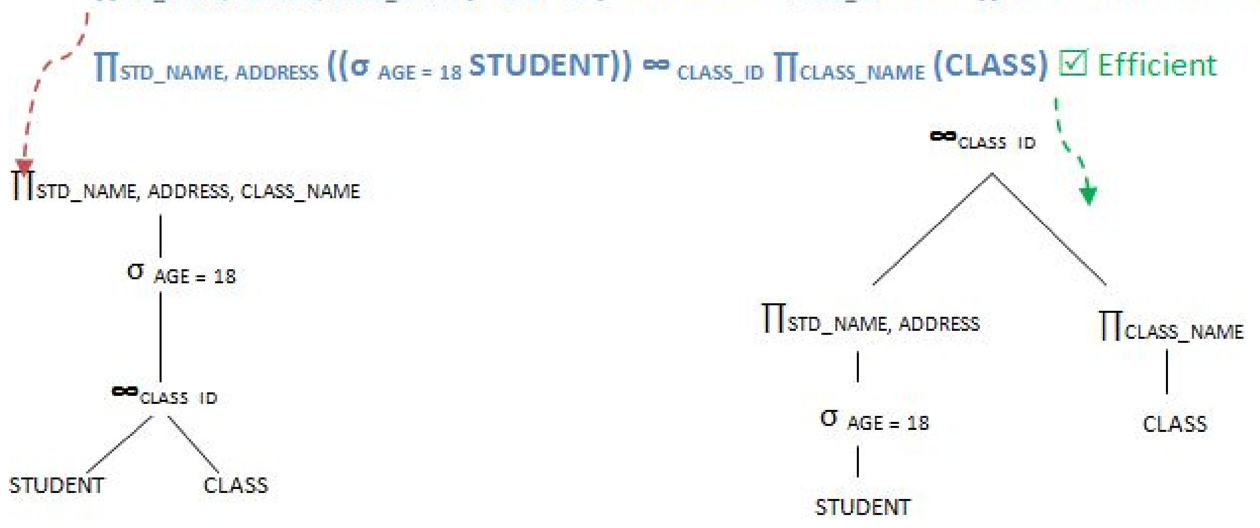
```
SELECT std.* FROM STUDENT std, CLASS cls
WHERE std.CLASS_ID = cls.CLASS_ID
AND std.AGE = 18
AND cls.CLASS_NAME = 'DESIGN_01';
```

```
SELECT std.* FROM
   (SELECT * FROM STUDENT WHERE AGE = 18) std,
   (SELECT * FROM CLASS WHERE CLASS_NAME = 'DESIGN_01') cls
WHERE std.CLASS_ID = cls.CLASS_ID
```

Here both the queries will return same result. But when we observe them closely we can see that first query will join the two tables first and then applies the filters. That means, it traverses whole table to join, hence the number of records involved is more. But he second query, applies the filters on each table first. This reduces the number of records on each table (in class table, the number of record reduces to one in this case!). Then it joins these intermediary tables. Hence the cost in this case is comparatively less.

$$\Pi_{STD\_NAME,\ ADDRESS,\ CLASS\_NAME}\ (\sigma_{AGE\ =\ 18}\ (STUDENT \bowtie_{CLASS\_ID} CLASS))$$ ☒ Not so efficient

$$\Pi_{STD\_NAME,\ ADDRESS}\ ((\sigma_{AGE\ =\ 18}\ STUDENT)) \bowtie_{CLASS\_ID} \Pi_{CLASS\_NAME}\ (CLASS)$$ ☑ Efficient


∏STD_NAME, ADDRESS, CLASS_NAME
σ AGE = 18
∞CLASS_ID
STUDENT
CLASS
∞CLASS_ID
∏STD_NAME, ADDRESS
∏CLASS_NAME
σ AGE = 18
STUDENT
CLASS