UNIT-IV: IMPLEMENTATION TECHNIQUES

UNIT-IV: IMPLEMENTATION TECHNIQUES: RAID(L2) – File Organization(L2) – Organization of Records in Files(L2) – Indexing and Hashing(L2) – Ordered Indices(L2) – B+ Tree Index Files(L3) – B Tree Index Files(L3) – Static Hashing(L2) – Dynamic Hashing(L2) – Query Processing Overview(L2) – Query Optimization using Heuristics and Cost Estimation(L3).

RAID

- RAID stands for Redundant Array of Independent Disks. This is a technology in which multiple secondary disks are connected together to increase the performance, data redundancy or both.
- For achieving the data redundancy - in case of disk failure, if the same data is also backed up onto another disk, we can retrieve the data and go on with the operation.
- It consists of an array of disks in which multiple disks are connected to achieve different goals.
- The main advantage of RAID, is the fact that, to the operating system the array of disks can be presented as a single disk.

Need for RAID

- RAID is a technology that is used to increase the performance.
- It is used for increased reliability of data storage.
- An array of multiple disks accessed in parallel will give greater throughput than a single disk.
- With multiple disks and a suitable redundancy scheme, your system can stay up and running when a disk fails, and even while the replacement disk is being installed and its data restored.

Features

(1) RAID is a technology that contains the set of physical disk drives.

(2) In this technology, the operating system views the separate disks as a single logical disk.

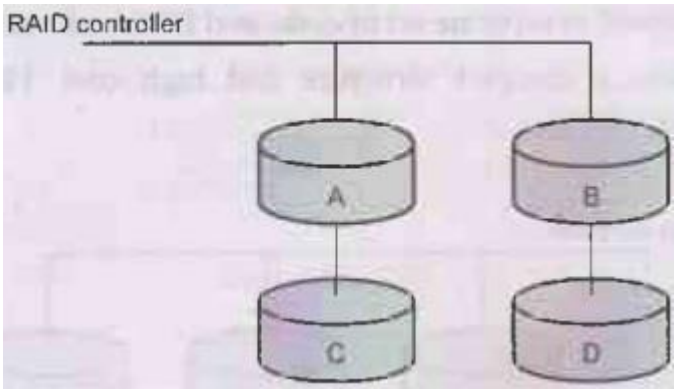
(3) The data is distributed across the physical drives of the array.

(4) In case of disk failure, the parity information can be helped to recover the data.

RAID Levels

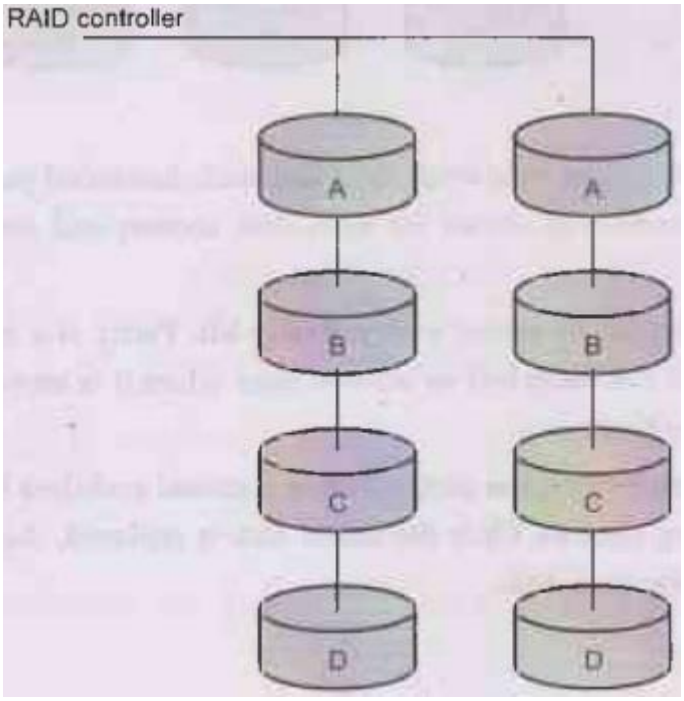
Level: RAID 0

- In this level, data is broken down into blocks and these blocks are stored across all the disks.
- Thus striped array of disks is implemented in this level. For instance in the following figure blocks "A B" form a stripe.
- There is no duplication of data in this level so once a block is lost then there is no int lovol diri way recover it.
- The main priority of this level is performance and not the reliability.



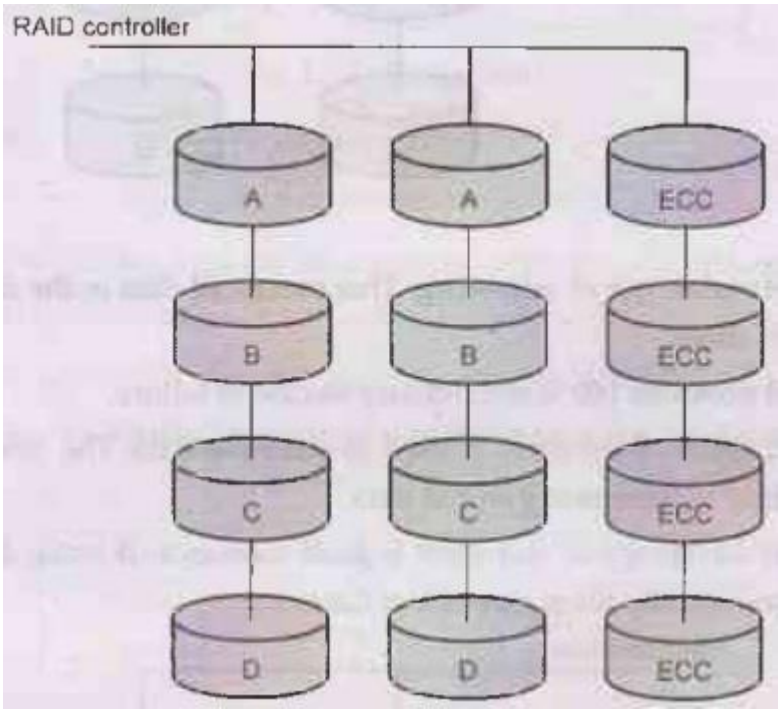
Level: RAID 1

- This level makes use of mirroring. That means all data in the drive is duplicated to another drive.
- This level provides 100% redundancy in case of failure.
- Only half space of the drive is used to store the data. The other half of drive is just a mirror to the already stored data.
- The main advantage of this level is fault tolerance. If some disk fails then the other automatically takes care of lost data.



Level: RAID 2

- This level makes use of mirroring as well as stores Error Correcting Codes (ECC) for its data striped on different disks.
- The data is stored in separate set of disks and ECC is stored another set of disks.
- This level has a complex structure and high cost. Hence it is not used commercially.

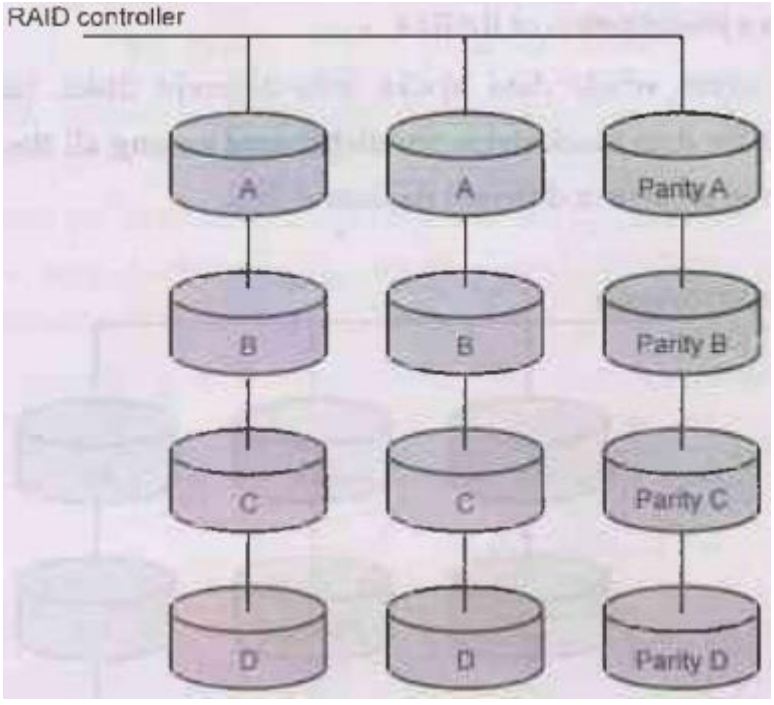


Level: RAID 3

• This level consists of byte-level stripping with dedicated parity. In this level, the parity information is stored for each disk section and written to a dedicated. parity drive.

• We can detect single errors with a parity bit. Parity is a technique that checks whether data has been lost or written over when it is moved from one place in storage to another.

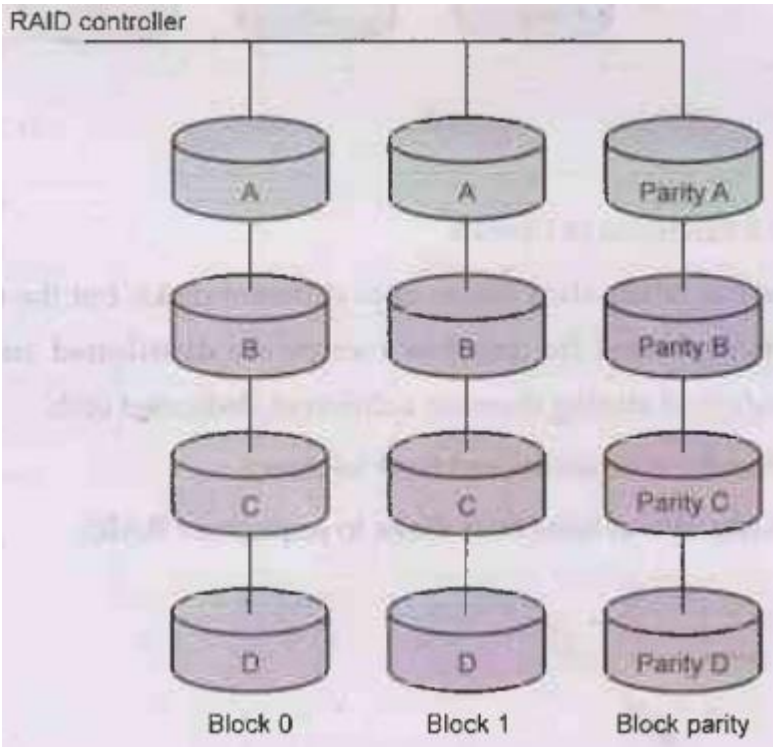
• In case of disk failure, the parity disk is accessed and data is reconstructed from the remaining devices. Once the failed disk is replaced, the missing data can be restored on the new disk.



Level: RAID 4

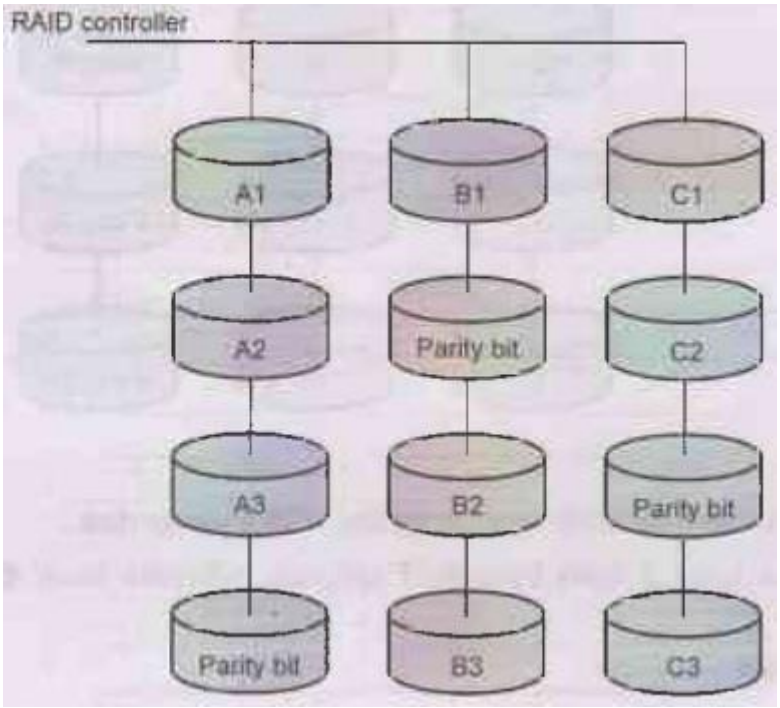
• RAID 4 consists of block-level stripping with a parity disk.

• Note that level 3 uses byte-level striping, whereas level 4 uses block-level striping.



Level: RAID 5

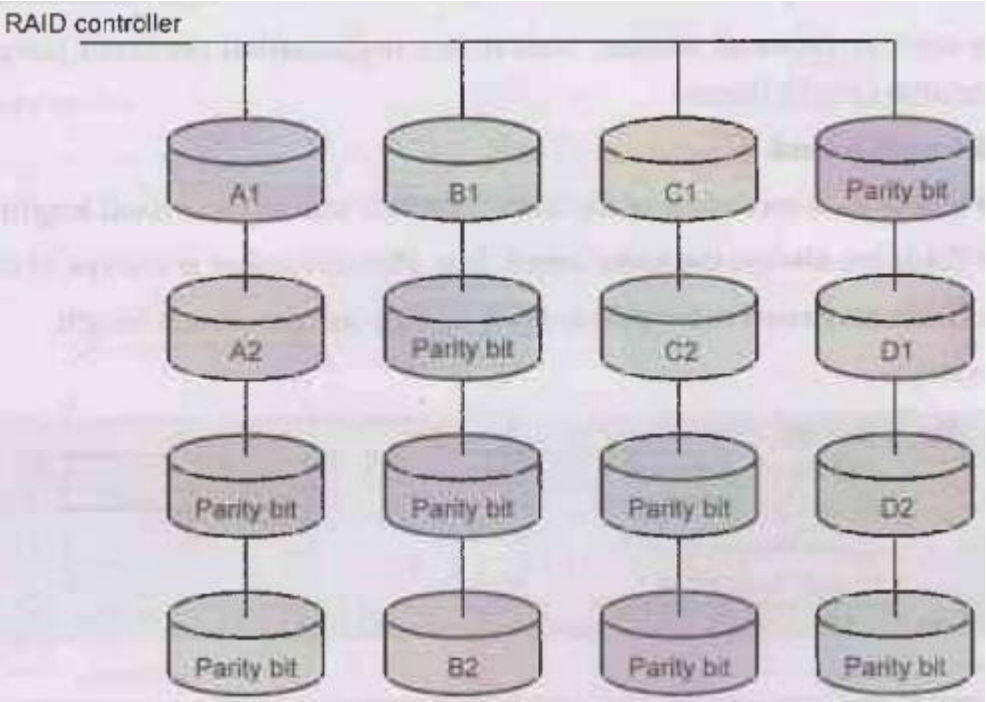
- RAID 5 is a modification of RAID 4.
- RAID 5 writes whole data blocks onto different disks, but the parity bits generated for data block stripe are distributed among all the data disks rather than storing them on a different dedicated disk.



Level: RAID 6

- RAID 6 is a extension of Level 5
- RAID 6 writes whole data blocks onto different disks, but the two independent parity bits generated for data block stripe are distributed among all the data disks rather than storing them on a different dedicated disk.
- Two parities provide additional fault tolerance.

• This level requires at least four disks to implement RAID.



The factors to be taken into account in choosing a RAID level are :

Monetary cost of extra disk-storage requirements.

1. Performance requirements in terms of number of I/O operations.

2. Performance when a disk has failed.

3. Performance during rebuild

File Organization

• A file organization is a method of arranging records in a file when the file is stored on disk.

• A file is organized logically as a sequence of records.

• Record is a sequence of fields.

• There are two types of records used in file organization (1) Fixed Length Record (2) Variable Length Record.

(1) Fixed length record

• A file where each records is of the same length is said to have fixed length records.

• Some fields are always the same length (e.g. Phone Number is always 10 characters).

• Some fields may need to be 'padded out' so they are the correct length. • For example - *type Employee = record EmpNo varchar(4)*

Ename varchar(10)

Salary integer(5)

Phone varchar(10)

End

EmpNo	EName	Salary	Phone
1111	AAA	1000	1111111111
2222	BBB	2000	2222222222
3333	CCC	3000	3333333333
4444	DDD	4000	4444444444

For instance the first record of example file can be stored as

1	1	1	1	A	A	A								1
0	0	0		1	1	1	1	1	1	1	1	1	1	

Thus total 29 bytes are required to store.

Advantage:

• Access is fast because the computer knows where each record starts.

Disadvantage:

(1) Due to fixed size, some larger sized record may cross the block boundaries. That of means part of record will be stored in one block and other part of the record may be stored in some another block. Thus we may require two block access for each read or write.

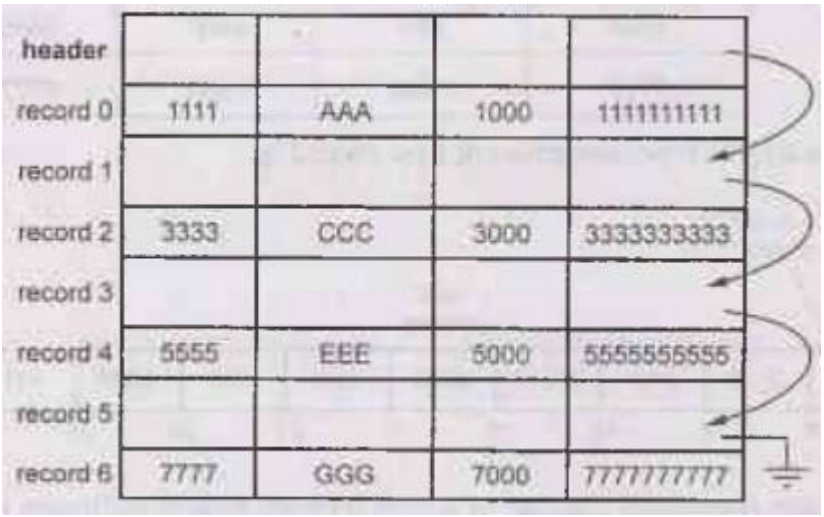
(2) It is difficult to delete the record from this structure. If some intermediate record is deleted from the file then the vacant space must be occupied by next subsequent records.

• When a record is deleted,we could move the record that came after it into the space formerly occupied by the deleted record. But this may require moving of multiple records to occupy the vacant space. This is an undesirable solution to fill up vacant space of deleted records.

