

### 4.3 Organization of Records in File

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.

---

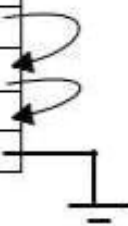
TECHNICAL PUBLICATIONS™ - An up thrust for knowledge

- (3) **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.

### 4.3.1 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

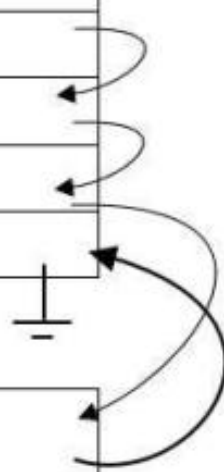
4	EEE
---	-----

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

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

4	EEE	
---	-----	--



**Fig. 4.3.1**

- However, maintaining physical sequential order is very difficult as there can be several insertions and deletions.
- 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.

#### 4.3.2 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
Prajakta	91

Sname	Cname	City
Ankita	ComputerSci	Chennai
Prajakta	Electronics	Pune

The multitable clustering organization for above tables is,

Ankita	ComputerSci	Chennai
Ankita	55	
Ankita	67	
Ankita	86	
Prajakta	Electronics	Pune
Prajakta	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

---

Ankita	ComputeSci	Chennai	
Ankita	55		
Ankita	67		
Ankita	86		
Prajakta	Electronics	Pune	
Prajakta	91		



2 (

## 4.6 B+ Tree Index Files

AU : May-03, 06,16, 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**.

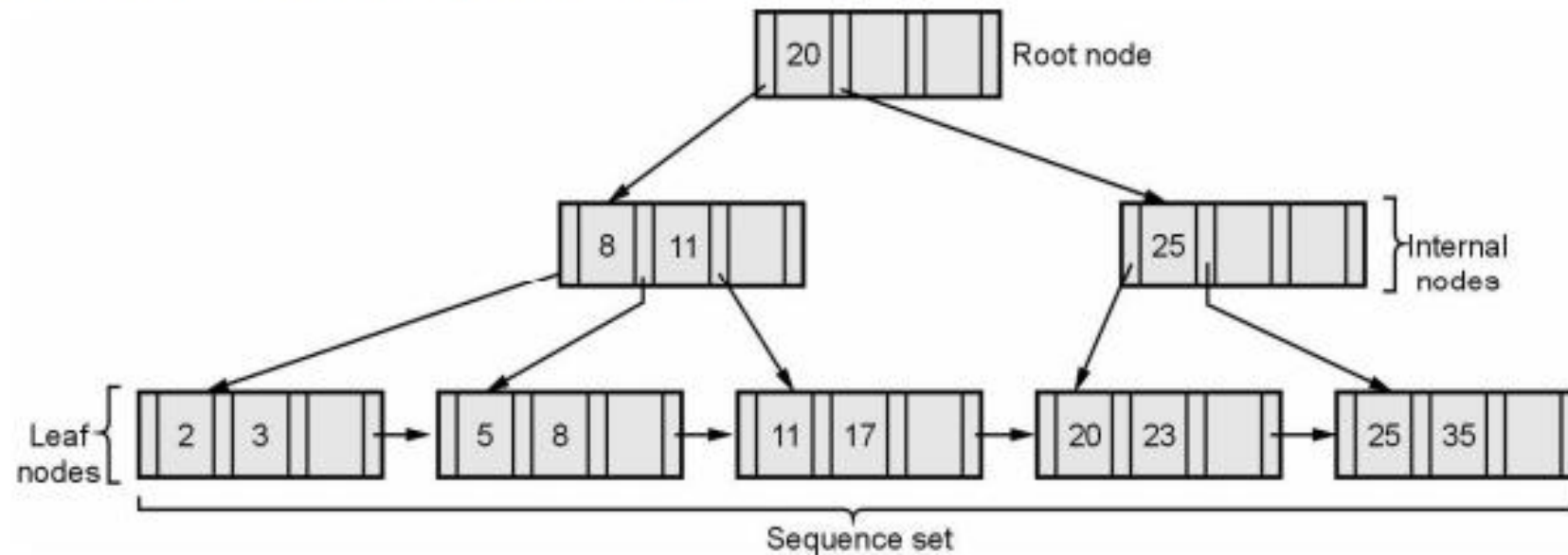
### Structure of B+ Tree

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

$P_1$	$K_1$	$P_2$	$K_2$	...	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	-------	-----	-----------	-----------	-------

- It contains up to  $n - 1$  search-key values  $K_1, K_2, \dots, K_{n-1}$ , and  $n$  pointers  $P_1, P_2, \dots, P_n$ .
  - 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. 4.6.1 represents the example of B+ tree.