• Another approach is to use a file header. At the beginning of the file, we allocate a certain number of bytes as a file header. The header will contain information such as address of the first record whose contents are deleted. We use this first record to store the address of the second available record and so on. Thus the stored record addresses are referred as pointer while the deleted records thus form a linked list which is called as free list. **For example** - Consider the employee record in a file is-

	Emp No	Ename	Salary	Phone
record 0	1111	AAA	1000	1111111111
record 1	2222	BBB	2000	2222222222
record 2	3333	CCC	3000	3333333333
record 3	4444	DDD	4000	4444444444
record 4	5555	EEE	5000	5555555555
record 5	6666	FFF	6000	6666666666
record 6	7777	GGG	7000	7777777777

The representation of records maintaining free list after deletion of record 1,3 and 5



• On insertion of a new record, we use the record pointed to by the header. We change the header pointer to point to the next available record. If no space is available, we add the new record to the end of the file.

(2) Variable length record

• Variable-length records arise in a database in several ways:

(i) Storage of multiple record types in a file.

(ii) Record types that allow variable lengths for one or more fields.

(iii) Record types that allow repeating fields such as arrays or multisets.

• The representation of a record with variable-length attributes typically has two parts:

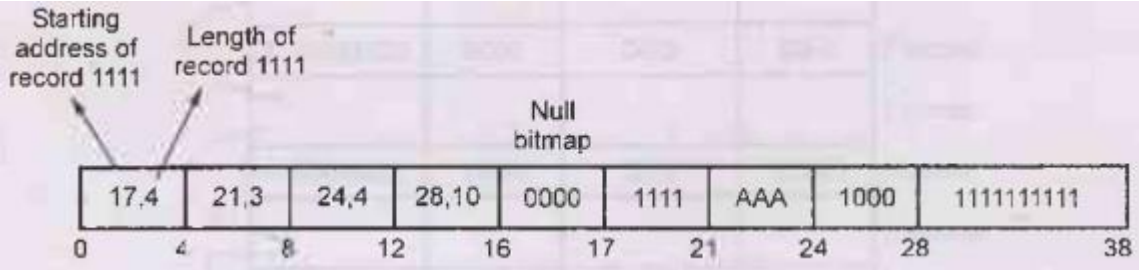
a) an initial part with fixed length attributes. This initial part of the record is represented by a pair (offset, length). The offset denotes the starting address of the record while the length represents the actual length of the length.

b) followed by data for variable length attributes.

• For example - Consider the employee records stored in a file as

record 0	1111	AAA	1000	1111111111
record 1	2222	BBB	2000	2222222222
record 2	3333	CCC	3000	3333333333
record 3	4444	DDD	4000	4444444444
record 4	5555	EEE	5000	5555555555
record 5	6666	FFF	6000	6666666666
record 6	7777	GGG	7000	7777777777

• The variable length representation of first record is,



• The figure also illustrates the use of a null bitmap, which indicates which attributes of the record have a null value.

• The variable length record can be stored in blocks. A specialized structure called slotted page structure is commonly used for organizing the records within a block. This structure is as shown by following Fig. 4.2.1.

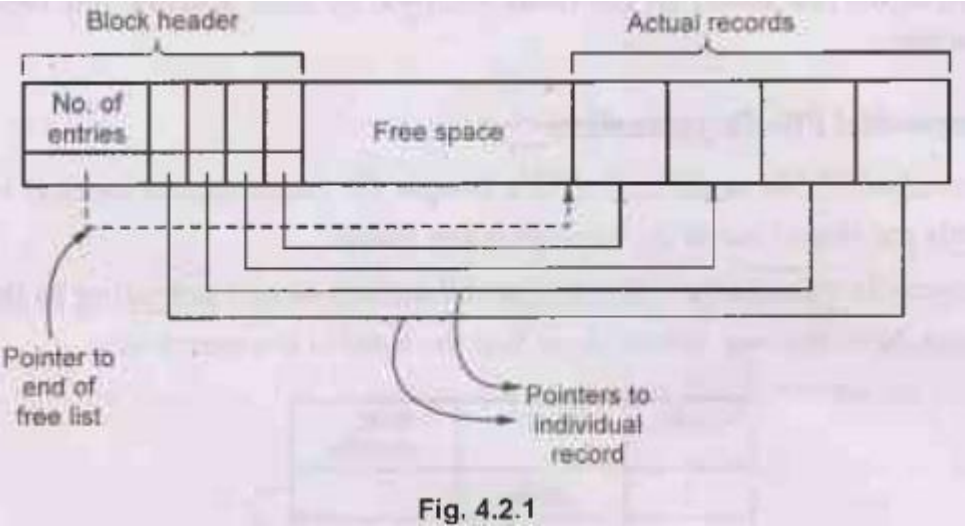


Fig. 4.2.1

• This structure can be described as follows -

• At the beginning of each block there is a block header which contains -

i. Total number of entries (i.e. records).

ii. Pointer to end of free list. iii. Followed by an array of entries that contain size and location of each record.

• The actual records are stored in contiguous block. Similarly the free list is also maintained as continuous block.

• When a new record is inserted then the space is allocated from the block of free list.

• Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.

Organization of Records in Files

There are three commonly used approaches of organizing records in file -

(1) **Heap file organization:** Any record can be placed anywhere in the file where there is a space. There is no ordering for placing the records in the file. Generally single file is used.

(2) **Sequential file organization:** Records are stored in sequential order based on the value of search key.

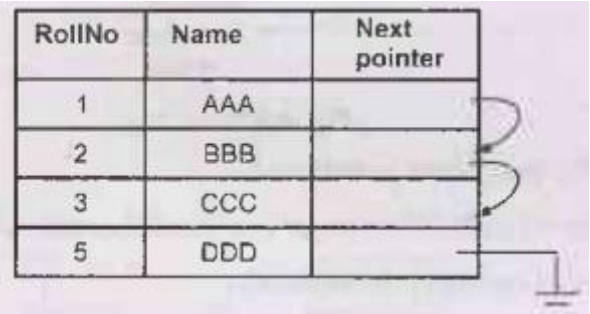
Hashing file organization: A hash function is used to obtain the location of the record in the file. Based on the value returned by hash function, the record is stored in the file.

(3) Sequential File Organization

The sequential file organization is a simple file organization method in which the records are stored based on the search key value.

• For example - Consider following set of records stored according to the RollNo of student.

Note that we assume here that the RollNo is a search key.



RollNo	Name	Next pointer
1	AAA	
2	BBB	
3	CCC	
5	DDD	

Now if we want to insert following record

4	EEE

Then, we will insert it in sorted order of RollNo and adjust the pointers accordingly.

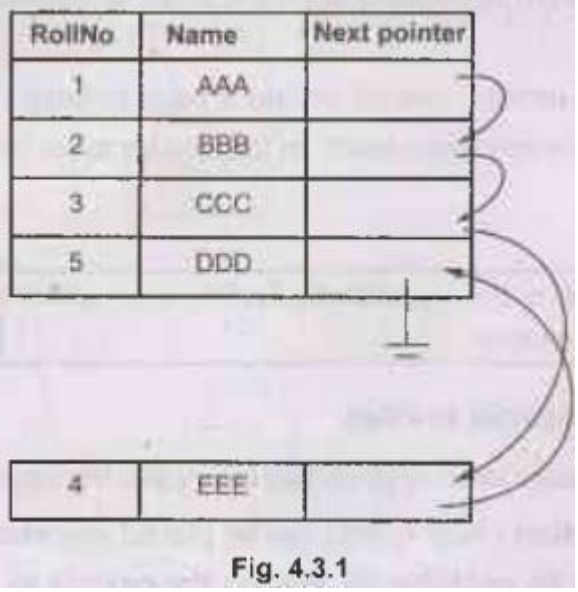


Fig. 4.3.1

- However, maintaining physical sequential order is very difficult as there can be several insertions and deletions.cos le
- We can maintain the deletion by next pointer chain.
- For insertion following rules can be applied -
- If there is free space insert record there.
- If no free space, insert the record in an overflow block.
- In either case, pointer chain must be updated.

Multi-table Clustering File Organization

In a multitable clustering file organization, records of several different relations are stored in the same file.

For example - Following two tables Student and Course

Sname	Marks
Ankita	55
Ankita	67
Ankita	86
Prajkta	91

Sname	Cname	City
Ankita	ComputerSci	Chennai
Prajkta	Electronics	Pune

The multitable clustering organization for above tables is,

Ankita	ComputerSci	Chennai
Ankita	55	
Ankita	67	
Ankita	86	
Prajkta	Electronics	Pune
Prajkta	91	

This type of file organization is good for join operations such as Student Course. This file organization results in variable size records.

The pointer chain can be added to the above records to keep track of address of next record. It can be as shown in following Fig. 4.3.2.

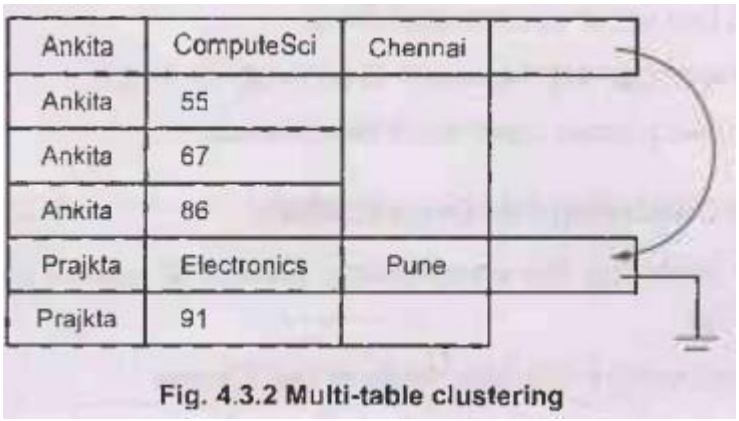


Fig. 4.3.2 Multi-table clustering

Data Dictionary Storage

• **Definition:**Data dictionary is mini database management system that manages the metadata.

• Data dictionaries are helpful to the database administrators in management of database.

• The general structure of data dictionary is as shown in the Fig. 4.4.1.

• The data in the database dictionary is maintained by several programs and generates the reports if required.

• The data dictionary is integrated with the database systems in which the data is controlled by the data dictionaries and is made available to the DBMS software. • Following type of information is maintained by data dictionary -

a) Description of the schema of the database.

b) Detailed information about the physical database design.

c) Description of database users, their roles and their access rights.

d) Description of database transactions.

e) The description of the relationship between database transactions and data items referenced by them.

f) Information about the usage statistics. That means, how many times the queries are raised to the database, how many transactions are made by the DBMS.

• For example - Consider a Student database, in which various fields are RollNo, FirstName, LastName, and CourseID. The data dictionary for this database ban maintains the information about this database. The data dictionary contains the dog column names, Data type of each field and the description of each column of the database.

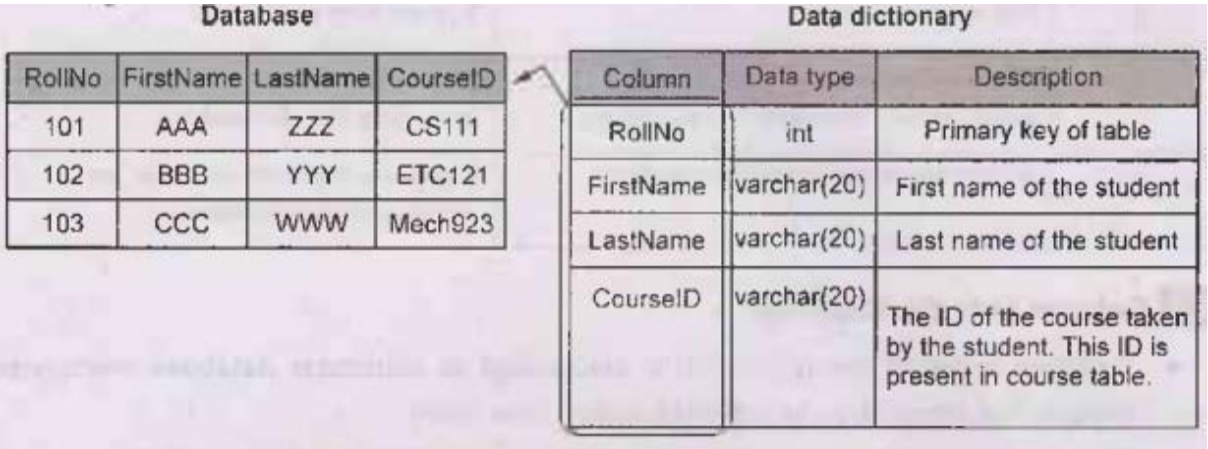


Fig. 4.4.1 : Representation of data dictionary

Active and Passive Data Dictionaries

Active data dictionary

• Active Data dictionary is managed automatically by the database management system.

• They are consistent with current structure.

• In the active data dictionary, when any modification or changes is executed by the DBMS, then this dictionary it also gets modified by the DBMS automatically. • Most of the active data dictionaries are derived from system catalog.

Passive data dictionary

- Passive data dictionary is used only for documentation purpose.
- Passive dictionary is a self-contained application and set of files used for documenting the data processing environment.
- The process of maintaining or modification of the database is manual.
- It is managed by the users of the database systems.

Difference between active and passive data dictionary

Sr. No.	Active data dictionary	Passive data dictionary
1.	The database management system automatically maintains the active data dictionary.	The passive data dictionary is modified whenever the structure of database gets changed.
2.	It is very consistent.	It is not very consistent.
3.	The database management systems automatically manages this dictionary.	The users are responsible for manually managing this dictionary.
4.	It does not require separate database	It requires separate database for working with dictionary.

Column Oriented Storage

- Column oriented storage which is also called as columnar database management system that stores data in columns rather than rows.
- The advantage of column oriented storage is to retrieve the result of query very efficiently.
- It also improves disk I/O performances.

- **Example –**

Row - oriented

ID	Name	Subject	Marks
1	AAA	English	87
2	BBB	Maths	95
3	CCC	History	83

Column - oriented

ID	Name
1.	AAA
2.	BBB
3.	CCC

ID	Subject	Marks
1.	English	87
2.	Maths	95
3.	History	83

- As column oriented database store data by columns instead of rows, it can store more data in smaller amount of memory.
- The data retrieval is done column by column only the columns that need to be required are retrieved. This makes it possible for efficient retrieval of data. Also, large amount of data can be handled.
- It is mainly used in data analytics, business intelligence and data warehousing.

Indexing and Hashing

AU: Dec.-11, Marks 6

- An index is a data structure that organizes data records on the disk to make the retrieval of data efficient.
- The search key for an index is collection of one or more fields of records using which we can efficiently retrieve the data that satisfy the search conditions.
- The indexes are required to speed up the search operations on file of records.
- There are two types of indices -
- Ordered Indices: This type of indexing is based on sorted ordering values.
- Hash Indices: This type of indexing is based on uniform distribution of values across range of buckets. The address of bucket is obtained using the hash function.
- There are several techniques of for using indexing and hashing. These techniques are evaluated based on following factors -
- **Access Types:** It supports various types of access that are supported efficiently.
- **Access Time:** It denotes the time it takes to find a particular data item or set items.
- **Insertion Time:** It represents the time required to insert new data item.
- **Deletion Time:** It represents the time required to delete the desired data item.
- **Space overhead:** The space is required to occupy the index structure. But allocating such extra space is worth to achieve improved performance.

Example 4.4.1 *Since indices speed query processing. Why might they not be kept on several search keys? List as many reasons as possible.* **AU: Dec.-11, Marks 6 Solution:** *Reasons for not keeping several search indices include:*

a. *Every index requires additional CPU time and disk I/O overhead during inserts and deletions.*

b. Indices on non-primary keys might have to be changed on updates, although an index on the primary key might not (as updates typically do not modify the primary key attributes).

c. Each extra index requires additional storage space.

d. For queries which involve conditions on several search keys, efficiency might not be bad even if only some of the keys have indices on them.

Therefore database performance is improved less by adding indices when many indices already exist.

Ordered Indices

Primary and Clustered Indices

Primary index :

- An index on a set of fields that includes the primary key is called a primary index. The primary index file should be always in sorted order.
- The primary indexing is always done when the data file is arranged in sorted order and primary indexing contains the primary key as its search key.
- Consider following scenario in which the primary index consists of few entries as compared to actual data file.

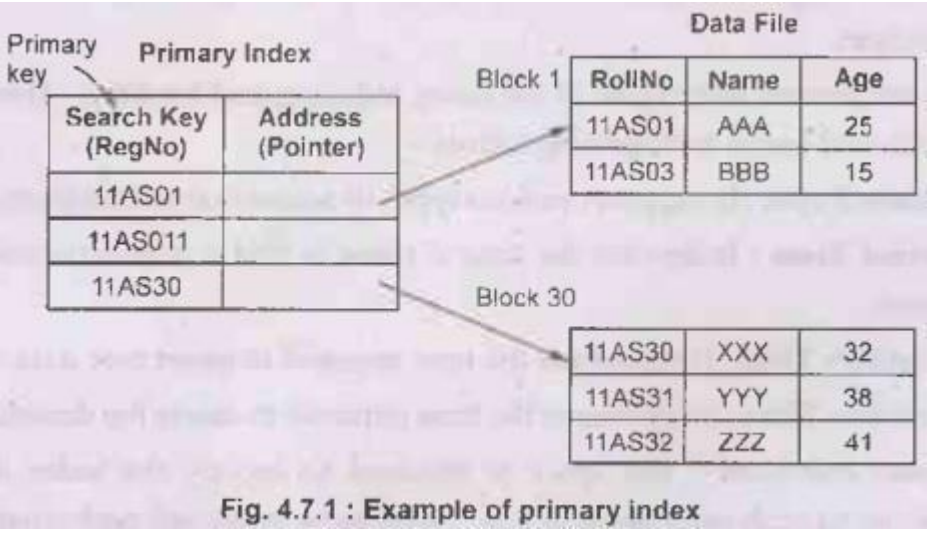


Fig. 4.7.1 : Example of primary index

- Once if you are able to locate the first entry of the record containing block, other entries are stored continuously. For example if we want to search a record for Reg No 11AS32 we need not have to search for the entire data file. With the help of primary index structure we come

to know the location of the record containing the RegNo 11AS30, now when the first entry of block 30 is located, then we can easily no rig locate the entry for 11AS32.

- We can apply binary search technique. Suppose there are n = 300 blocks in a main data file then the number of accesses required to search the data file will be $\log_2 n + 1 = (\log_2 300) + 1 \approx 9$
- If we use primary index file which contains at the most n = 3 blocks then using binary search technique, the number of accesses required to search using the primary index file will be $\log_2 n+1 = (\log_2 3)+1=3$
- This shows that using primary index the access time can be deduced to great extent.

Clustered index:

- In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as clustering index.
- When a file is organized so that the ordering of data records is the same as the ordering of data entries in some index then say that index is clustered, otherwise it is an unclustered index.
- Note that, the data file need to be in sorted order.
- Basically, records with similar characteristics are grouped together and indexes are created for these groups.
- For example, students studying in each semester are grouped together. i.e.; 1st semester students, 2nd semester students, 3rd semester students etc. are grouped.

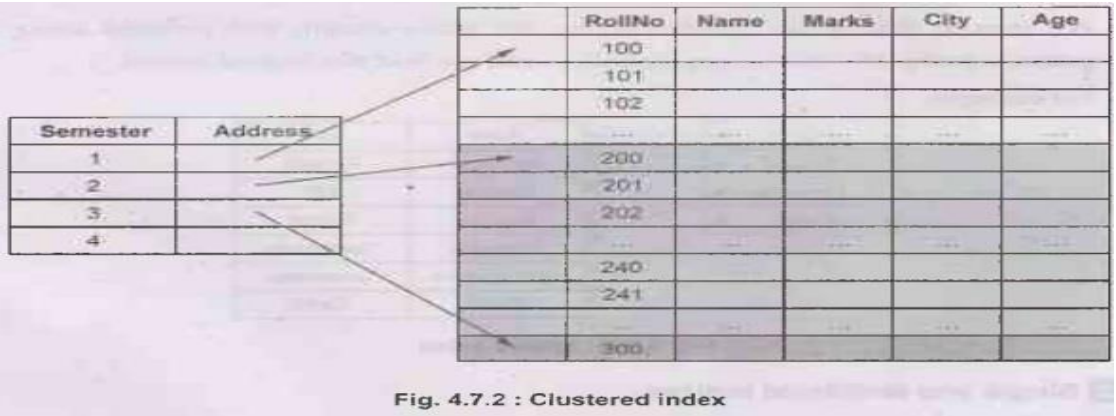


Fig. 4.7.2 : Clustered index

Dense and Sparse Indices

There are two types of ordered indices :

1) Dense index:

- An index record appears for every search key value in file.
- This record contains search key value and a pointer to the actual record.
- For example:

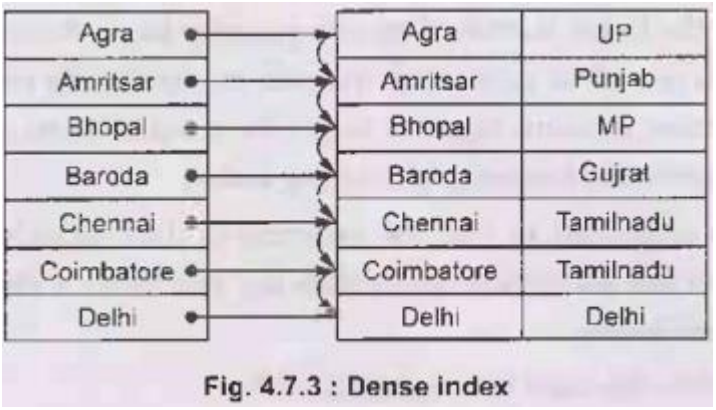


Fig. 4.7.3 : Dense index

2) Sparse index:

- Index records are created only for some of the records.
- To locate a record, we find the index record with the largest 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.
- For example -

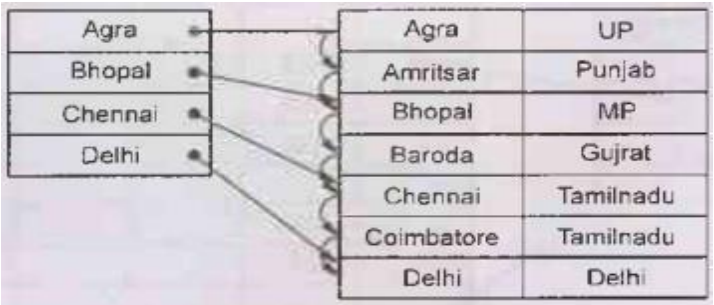


Fig. 4.7.4 : Sparse index

Single and Multilevel Indices

Single level indexing:

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields).
- Each index can be in the following form.

Search Key	Pointer to Record

- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller.
- A binary search on the index yields a pointer to the file record.

• The types of single level indexing can be primary indexing, clustering index or secondary indexing.

• Example: Following Fig. 4.7.5 represents the single level indexing -

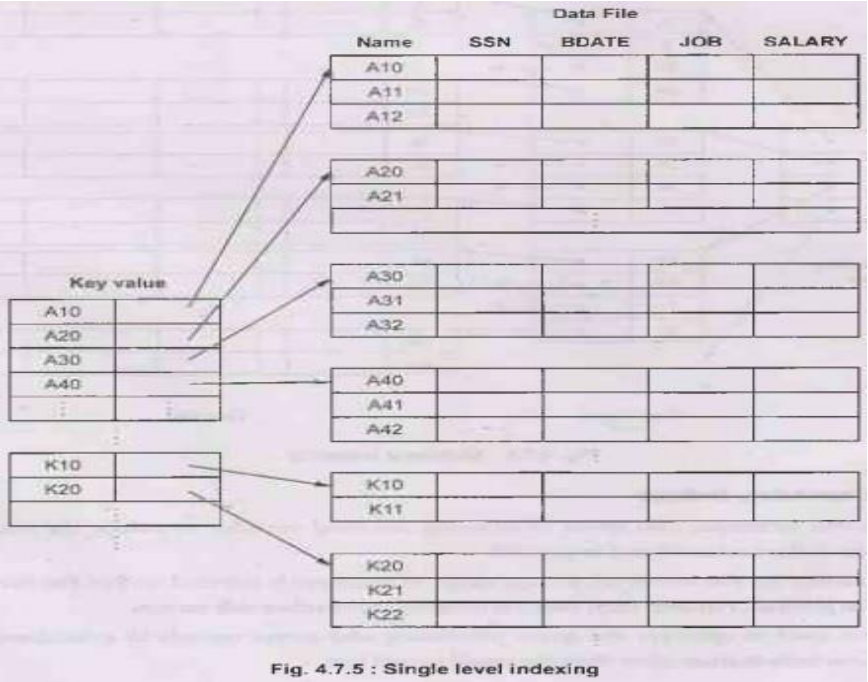


Fig. 4.7.5 : Single level indexing

Multilevel indexing:

• 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.

• The multilevel indexing can be represented by following Fig. 4.7.6.

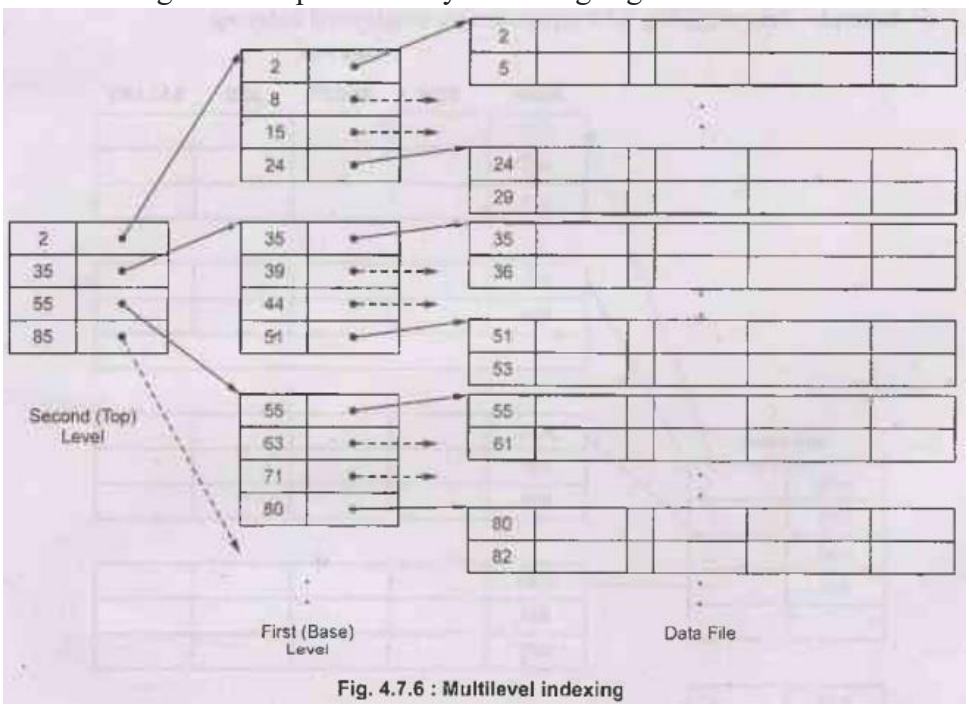


Fig. 4.7.6 : Multilevel indexing

Secondary Indices

- In this technique two levels of indexing are used in order to reduce the mapping size of the first level and in general.
- Initially, for the first level, a large range of numbers is selected so that the mapping size is small. Further, each range is divided into further sub ranges.
- It is used to optimize the query. processing and access records in a database with some information other than the usual search key.

For example -

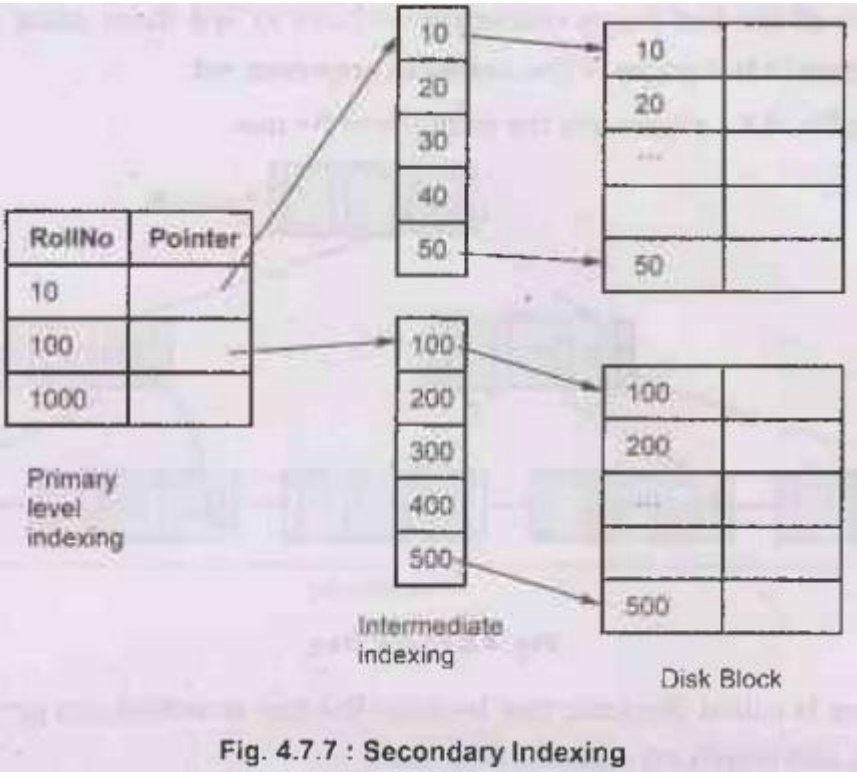


Fig. 4.7.7 : Secondary Indexing

B+ Tree Index Files

AU: May-03,06,16,19, Dec.-17, Marks 16

- The B+ tree is similar to binary search tree. It is a balanced tree in which the internal nodes direct the search.
- The leaf nodes of B+ trees contain the data entries.

Structure of B+ Tree

- The typical node structure of B+ node is as follows –

- It contains up to n – 1 search-key values k_1, k_2, ……, k_{n-1} and n pointers

P_1, P_2,..., P_n

- The search-key values within a node are kept in sorted order; thus, if i < j, then K_i<K_j.

• To retrieve all the leaf pages efficiently we have to link them using page pointers. The sequence of leaf pages is also called as sequence set.

• Following Fig. 4.8.1 represents the example of B+ tree.

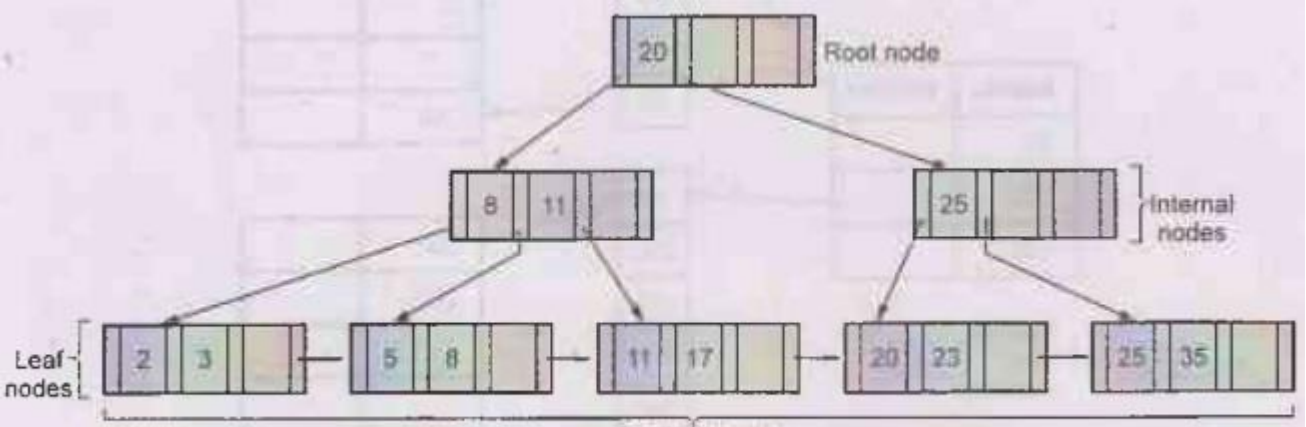


Fig. 4.8.1 : B+ Tree

• The B+ tree is called dynamic tree because the tree structure can grow on insertion of records and shrink on deletion of records.

Characteristics of B+ Tree

Following are the characteristics of B+ tree.

1) The B+ tree is a balanced tree and the operations insertions and deletion keeps the tree balanced.

2) A minimum occupancy of 50 percent is guaranteed for each node except the root.

3) Searching for a record requires just traversal from the root to appropriate leaf.

Insertion Operation

Algorithm for insertion :

Step 1: Find correct leaf L. **Step**

2: Put data entry onto L.

i) If L has enough space, done!

ii) Else, must split L (into L and a new node L2)

• Allocate new node • Redistribute entries evenly

• Copy up middle key.

• Insert index entry pointing to L2 into parent of L.

Step 3: This can happen recursively

i) To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)

Step 4: Splits "grow" tree; root split increases height.

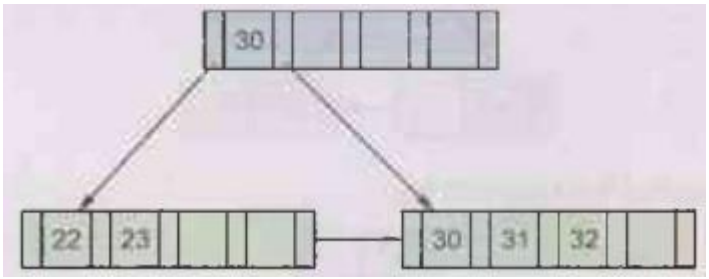
i) Tree growth: gets wider or one level taller at top.

Example 4.8.1 Construct B+ tree for following data. 30,31,23,32,22,28,24,29, where number of pointers that fit in one node are 5.

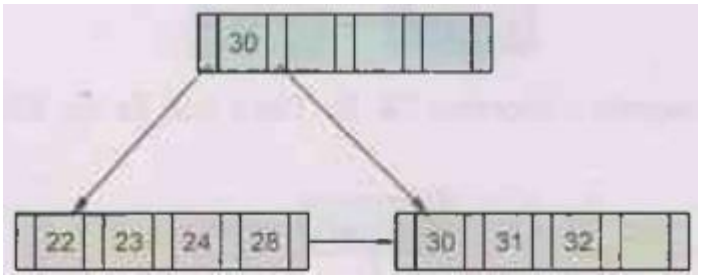
Solution: In B+ tree each node is allowed to have the number of pointers to be 5. That means at the most 4 key values are allowed in each node.

Step 1: Insert 30,31,23,32. We insert the key values in ascending order.

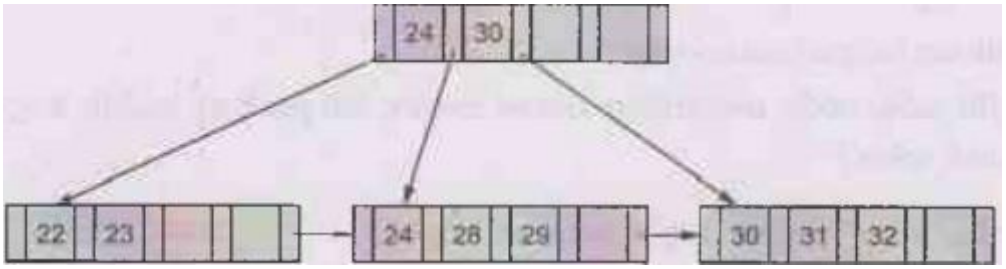
Step 2: Now if we insert 22, the sequence will be 22, 23, 30, 31, 32. The middle key 30, will go up.



Step 3: Insert 28,24. The insertion is in ascending order.



Step 4: Insert 29. The sequence becomes 22, 23, 24, 28, 29. The middle key 24 will go up. Thus we get the B+ tree.

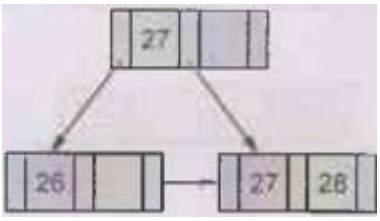


Example 4.8.2 Construct B+ tree to insert the following (order of the tree is 3) 26,27,28,3,4,7,9,46,48,51,2,6 **Solution:**

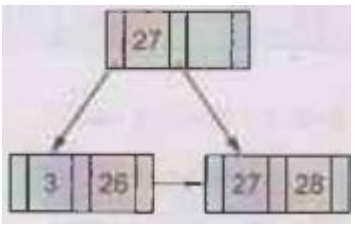
Order means maximum number of children allowed by each node. Hence order 3 means at the most 2 key values are allowed in each node.