**Fig. 4.6.1 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.



#### 4.6.1 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.

---

TECHNICAL PUBLICATIONS™ - An up thrust for knowledge

**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.



## 4.1 RAID

AU : May-07,15,16,17, Dec.-06,13,14,15,16, Marks 16

- 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.



### 4.1.1 RAID Levels

#### Level : RAID 0

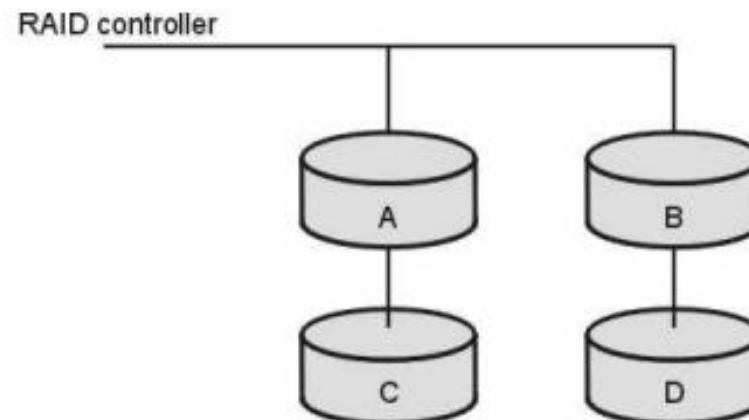
- In this level, data is broken down into blocks and these blocks are stored across all the disks.
- Thus **striped array of disks** is implemented in this level. For instance in the following figure blocks “A B” form a stripe.