Step 1: Insert 26, 27 in ascending order

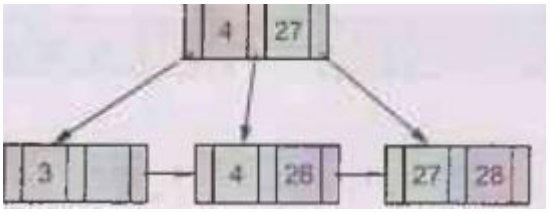
Step 2: Now insert 28. The sequence becomes 26,27,28. As the capacity of the node is full, 27 will go up. The B+ tree will be,



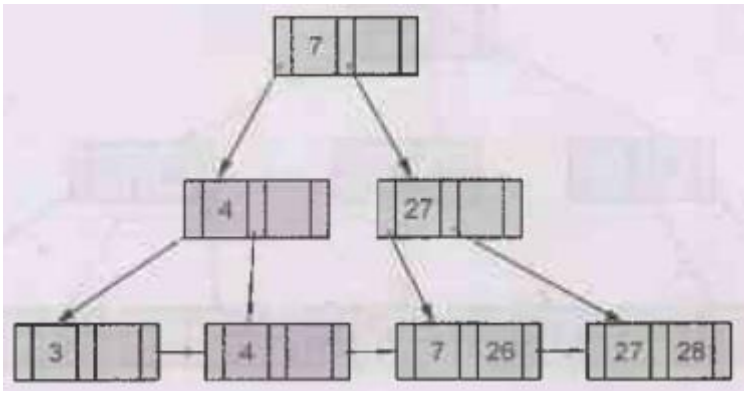
Step 3: Insert 3. The partial B+ Tree will be,



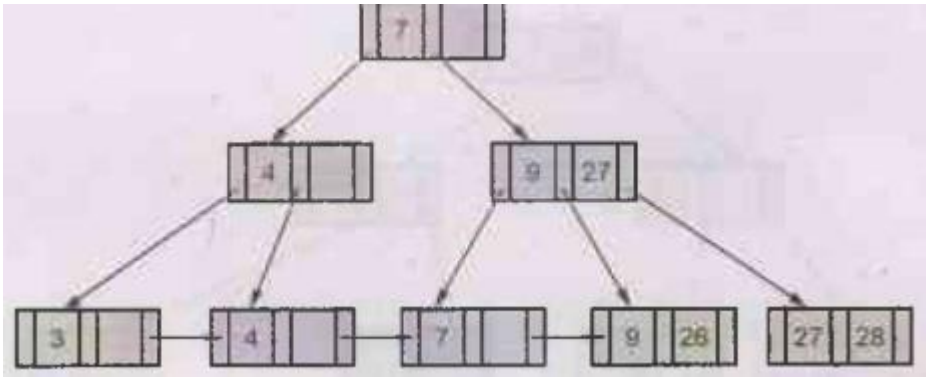
Step 4: Insert 4. The sequence becomes 3,4, 26. The 4 will go up. The partial B+ tree will be –



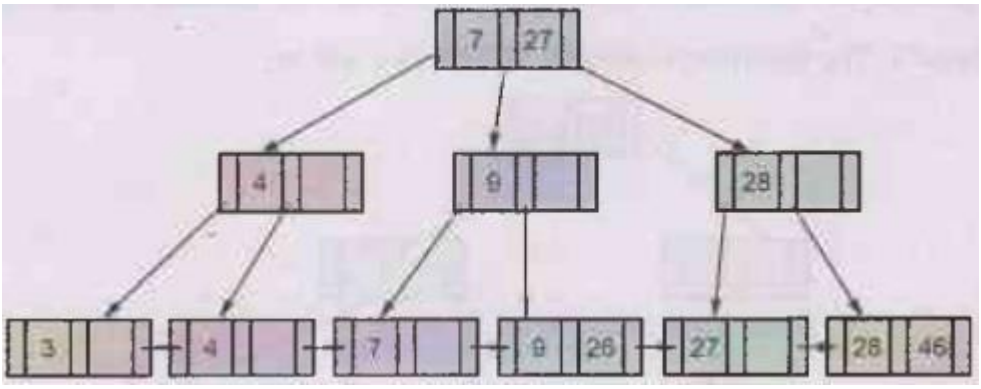
Step 5: Insert 7. The sequence becomes 4,7,26. The 7 will go up. Again from 4,7,27. the 7 will go up. The partial B+ Tree will be,



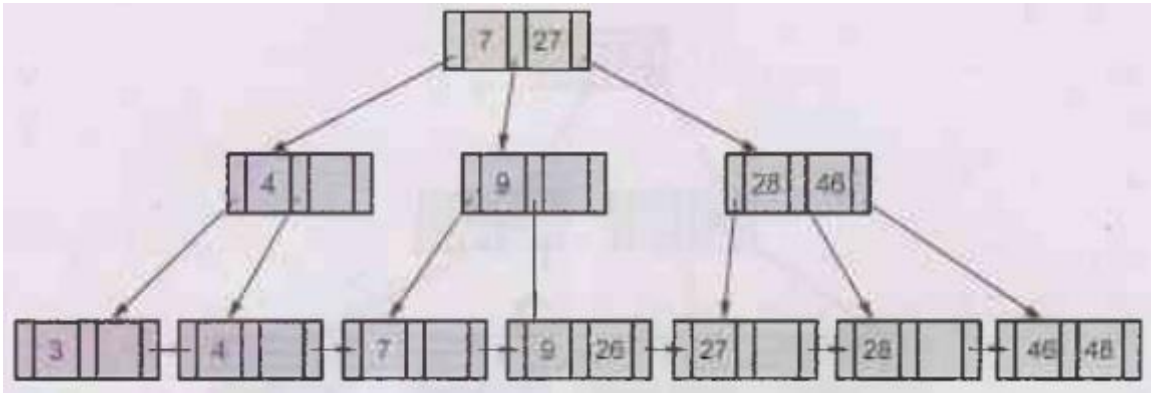
Step 6: Insert 9. By inserting 7,9, 26 will be the sequence. The 9 will go up. The partial B+ tree will be,



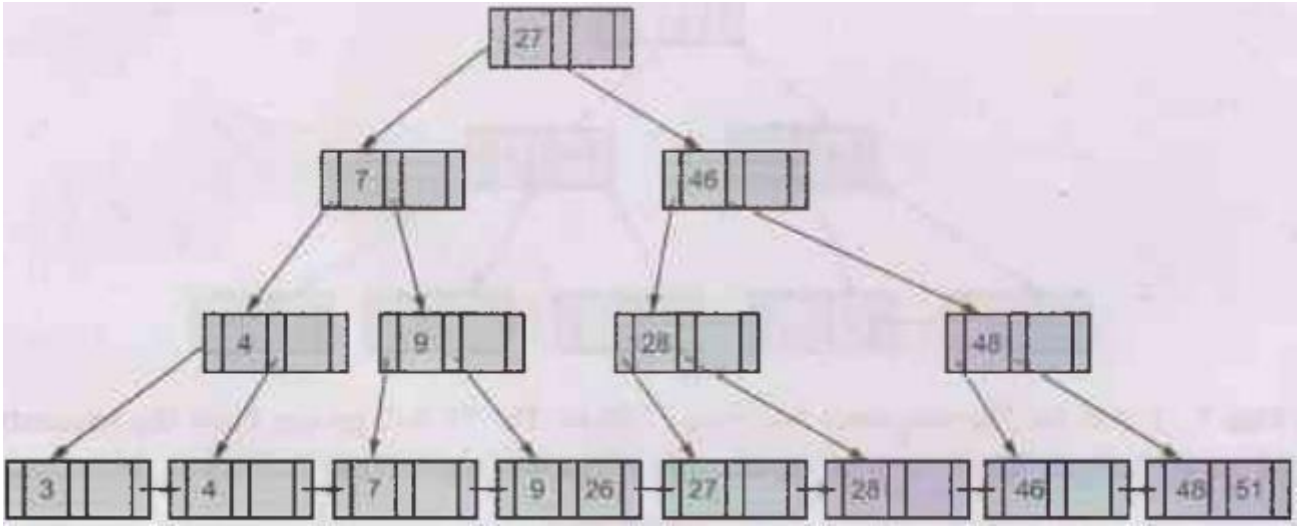
Step 7: Insert 46. The sequence becomes 27,28,46. The 28 will go up. Now the sequence becomes 9, 27, 28. The 27 will go up and join 7. The B+ Tree will be,



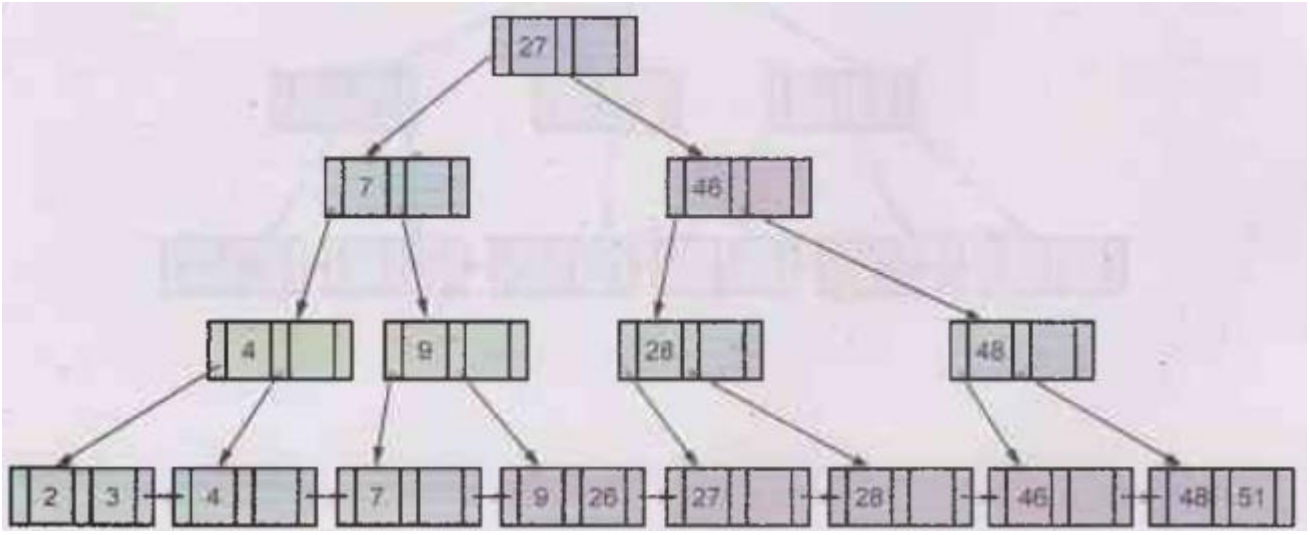
Step 8: Insert 48. The sequence becomes 28,46,48. The 46 will go up. The B+ Tree will become,



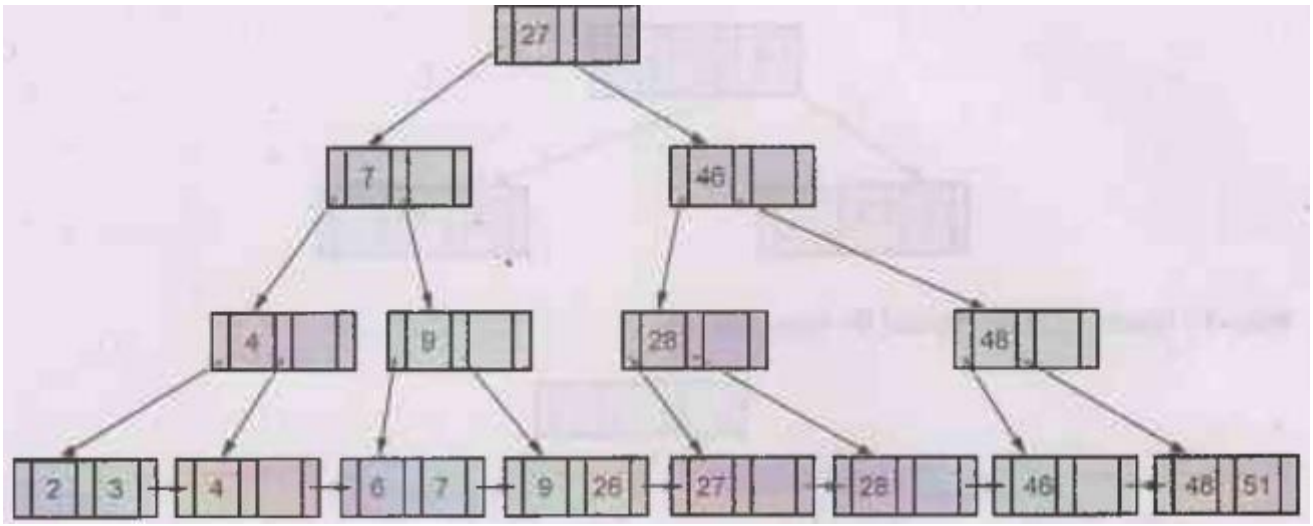
Step 9: Insert 51. The sequence becomes 46,48,51. The 48 will go up. Then the sequence becomes 28, 46, 48. Again the 46 will go up. Now the sequence becomes 7,27, 46. Now the 27 will go up. Thus the B+ tree will be



Step 10: Insert 2. The insertion is simple. The B+ tree will be,



Step 11: Insert 6. The insertion can be made in a vacant node of 7(the leaf node). The final B+ tree will be,



Deletion Operation

Algorithm for deletion:

Step 1: Start at root, find leaf L with entry, if it exists.

Step 2: Remove the entry.

i) If L is at least half-full, done!

ii) If L has only d-1 entries,

• Try to re-distribute, borrowing keys from sibling.

(adjacent node with same parent as L).

• If redistribution fails, merge L and sibling.

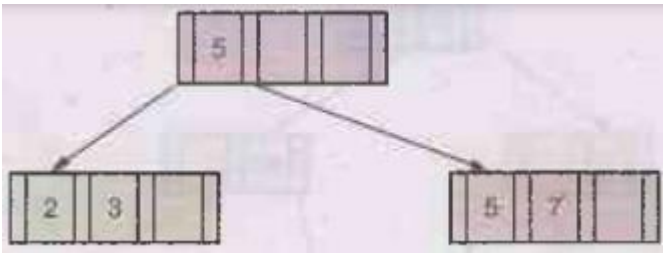
Step 3: If merge occurred, must delete entry (pointing to L or sibling) from parent of L.

Step 4: Merge could propagate to root, decreasing height.

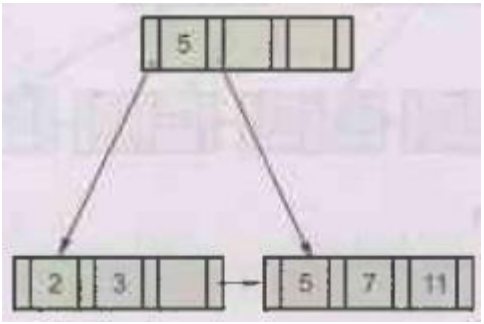
Example 4.8.3 *Construct B+ Tree for the following set of key values (2,3,5,7,11,17,19,23,29,31) Assume that the tree is initially empty and values are added in ascending order. Construct B+ tree for the cases where the number of pointers that fit one node is four. After creation of B+ tree perform following series of operations:*
(a) Insert 9. (b) Insert 10. (c) Insert 8. (d) Delete 23. (e) Delete 19.
Solution: The number of pointers fitting in one node is four. That means each node contains at the most three key values.

Step 1: Insert 2, 3, 5.

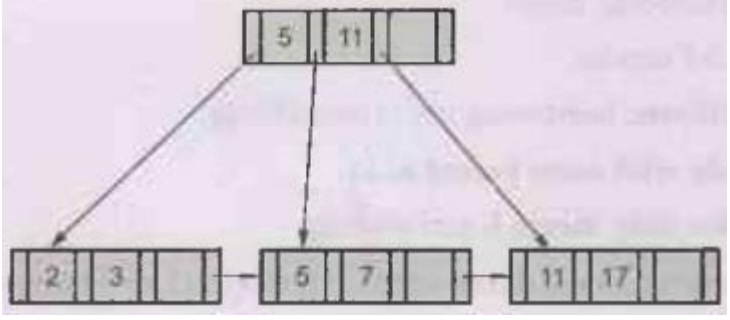
Step 2: If we insert 7, the sequence becomes 2, 3, 5, 7. Since each node can accommodate at the most three key, the 5 will go up, from the sequence 2, 3, 5, 7.



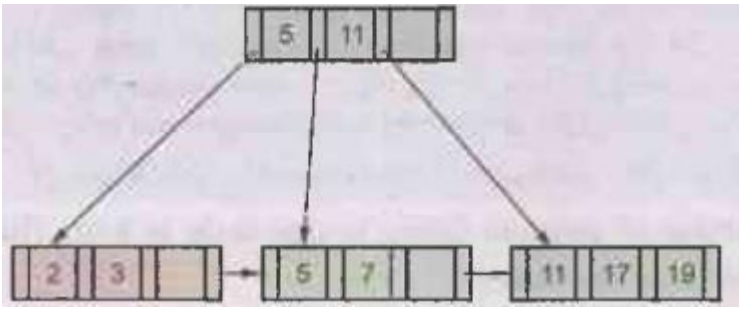
Step 3: Insert 11. The partial B+ tree will be,



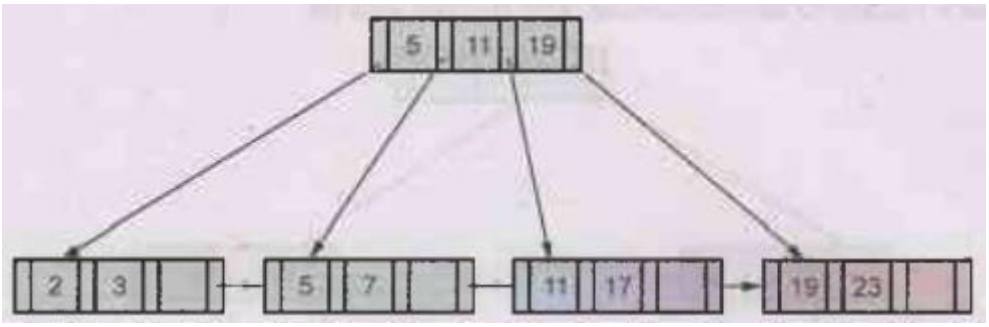
Step 4: Insert 17. The sequence becomes 5,7, 11,17. The element 11 will go up. Then the partial B+ tree becomes,



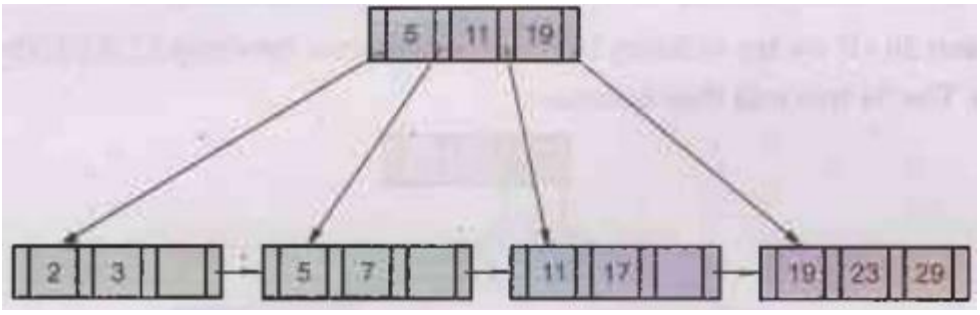
Step 5: Insert 19.



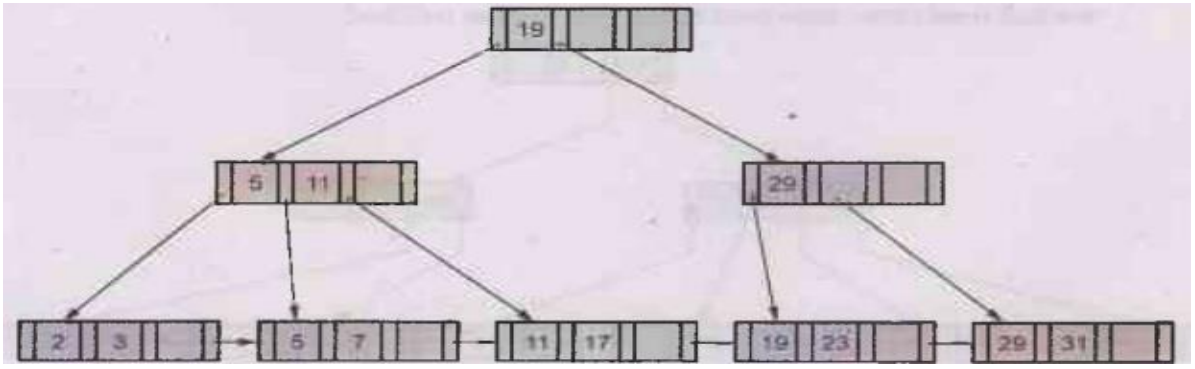
Step 6: Insert 23. The sequence becomes 11,17,19,23. The 19 will go up.