---

TECHNICAL PUBLICATIONS™ - An up thrust for knowledge

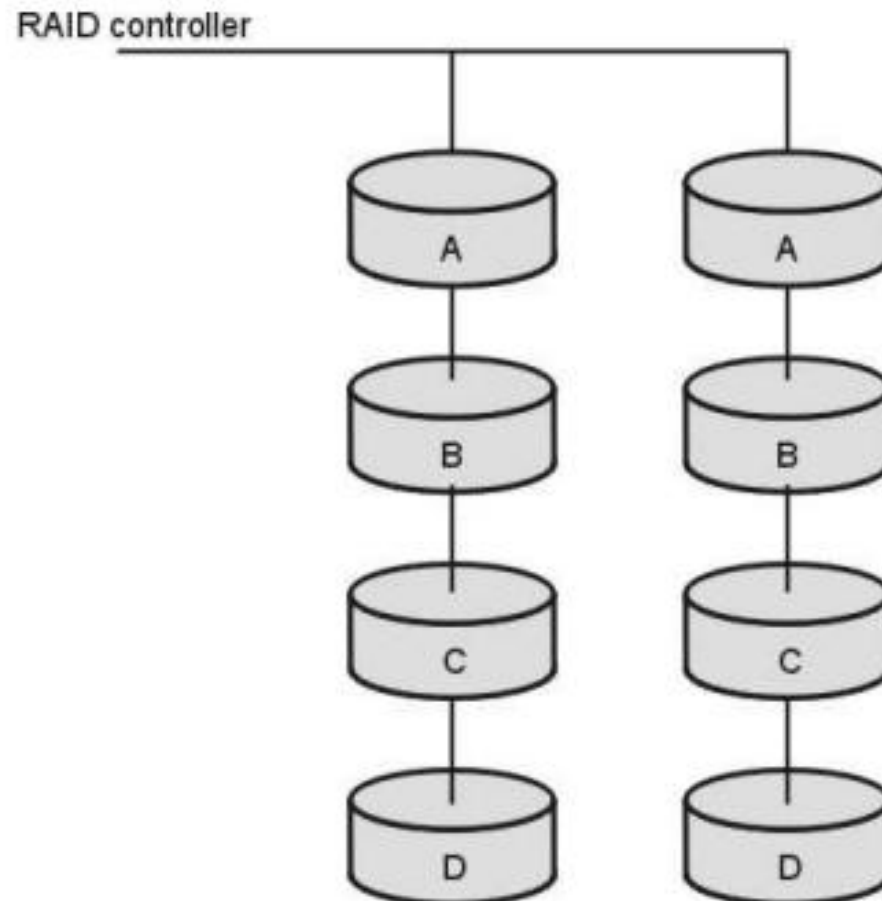
---

- There is no duplication of data in this level so once a block is lost then there is no way recover it.
- The main **priority of this level is performance and not the reliability.**



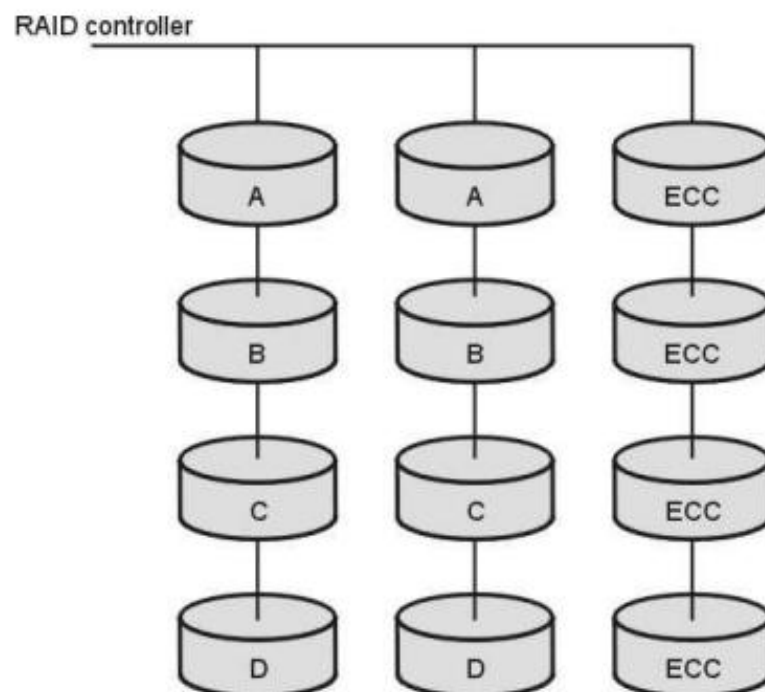
## Level : RAID 1

- 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

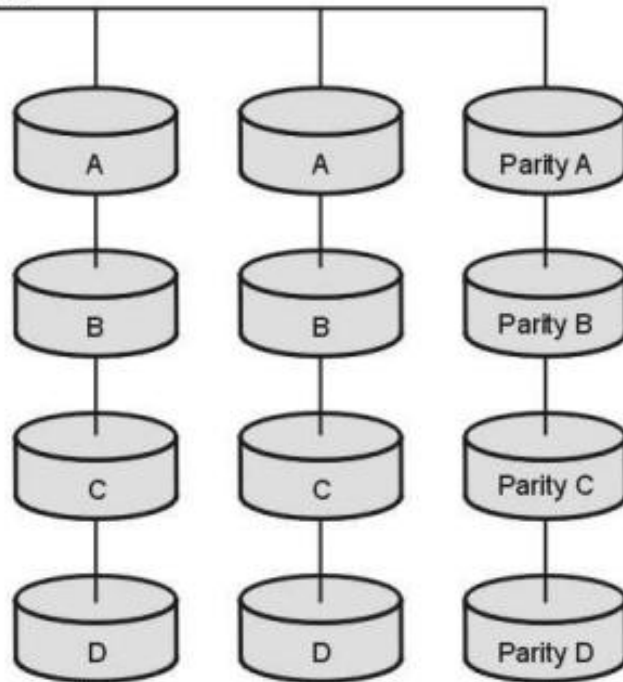
- This level makes use of **mirroring as well as stores Error Correcting Codes (ECC)** for its data striped on different disks.
- The data is stored in separate set of disks and ECC is stored another set of disks.
- This level has a complex structure and high cost. Hence it is not used commercially.