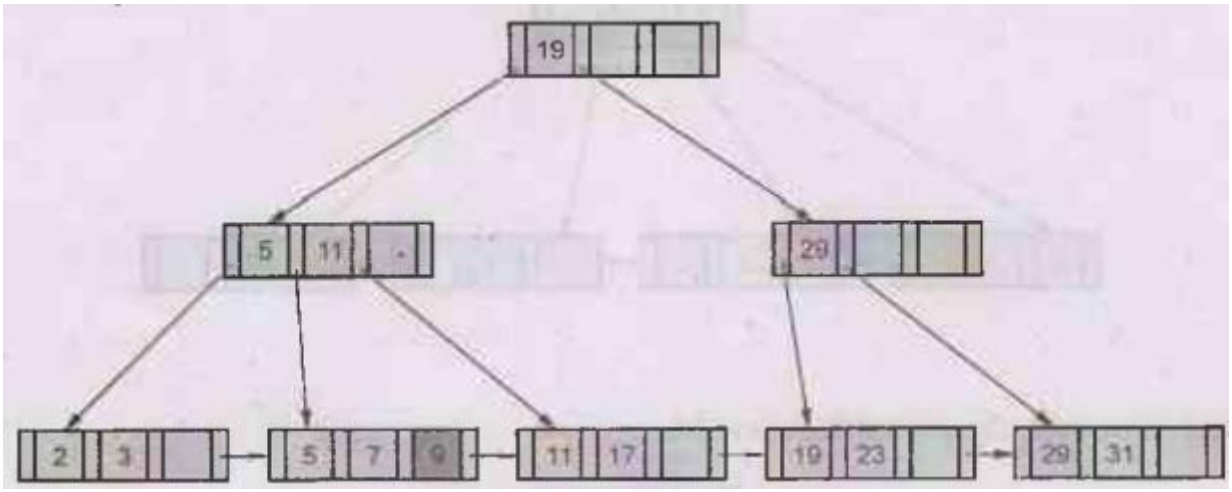
Step 7: Insert 29. The partial B+ tree will be,



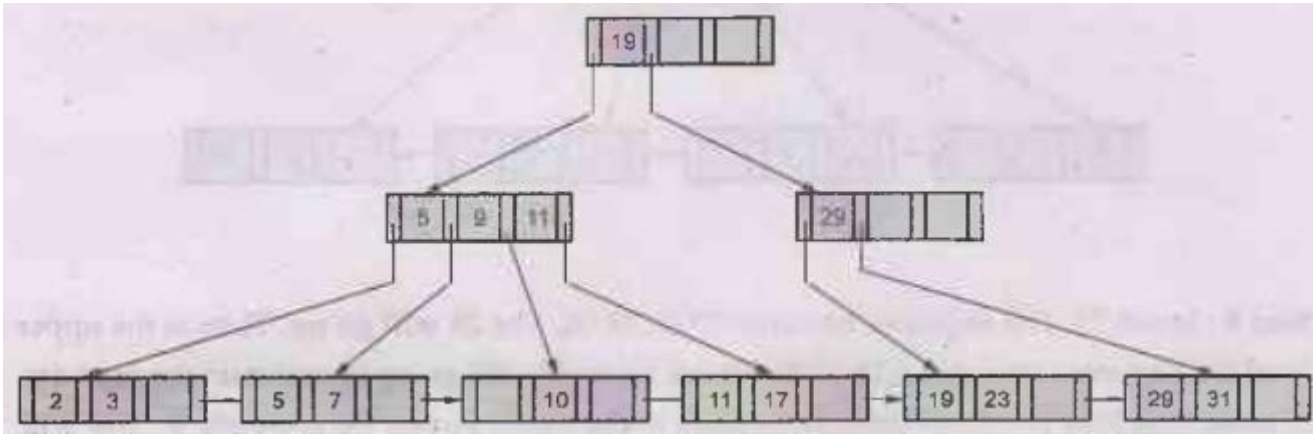
Step 8: Insert 31. The sequence becomes 19,23,29, 31. The 29 will go up. Then at the upper level the sequence becomes 5,11,19,29. Hence again 19 will go up to maintain the capacity of node (it is four pointers three key values at the most). Hence the complete B+ tree will be,



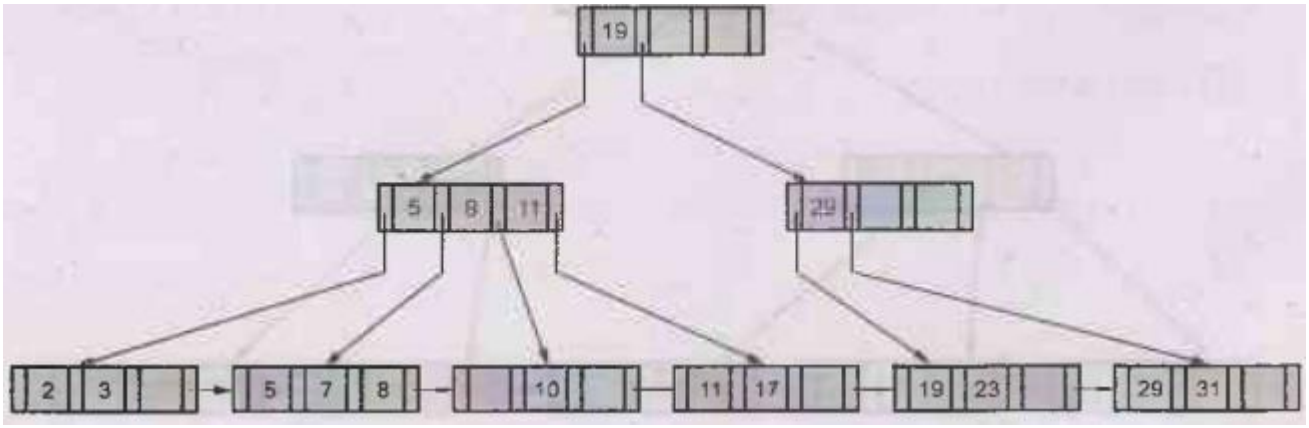
(a) Insertion of 9: It is very simple operation as the node containing 5,7 has one space vacant to accommodate. The B+ tree will be,



(b) Insert 10: If we try to insert 10 then the sequence becomes 5,7,9,10. The 9 will go up. The B+ tree will then become –

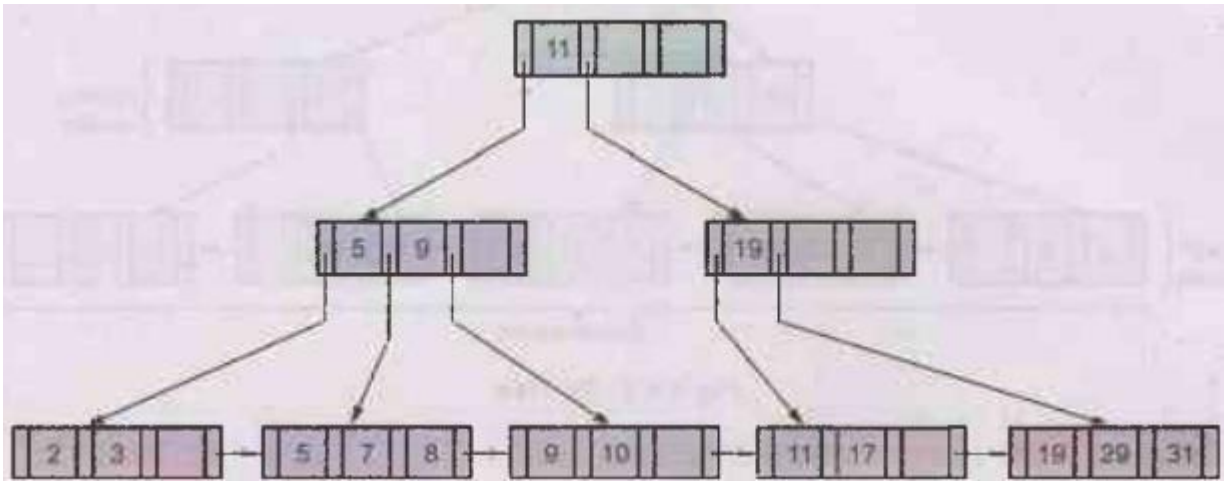


(c) Insert 8: Again insertion of 8 is simple. We have a vacant space at node 5,7. So we just insert the value over there. The B+ tree will be-



(d) Delete 23: Just remove the key entry of 23 from the node 19,23. Then merge the sibling node to form a node 19,29,31. Get down the entry of 11 to the leaf node. Attach the node of 11,17 as a left child of 19.

(e) Delete 19: Just delete the entry of 19 from the node 19,29,31. Delete the internal node key 19. Copy the 29 up as an internal node as it is an inorder successor node.



Search Operation

1. Perform a binary search on the records in the current node.
2. If a record with the search key is found, then return that record.

3. If the current node is a leaf node and the key is not found, then report an unsuccessful search.
4. Otherwise, follow the proper branch and repeat the process. **For example-**

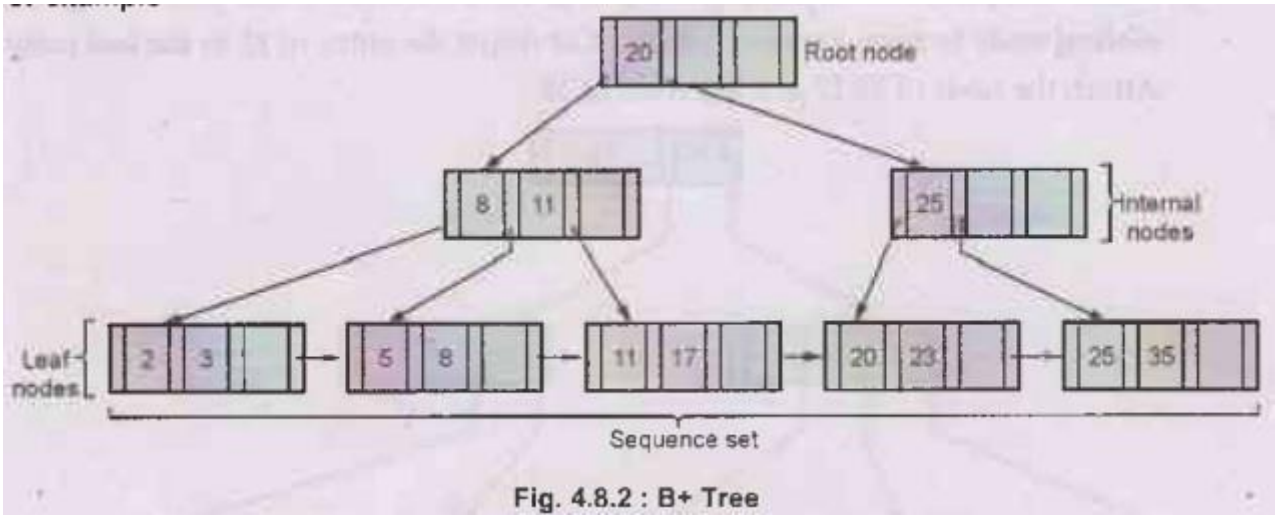


Fig. 4.8.2 : B+ Tree

Consider the B+ tree as shown in above Fig. 4.8.2.

For searching a node 25, we start from the root node -

(1) Compare 20 with key value 25. As 25>20, move on to right branch.

(2) Compare 25 with key value 25. As the match is found we declare, that the given node is present in the B+ tree.

For searching a node 10, we start form the root node -

(1) Compare 20 with key value 10, as 10<20, we follow left branch

(2) Compare 8 with 10, 10>8, then we compare 10 with the next adjacent value of the same node. It is 11, as 10<11, we follow left branch of 11.

(3) We compare 10, with all the values in that node, as match is not found we report unsuccessful search or node is not present in given B+ tree.

Merits of B+ Index Tree Structure

1. In B+ tree the data is stored in leaf node so searching of any data requires scanning only of leaf node alone.
2. Data is ordered in linked list.
3. Any record can be fetched in equal number of disk accesses.
4. Range queries can be performed easily as leaves are linked up.
5. Height of the tree is less as only keys are used for indexing.
6. Supports both random and sequential access.

Demerits of B+ Index Tree Structure

1. Extra insertion of non leaf nodes.
2. There is space overhead.

B Tree Index Files

AU: Dec.-12,14, May-08, Marks 16

- B-tree indices are similar to B+-tree indices.
- The primary distinction between the two approaches is that a B-tree eliminates the redundant storage of search-key values.
- B-tree is a specialized multiway tree used to store the records in a disk.
- There are number of subtrees to each node. So that the height of the tree is relatively small. So that only small number of nodes must be read from disk to retrieve an item. The goal of B- trees is to get fast access of the data.
- A B-tree allows search-key values to appear only once (if they are unique), unlike a B+- tree, where a value may appear in a nonleaf node, in addition to appearing in a leaf node.

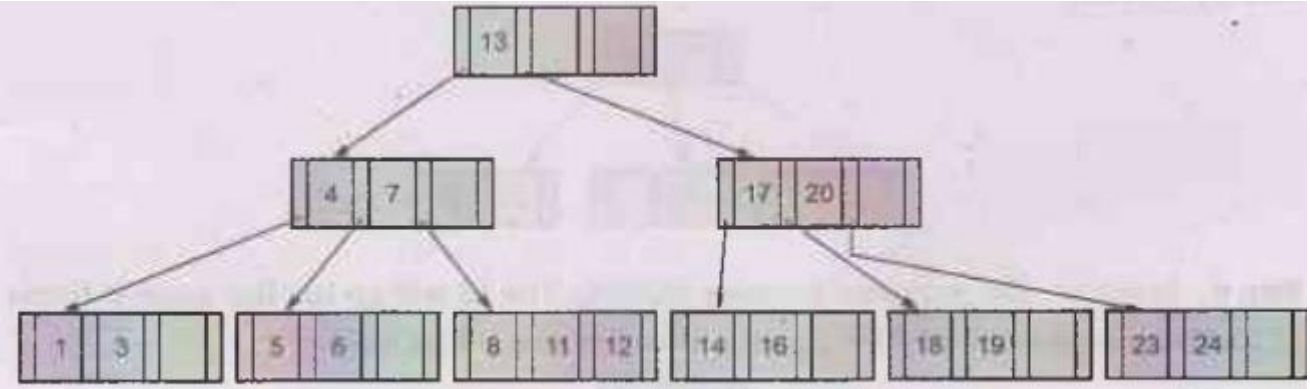
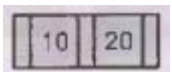


Fig. 4.9.1 : B-Tree

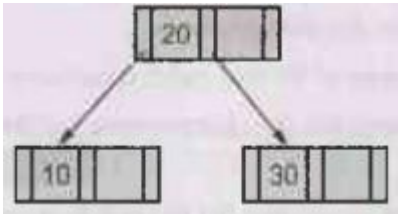
Example 4.9.1 *Create B tree of order 3 for following data: 20,10,30,15,12,40,50. Solution:*

The B tree of order 3 means at the most two key values are allowed in each node of B-Tree.

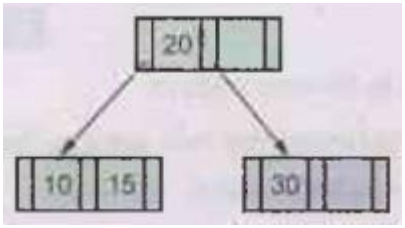
Step 1: Insert 20,10 in ascending order



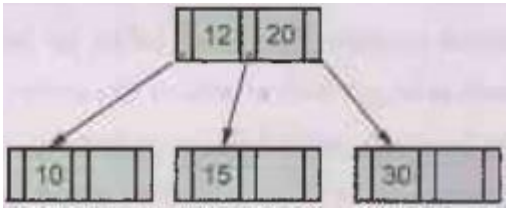
Step 2: If we insert the value 30. The sequence becomes 10,20,30. As only two key values are allowed in each node (being order 3), the 20 will go up.



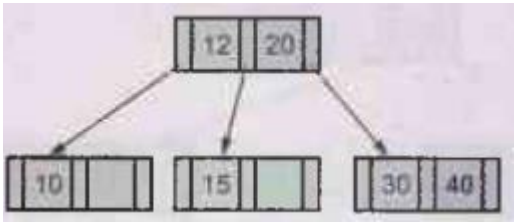
Step 3: Now insert 15.



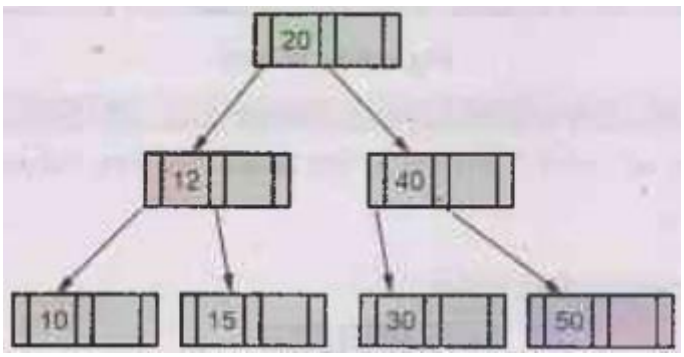
Step 4: Insert 12. The sequence becomes 10, 12, 15. The middle element 12 will go up.



Step 5: Insert 40



Step 6: Insert 50. The sequence becomes 30,40,50. The 40 will go up. But again it forms 12,20,40 sequence and then 20 will go up. Thus the final B Tree will be,



This is the final B-Tree

Difference between B Tree and B+ Tree

B Trees	B+ Trees
In a B-tree you can store both keys and data in the internal and leaf nodes.	In a B+ tree you have to store the data in the leaf nodes only.
It wastes space.	It does not waste space.
The leaf node cannot store using linked list.	The leaf nodes are connected using linked list.
Searching becomes difficult in B-tree as data cannot be found in the leaf node.	Searching of any data in a B+ tree is very easy because all data is found in leaf nodes.
The B-tree does not store redundant search key.	The B-tree stores redundant search key.

Concept of Hashing

AU: Dec.-04,05, May-05,14, Marks 8

- Hash file organization method is the one where data is stored at the data blocks whose address is generated by using hash function.
- The memory location where these records are stored is called as data block or bucket. This bucket is capable of storing one or more records.
- The hash function can use any of the column value to generate the address. Most of the time, hash function uses primary key to generate the hash index - address of the or data block.
- Hash function can be simple mathematical function to any complex mathematical function.
- For example - Following figure represents the records of student can be searched using hash based indexing. In this example the hash function is based on the age field of the record. Here the index is made up of data entry k* which is actual data record. For a hash function the age is converted to binary number format and the last two digits are considered to locate the student record.