### 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.

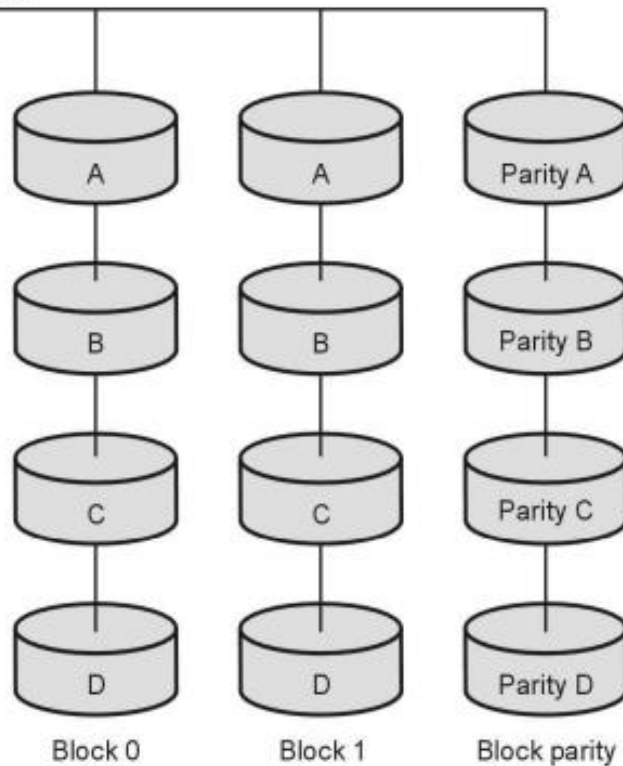
RAID controller



#### Level : RAID 4

- RAID 4 consists of block-level stripping with a parity disk.
- Note that level 3 uses byte-level stripping, whereas level 4 uses block-level stripping.