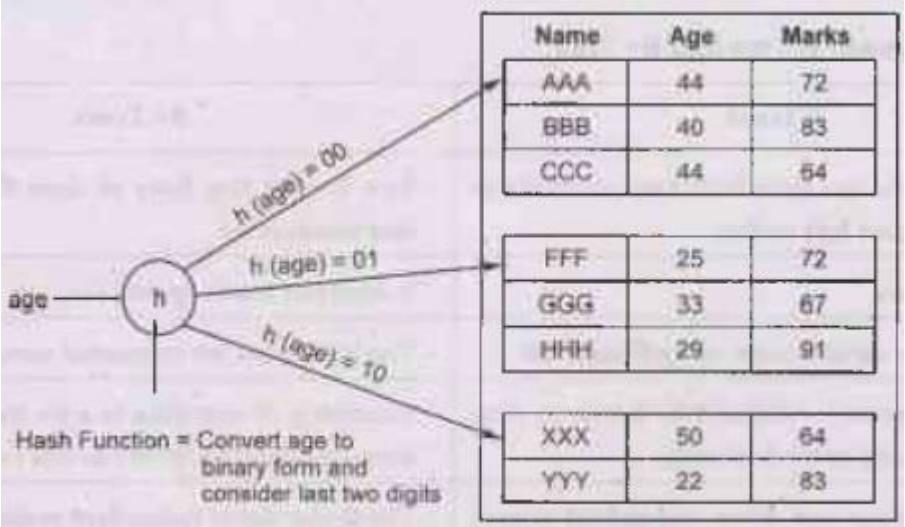


Fig. 4.10.1 : Hash based indexing

Basic Terms used in Hashing

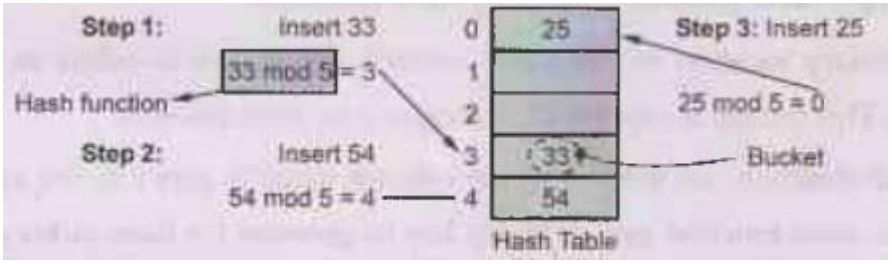
1) Hash Table: Hash table is a data structure used for storing and retrieving data quickly. Every entry in the hash table is made using Hash function.

2) Hash function:

• Hash function is a function used to place data in hash table.

• Similarly hash function is used to retrieve data from hash table.
Thus the use of hash function is to implement hash table

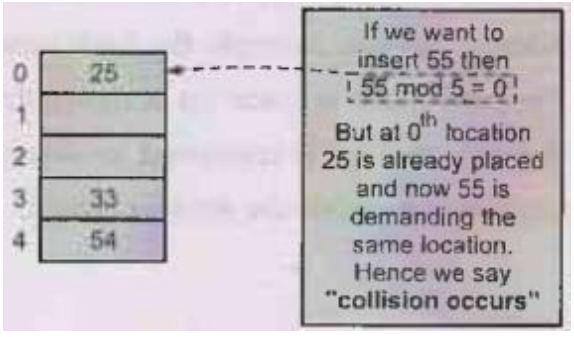
For example: Consider hash function as key



3) Bucket: The hash function H(key) is used to map several dictionary entries in the hash table. Each position of the hash table is called bucket.

4) Collision: Collision is situation in which hash function returns the same address for more than one record.

For example:



1) Probe: Each calculation of an address and test for success is known as a à probe.

2) Synonym: The set of keys that has to the same location are called synonyms. For example - In above given hash table computation 25 and 55 are synonyms.

3) Overflow: When hash table becomes full and new record needs to be inserted then ore it is called overflow.

For example -

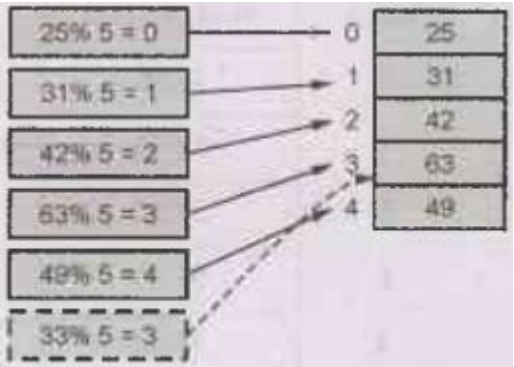


Fig. 4.10.2 : Overflow situation

Static Hashing

• In this method of hashing, the resultant data bucket address will be always same.

Index	Stud_RollNo
0	34780
1	34781
2	34782
3	34783
4	34784
5	34785
6	34786
7	34787
8	34788
9	34789

• That means, if we want to generate address for Stud_RollNo = 34789. Here if we use mod 10 hash function, it always result in the same bucket address 9. There will not be any changes to the bucket address here.

• Hence number of data buckets in the memory for this static hashing remains constant throughout. In our example, we will have ten data buckets in the memory used to store the data.

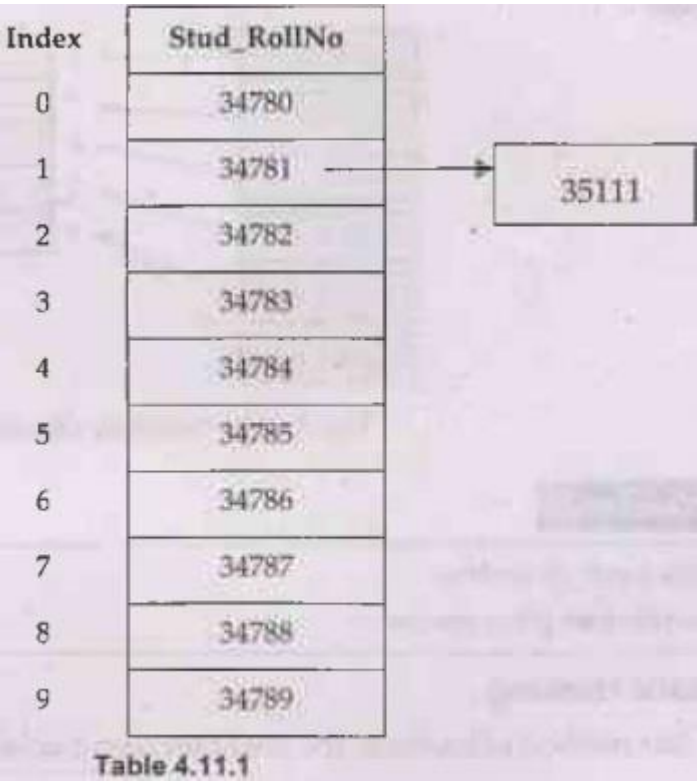


Table 4.11.1

• If there is no space for some data entry then we can allocate new overflow page, put the data record onto that page and add the page to overflow chain of the bucket.

Example 4.11.1 *Why is hash structure not the best choice for a search key on which range of queries are likely ?* **AU: May-06, Marks 8 Solution :**

• A range query cannot be answered efficiently using a hash index, we will have to read all the buckets.

• This is because key values in the range do not occupy consecutive locations in the buckets, they are distributed uniformly and randomly throughout all the buckets.

Advantages of Static Hashing

(1) It is simple to implement.

(2) It allows speedy data storage.

Disadvantages of Static Hashing

There are two major disadvantages of static hashing:

1) In static hashing, there are fixed number of buckets. This will create a problematic situation if the number of records grow or shrink. 2) The ordered access on hash key makes it inefficient.

Open Hashing

The open hashing is a form of static hashing technique. When the collision occurs, that means if the hash key returns the same address which is already allocated by some data record, then

the next available data block is used to enter new record instead of overwriting the old record. This technique is also called as linear probing. For example

Consider insertion of record 105 in the hash table below with the hash function h (key) mod 10.

Index	Stud_RollNo
0	10
1	1
2	22
3	
4	
5	55
6	106
7	
8	88
9	19

The 105 is probed at next empty data block as follows -

Index	Stud_RollNo
0	10
1	1
2	22
3	
4	
5	55
6	106
7	**105**
8	88
9	19

Advantages:

1) It is faster technique.

2) It is simple to implement.

Disadvantages:

1) It forms clustering, as the record is just inserted to next free available slot.

2) If the hash table gets full then the next subsequent records can not be accommodated.

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.
- The most commonly used technique of dynamic hashing is extendible hashing.

Extendible Hashing

The extendible hashing is a dynamic hashing technique in which, if the bucket is overflow, then the number of buckets are doubled and data entries in buckets are re- distributed.

Example of extendible hashing:

In extendible hashing technique the directory of pointers to bucket is used. Refer following Fig. 4.12.1

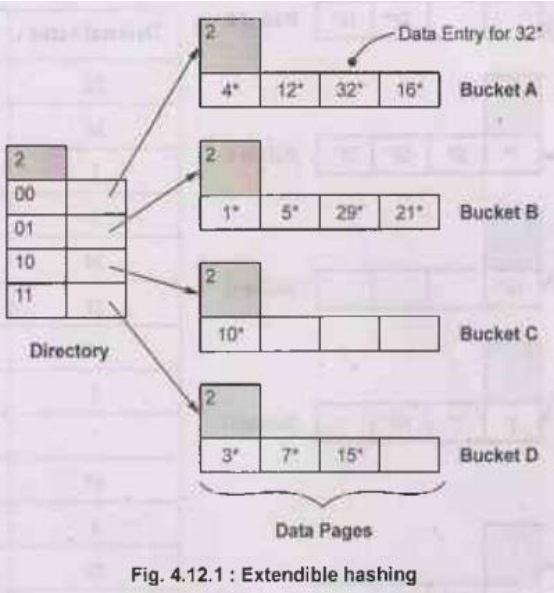
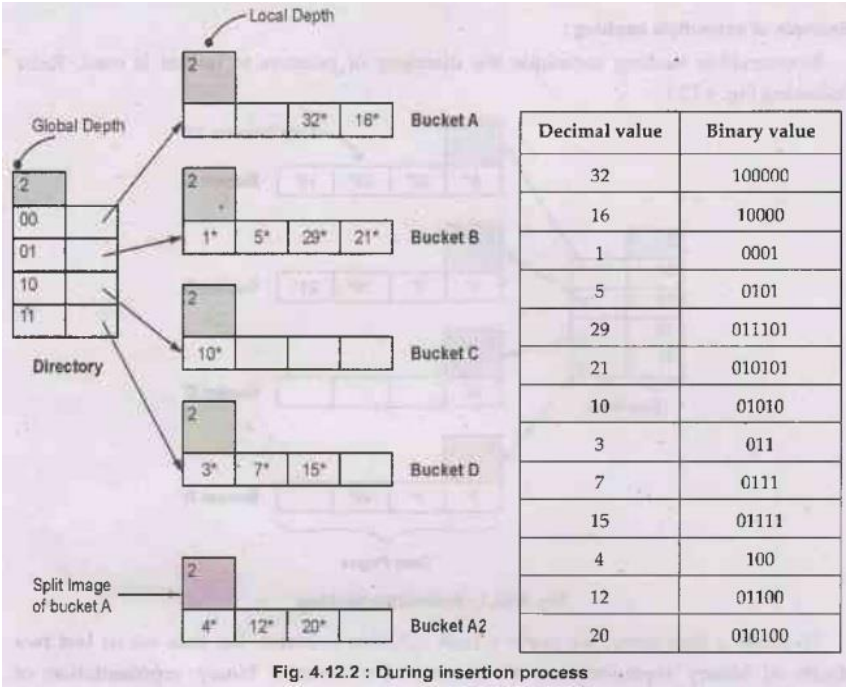


Fig. 4.12.1 : Extendible hashing

To locate a data entry, we apply a hash function to search the data we us last two digits of binary representation of number. For instance binary representation of 32* = 10000000. The last two bits are 00. Hence we store 32* accordingly.

Insertion operation :

- Suppose we want to insert 20* (binary 10100). But with 00, the bucket A is full. So we must split the bucket by allocating new bucket and redistributing the contents, bellsp across the old bucket and its split image.
- For splitting, we consider last three bits of h(r).
- The redistribution while insertion of 20* is as shown in following Fig. 4.12.2.



Decimal value	Binary value
32	100000
16	10000
1	0001
5	0101
29	011101
21	010101
10	01010
3	011
7	0111
15	01111
4	100
12	01100
20	010100

Fig. 4.12.2 : During insertion process

The split image of bucket A i.e. A2 and old bucket A are based on last two bits i.e. 00. Here we need two data pages, to adjacent additional data record. Therefore here it is necessary to double the directory using three bits instead of two bits. Hence, • There will be binary versions for buckets A and A2 as 000 and 100.

- In extendible hashing, last bits d is called global depth for directory and d is called local depth for data pages or buckets. After insetion of 20*, the global depth becomes 3 as we consider last three bits and local depth of A and A2 buckets become 3 as we are considering last three bits for placing the data records. Refer Fig. 4.12.3.

*(**Note:** Student should refer binary values given in Fig. 4.12.2, for understanding insertion operation)*

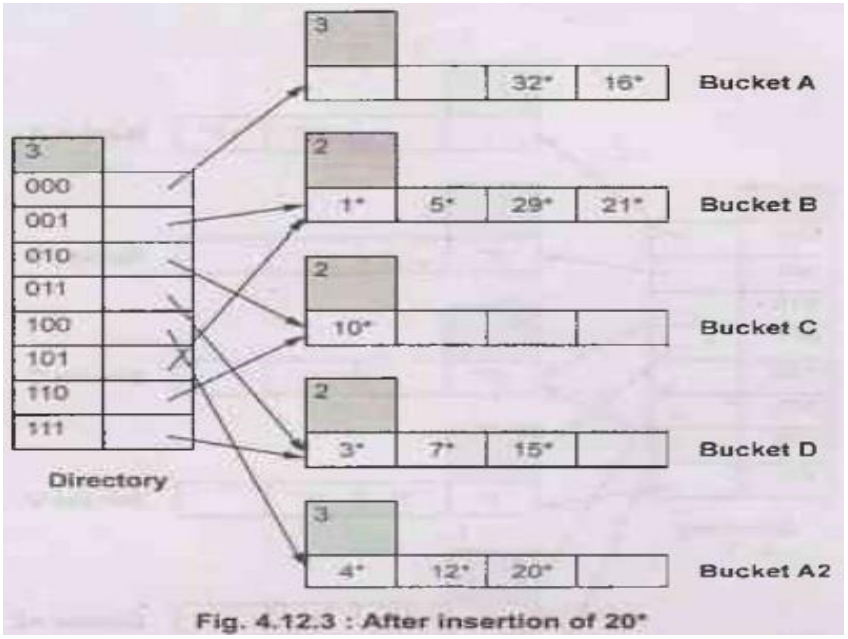


Fig. 4.12.3 : After insertion of 20*

• Suppose if we want to insert 11*, it belongs to bucket B, which is already full. Hence let us split bucket B into old bucket B and split image of B as B2.

• The local depth of B and B2 now becomes 3.

• Now for bucket B, we get and 1=001

11 100011

• For bucket B2, we get

5=101 29 =

11101 and 21

=10101

After insertion of 11* we get the scenario as follows,

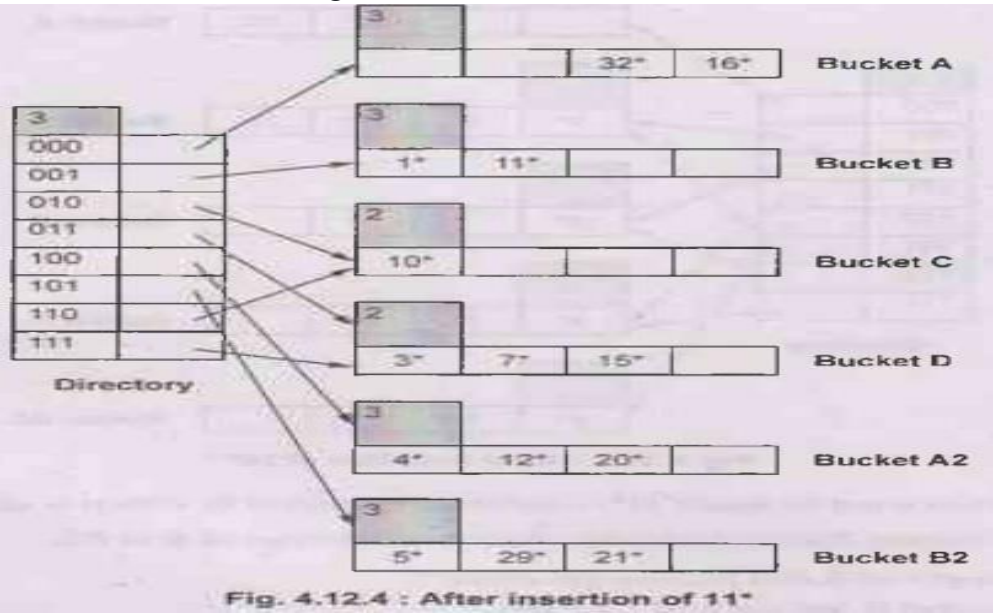


Fig. 4.12.4 : After insertion of 11*

Example 4.12.1 *The following key values are organized in an extendible hashing technique. 13589 12 17 28. Show the extendible hash structure for this file if the hash function is h(x) = x mod 8 and buckets can hold three records. Show how extendable hash structure changes as the result of each of the following steps:*

Insert 2

Insert 24

Delete 5

Delete 12

Solution:

Step 1: Initially we assume the hash function based on last two bits, of result of hash function.

1 mod 8=1=001

3 mod 8=3 = 011

5 mod 8=5= 101

8 mod 8=0=000

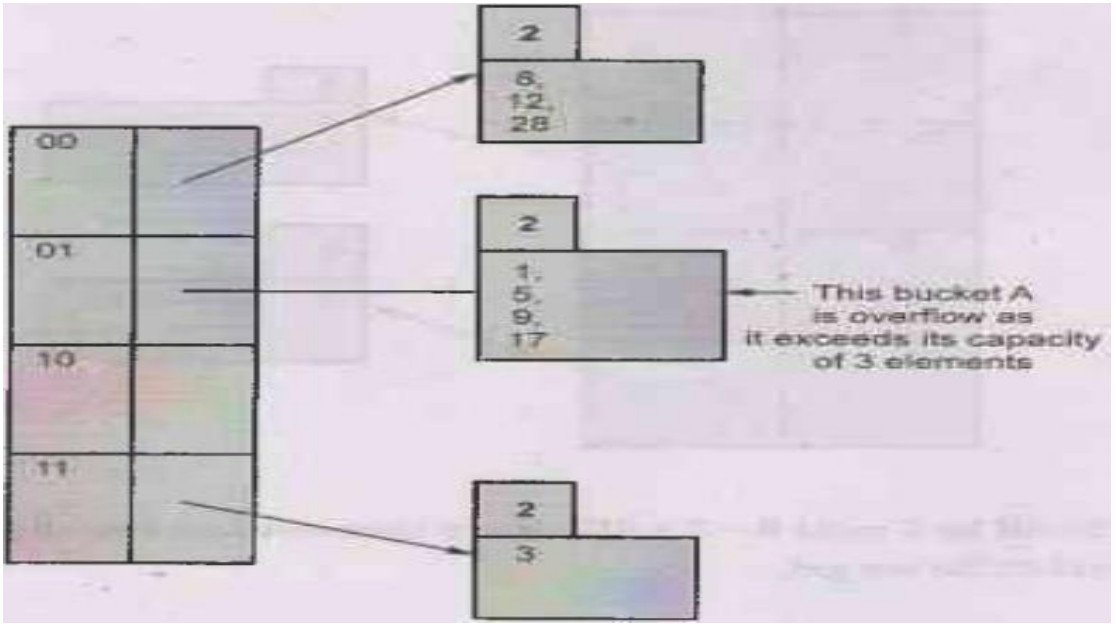
9 mod 8=1=001

12 mod 8 = 4=100

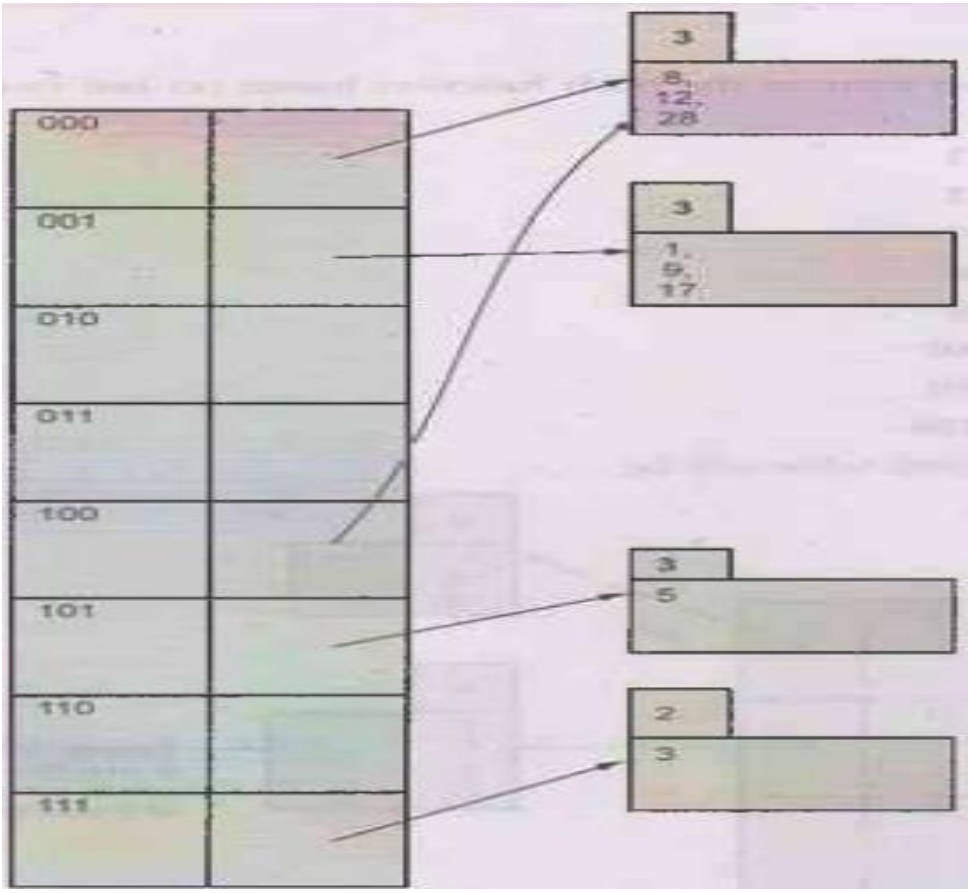
17 mod 8=1=001

28 mod 84 = 100

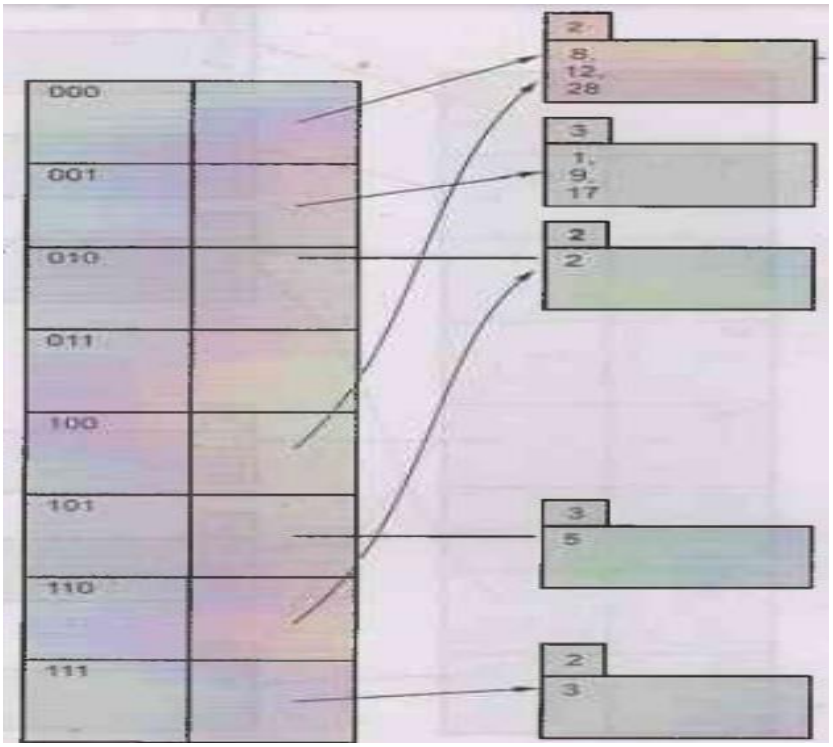
The extendible hash table will be,



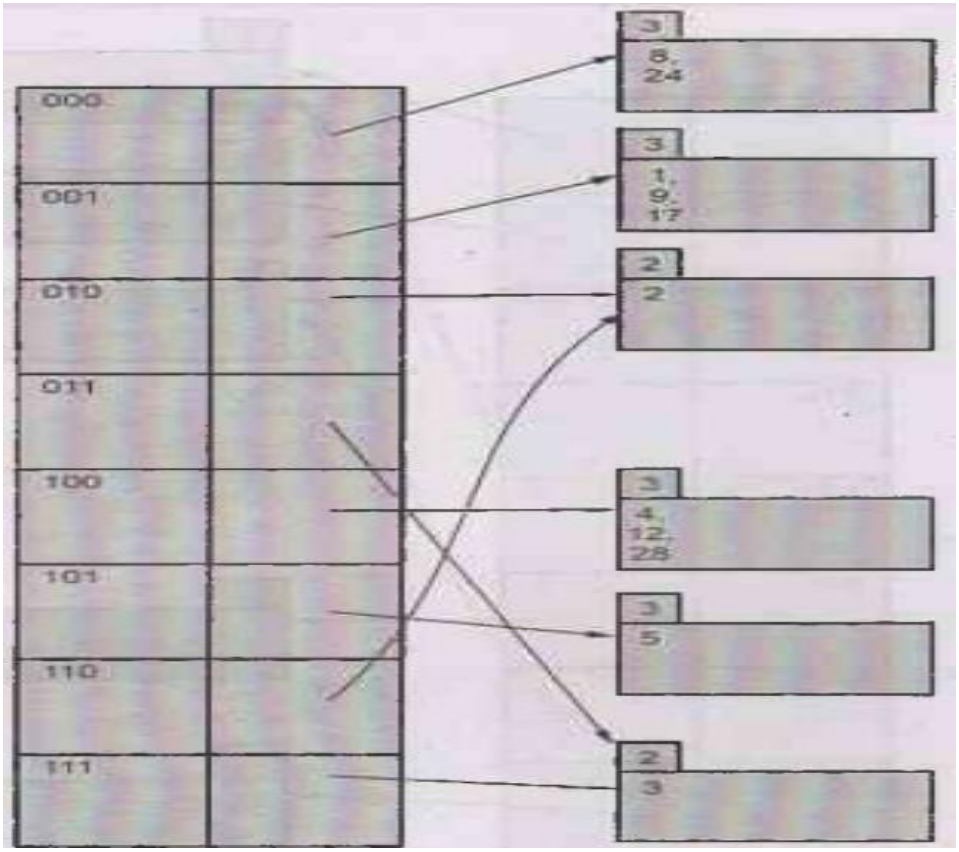
Hence we will extend the table by assuming the bit size as 3. The above indicated bucket A will split based on 001 and 101.



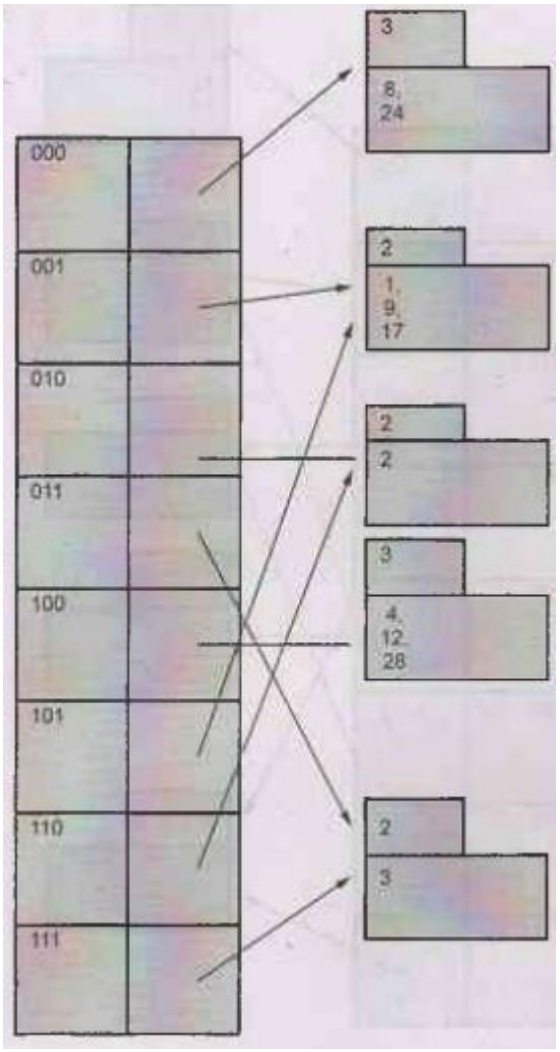
a) **Insert 2** will be 2 mod 8 = 2 = 010. If we consider last two digits i.e. 10 then there is no bucket. So we get,



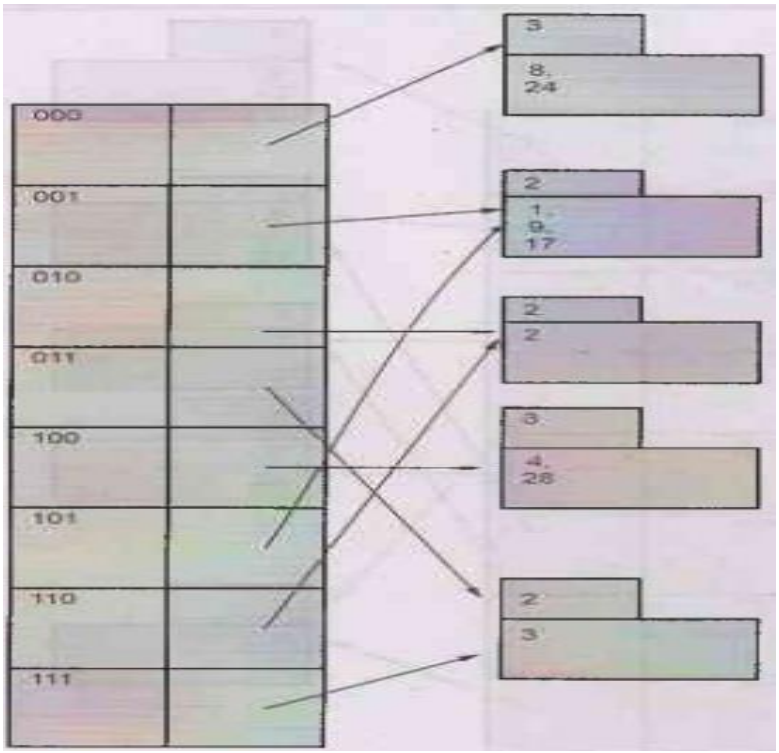
b) Insert 24 : 24 mod 8 = 0 = 000. The bucket in which 24 can be inserted is 8, 12, 28. But as this bucket is full we split it in two buckets based on digits 000 100.



c) Delete 5: On deleting 5, we get one bucket pointed by 101 as empty. This will also result in reducing the local depth of the bucket pointed by 001. Hence we get,



d) Delete 12: We will simply delete 12 from the corresponding bucket there can not be any merging of buckets on deletion. The result of deletion is as given below



Difference between Static and Dynamic Hashing

Sr. No.	Static Hashing	Dynamic Hashing
1.	The number of buckets are fixed.	The number of buckets are not fixed.
2.	Chaining is used	There is no need of chaining.
3.	Open hashing and Closed hashing are forms of static hashing.	Extendible hashing and linear hashing are forms of dynamic hashing.
4.	Space overhead is more.	Minimum space overhead due to dynamic nature.
5.	As file grows the performance of static hash function decreases.	There is no degradation in performance when the file grows.
6.	The bucket address table is not required.	The bucket address table is required.
7.	The bucket is directly accessed.	The bucket address table is used to access the bucket.

Part II: Query Processing

Query Processing Overview

AU: May-14,16,18, Dec.-19, Marks 16

- Query processing is a collection of activities that are involved in extracting data from database.
- During query processing there is translation high level database language queries into the expressions that can be used at the physical level of filesystem.
- There are three basic steps involved in query processing and those are -

1. Parsing and Translation

- In this step the query is translated into its internal form and then into relational algebra.
- Parser checks syntax and verifies relations.
- For instance - If we submit the query as,

SELECT RollNo, name

FROM Student

HAVING RollNo=10

Then it will issue a syntactical error message as the correct query should be

SELECT RollNo, name

FROM Student

HAVING RollNo=10

Thus during this step the syntax of the query is checked so that only correct and verified query can be submitted for further processing.

2. Optimization

- During this process thequery evaluation plan is prepared from all the relational algebraic expressions. bud off
- The query cost for all the evaluation plans is calculated.
- Amongst all equivalent evaluation plans the one with lowest cost is chosen.

• Cost is estimated using statistical information from the database catalog, such asthe number of tuples in each relation, size of tuples, etc.

3. Evaluation

• The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

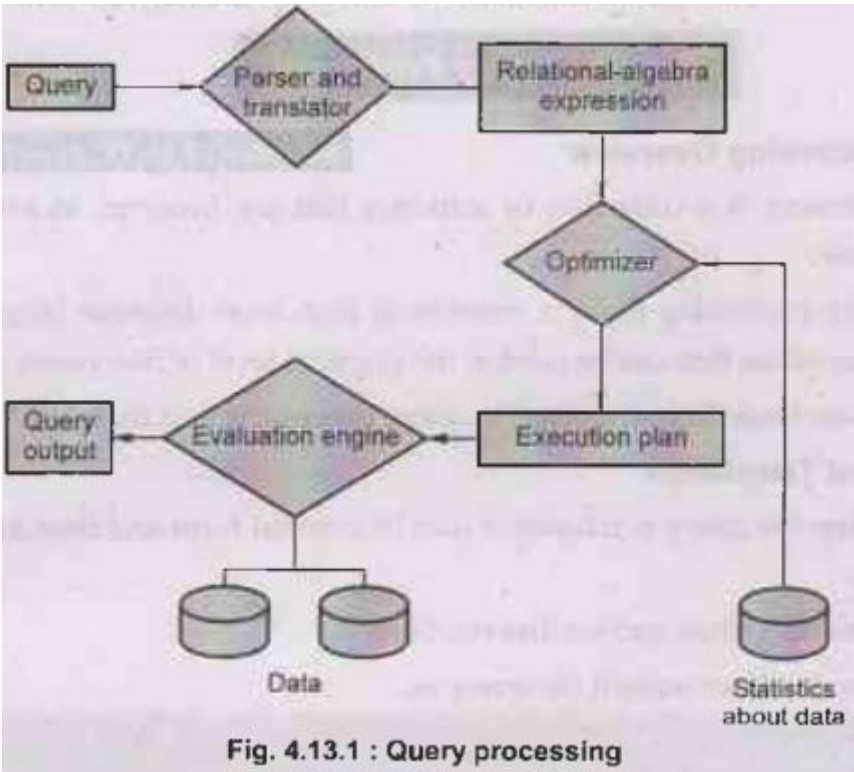


Fig. 4.13.1 : Query processing

For example - If the SQL query is,

SELECT balance

FROM account

WHERE balance<1000

Step 1: This query is first verified by the parser and translator unit for correct syntax. If so then the relational algebra expressions can be obtained. For the above given queries there are two possible relational algebra

(1) $\sigma_{balance<1000}(\Pi_{balance}$ (account))

(2) $\Pi_{balance}$ ($\sigma_{balance<1000}$ (account))

Step 2: Query Evaluation Plan: To specify fully how to evaluate a query, we need not only to provide the relational-algebra expression, but also to annotate it with instructions specifying how to evaluate each operation. For that purpose, using the order of evaluation of queries, two query evaluation plans are prepared. These are as follows

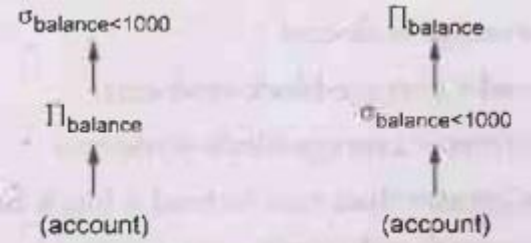


Fig. 4.13.2 : Query evaluation plans

Associated with each query evaluation plan there is a query cost. The query optimization selects the query evaluation plan having minimum query cost.

Once the query plan is chosen, the query is evaluated with that plan and the result of the query is output.

Measure of Query Cost

AU: Dec.-19, Marks 3

- There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their estimated cost and choose the best plan. • There are many factors that contribute to query cost are -
- Disk access
- CPU time to execute the query
- Cost of communication in distributed and parallel database system.
- The cost of access data from disk is an important cost. Normally disk access is relatively slow as compared to in-memory operations. Moreover, the CPU speed is much faster than the disk speed. Hence the time spent in disk is relatively dominant factor in query execution.
- Computing CPU access time is comparatively harder, hence we will not consider the CPU time to execute the query.
- Similarly the cost of communication does not matter for simple large databases ero present in the centralized database system.
- Typically disk access is the predominant cost and is also relatively easy to estimate, taking into account :
- Number of seeks × average-seek-cost
- Number of blocks read × average-block-read-cost
- Number of blocks written × average-block-write-cost
- Cost to write a block is greater than cost to read a block because data is read back after being written to ensure that the write was successful.
- We use number of block transfers from disk and number of disk seeks to estimate the Query cost.
- Let,
- b be the number to blocks
- S be the number of Seeks

- t_T is average time required to transfer a block of data, in seconds
- t_S is average block access time in seconds.

Then query cost can be computed using following formula b*tr+S*ts

Algorithms for Selection, Sorting and join Operations

AU: Dec.-19, Marks 3

Algorithm for Selection Operation

For selection operation, the file scan is an important activity. Basically file scan is a based on searching algorithms. These searching algorithms locate and retrieve the records that fulfills a selection condition.

Let us discuss various algorithms used for SELECT Operation based in file scan

Algorithm A1: Linear Search

• Scan each file block and test all records to see whether they satisfy the selection condition

Cost - b_r block transfers + 1 seek Where, b_r denotes number of

blocks containing records from relation r

• If selection is on a key attribute, can stop on finding record

Cost = (br/2) block transfers + 1 seek

• **Advantages of Linear Search**

- Linear search works even-if there is no selection condition specified.
- For linear search, there is no need to have records in the file in ordered form.
- Linear search works regardless of indices.

Algorithm A2: Binary Search

• Applicable if selection is an equality comparison on the attribute on which file is ordered.

• Assume that the blocks of a relation are stored contiguously.

• Cost estimate is nothing but the number of disk blocks to be scanned.

• Cost of locating the first tuple by a binary search on the blocks = $[\log_2(b_r)]$ x $(t_T + t_S)$ Where,

b_r denotes number of blocks containing records from relation r t_T

is average time required to transfer a block of data, in seconds t_S

is average block access time, in second

• If there are multiple records satisfying the selection add transfer cost of the number of blocks containing records that satisfy selection condition.

Algorithm for Sorting Operation

External sorting

• In external sorting, the data stored on secondary memory is part by part loaded into main memory, sorting can be done over there.

• The sorted data can be then stored in the intermediate files. Finally these intermediate files can be merged repeatedly to get sorted data.

• Thus huge amount of data can be sorted using this technique.

• The external merge sort is a technique in which the data is loaded in intermediate files. Each intermediate file is sorted independently and then combined or merged to get the sorted data.

For example : Consider that there are 10,000 records that has to be sorted. Clearly we need to apply external sorting method. Suppose main memory has a capacity to store 500 records in blocks, with each block size of 100 records

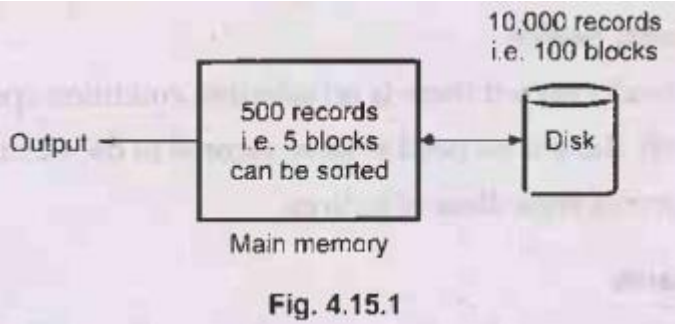


Fig. 4.15.1

The sorted 5 blocks (i.e. 500 records) are stored in intermediate file. This process will be repeated 20 times to get all the records sorted in chunks.

In the second step, we start merging a pair of intermediate files in the main memory to get output file.

Multiway merge

• Multiway merge sort is a technique of merging 'm' sorted lists into single sorted list. The two-way merge is a special case of multiway merge sort.

Let us understand it with the help of an example.

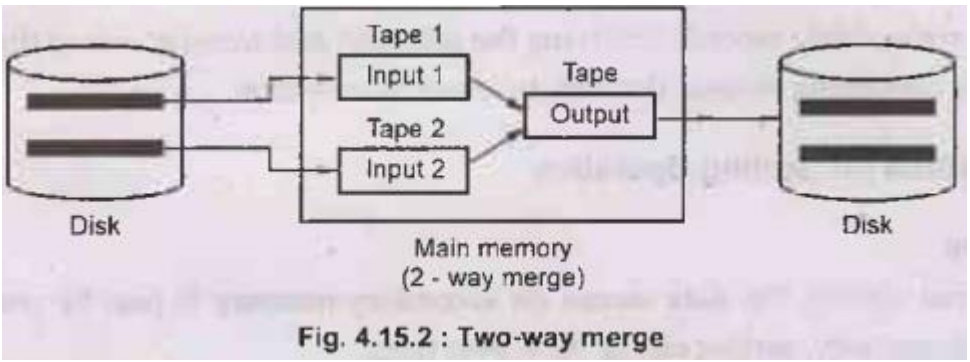


Fig. 4.15.2 : Two-way merge

Example 4.15.1 *Sort the following list of elements using two way merge sort with M = 3. 20, 47, 15, 8, 9, 4, 40, 30, 12, 17, 11, 56, 28, 35.*

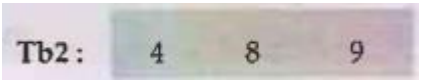
Solution: As M = 3, we will break the records in the group of 3 and sort them. Then we will store them on tape. We will store data on alternate tapes.

Stage I: Sorting phase

1) 20, 47, 15. We arrange in sorted order → 15, 20, 47

2) Read next three records sort them and store them on Tape Tb2

8,9,4→4,8,9



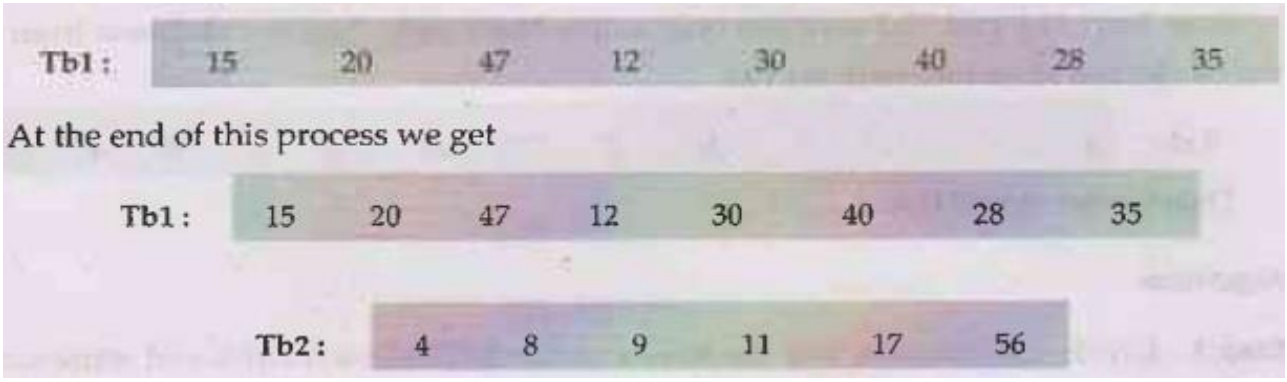
3) Read next three records, sort them and store on tape Tb1.

40,30,12→12,30,40

4) Read next three records, sort them and store on tape Tb2.

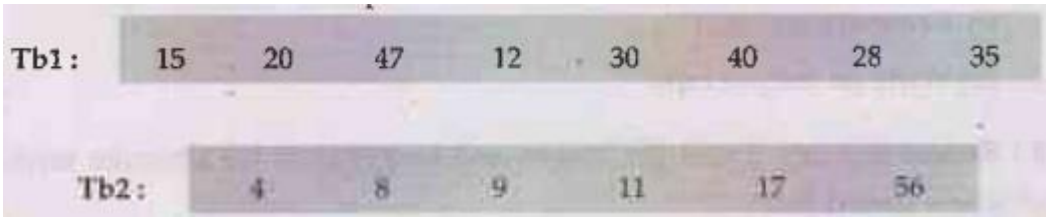
17, 11, 56→11, 17, 56

5) Read next two remaining records, sort them and store on Tape Tb1 28, 35→28, 35

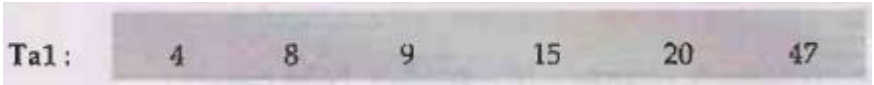


Stage II: Merging of runs

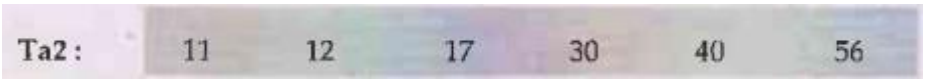
The input tapes Tb1 and Tb2 will use two more output tapes Ta1 and Ta2, for sorting. Finally the sorted data will be on tape Ta1.



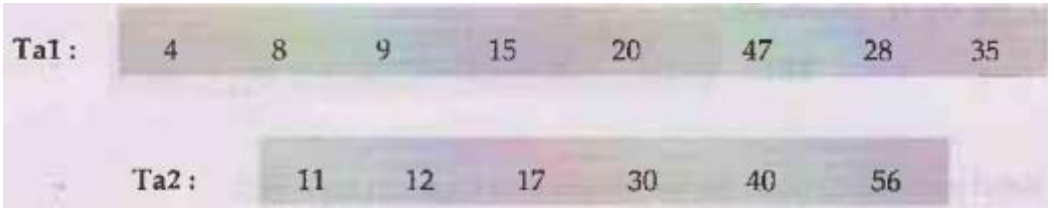
We will read the elements from both the tapes Tb1 and Tb2, compare them and store on Ta1 in sorted order.



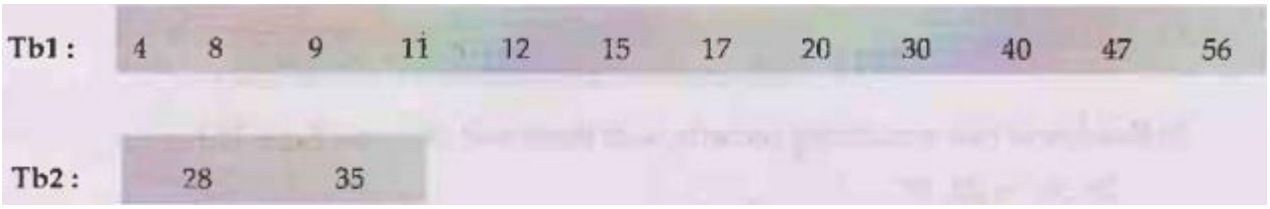
Now we will read second blocks from Tb1 and Tb2. Sort the elements and store on Ta2.



Finally read the third block from Tb1 and store in sorted manner on Tal. We will not compare this block with Ta2 as there is no third block. Hence we will get



Now compare first blocks of Tal and Ta2 and store sorted elements on Tb1.



Now both Tb1 and Tb2 contains only single block each. Sort the elements from both the blocks and store the result on Ta1.

Thus we get sorted list.

Algorithm

Step 1: Divide the elements into the blocks of size M. Sort each block and write runs on disk.

Step 2: Merge two runs

i) Read first value on each of two

runs ii) Compare and sort iii) Write

on output tape.

Step 3: Repeat the step 2 and get longer and longer runs on alternate tapes. Finally we will get single sorted list.

Analysis: The algorithm requires log(N/M) Passes plus initial run construction pass.

• Initially

1st tape contains N records = M records * N/M runs.

After storing the runs on two tapes, each contains half of the runs

Two tapes * M_records per run*1/2 (N/M) = = N records

• After merge 1st pass - double the length of runs, halve the number of runs

Two tapes *2M_records run *1/2*1/2= (N,M) per runs = N records

• After merge 2nd,pass

Two tapes 4 M_records per run *1/2*1/2 *(N,M) runs = N records

• After merge S-th pass

Two tapes * 2^S M records per run *$(1/2^s)$ (1/2) (N/M) runs = N records

• After the last merge, there will be only one run equal to whole file.

2^S M = N

2^S = N/M

S= log (N/M)

Hence at each pass the N records are processed and we get the time complexity as O (N log (N/M)).

• The two way merge sort makes use of two input tapes and two output tapes for sorting the records.

• It works in two stages -

Stage 1: Break the records into block. Sort individual record with the help of two input tapes.

Stage 2: Merge the sorted blocks and create a single sorted file with the help of two output tapes.

Algorithm for Join Operations

• JOIN operation is the most time consuming operation to process.

• There are Several different algorithms to implement joins

1. Nested-loop join
2. Block nested-loop join
3. Indexed nested- loop join
4. Merge-join
5. Hash-join

Choice of a particular algorithm is based on cost estimate.

Algorithm For Nested Loop Join

This algorithm is for computing r ⋈ Θ s

Let, r is called the outer relation and s the inner relation of the join. A

for each tuplet t_r in r do begin for each tuple t_s in s do begin test pair

(t_r,t_s) to see if they satisfy the join condition if 0 is satisfied, then, add

(t_r, t_s) to the result, end end

• This algorithm requires no indices and can be used with any kind of join condition.

• It is expensive since it examines every pair of tuples in the two relations.

• In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

• $n_r \times b_s + b_r$ block transfers, plus $n_r + b_r$ seeks

• If the smaller relation fits entirely in memory, use that as the inner relation

• Reduces cost to $b_r + b_s$ block transfers and 2 seeks

• For example Assume the query CUSTOMERS ⋈ ORDERS (with join attribute only being CName)

Number of records of customer: 10000 order: 5000

Number of blocks of customer: 400 order: 100 **Formula**

Used:

(1) $n_r \times b_s + b_r$ block transfers, (2) $n_r +$

b_r seeks r is outer relation and s is inner

relation.

With order as outer relation:

n_r= 5000 b_s= 400, b_r = 100

5000 × 400 + 100 = 2000100 block transfers and
5000+100 5100 seeks

With customer as the outer relation :

n_r =10000, b_s =100, b_r =400

10000 x 100 + 400 = 1000400 block transfers and

10000+400=10400 seeks

If smaller relation (order) fits entirely in memory, the cost estimate will be:

b_r+b_s =500 block transfers **Algorithm For Block Nested Loop Join**

Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

This algorithm is for computing r ⋈Θ s

Let, r is called the outer relation and s the inner relation of the join.

for each block B_r of r do for each block B_s of s do

for each tuple t_r in B_r do begin for each tuple t_s in B_s

do begin test pair (t_r, t_s) to see if they satisfy the join

condition if Θ is satisfied, then, add (t_r, t_s) to the

result.

end

end

end

end

Worst case estimate: $b_r \times b_s + b_r$ block transfers + 2 × b_r seeks

Each block in the inner relation s is read once for each block in the outer relation. Best case:

b_r+ b_s block transfers + 2 seeks

(3) Merge Join

• In this operation, both the relations are sorted on their join attributes. Then merged these sorted relations to join them.

• Only equijoin and natural joins are used. • The cost of merge join is, $b_r + b_s$ block transfers + $[b_r / b_b]$ + $[b_s/b_b]$ seeks + the cost of sorting, if relations are unsorted.

(4) Hash Join

• In this operation, the hash function h is used to partition tuples of both the relations.

• h maps A values to {0, 1, ..., n}, where A denotes the attributes of r and s used in the join.

• Cost of hash join is:
$3(b_r+b_s)$ + 4 x n block transfers + $2([b_r / b_b] + [b_s/b_b])$ seeks

If the entire build input can be kept in main memory no partitioning is required Cost estimate goes down to br+bs.

Query Optimization using Heuristics - Cost Estimation

AU: Dec,-13,16, May-15, Marks 16 Heuristic

Estimation

• Heuristic is a rule that leads to least cost in most of cases.

• Systems may use heuristics to reduce the number of choices that must be made in a cost-based fashion.

• Heuristic optimization transforms the query-tree by using a set of rules that typically t improve execution performance. These rules are 1. Perform selection early (reduces the number of tuples)

2. Perform projection early (reduces the number of attributes)

3. Perform most restrictive selection and join operations before other similar on she is ont ni bold do operations (such as cartesian product).

• Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Steps in Heuristic Estimation

Step 1: Scanner and parser generate initial query representation **Step 2:** Representation is optimized according to heuristic rules

Step 3: Query execution plan is developed

For example: Suppose there are two relational algebra -

(1) $\sigma_{city= \text{"Pune"}}$ (T_{cname} Branch) ⋈ Account ⋈ Customer)

(2) $\Pi_{cname}(\sigma_{city=\text{"Pune}}$ (Branch ⋈ Account ⋈ Customer))

The query evaluation plan can be drawn using the query trees as follows-

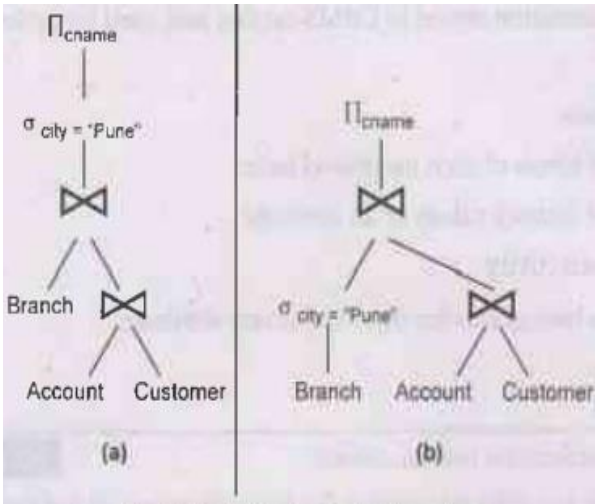


Fig. 4.16.1 : Query evaluation plan

Out of the above given query evaluation plans, the Fig. 4.16.1 (b) is much faster than Fig. 4.16.1 (a) because - in Fig. 4.16.1 (a) the join operation is among Branch, Account and Customer, whereas in Fig. 4.16.1 (b) the join of (Account and Customer) is made with the selected tuple for City="Pune". Thus the output of entire table for join operation is much more than the join for some selected tuples. Thus we get choose the optimized query.

Cost based Estimation

- A cost based optimizer will look at all of the possible ways or scenarios in which a query can be executed.
- Each scenario will be assigned a 'cost', which indicates how efficiently that query can be run.
- Then, the cost based optimizer will pick the scenario that has the least cost and execute the query using that scenario, because that is the most efficient way to run the query.
- Scope of query optimization is a query block. Global query optimization involves multiple query blocks.
- Cost components for query execution
- Access cost to secondary storage
- Disk storage cost
- Computation cost
- Memory usage cost
- Communication cost
- Following information stored in DBMS catalog and used by optimizer
- File size
- Organization
- Number of levels of each multilevel index
- Number of distinct values of an attribute
- Attribute selectivity
- RDBMS stores histograms for most important attributes