RAID controller

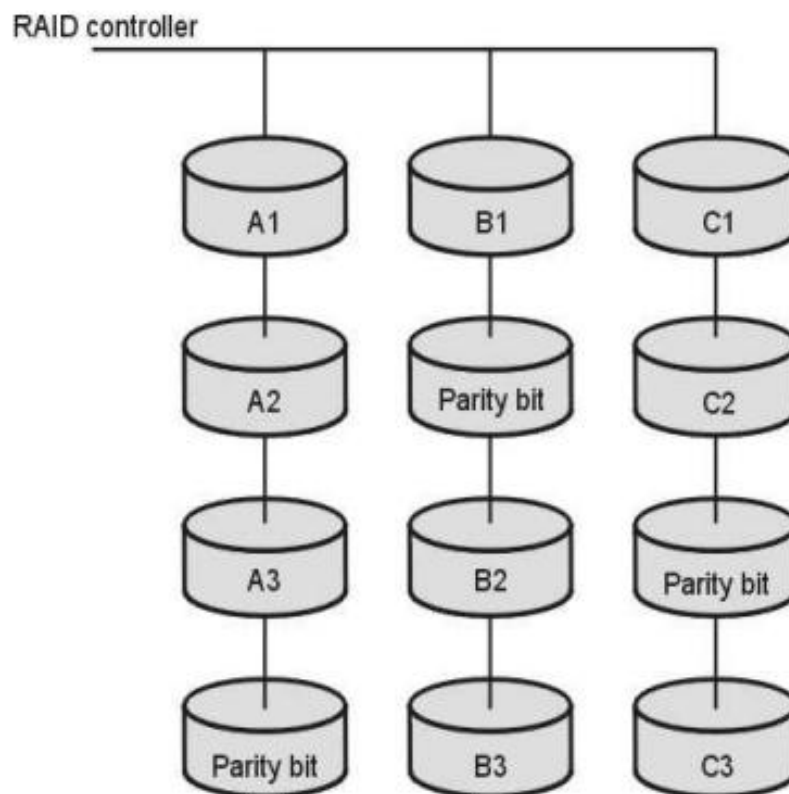




---

### Level : RAID 5

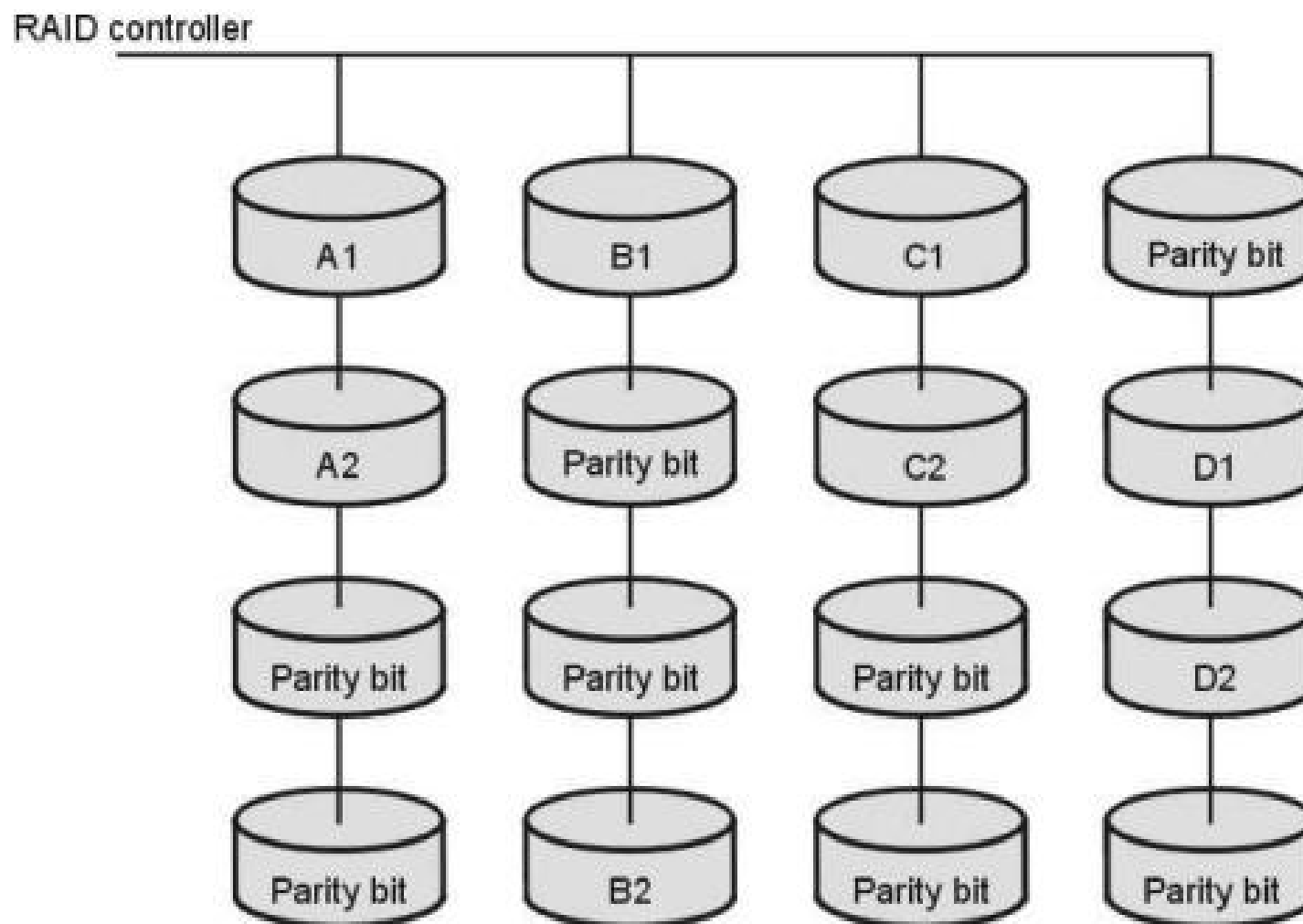
- RAID 5 is a modification of RAID 4.
- RAID 5 writes whole data blocks onto different disks, but the **parity bits** generated for data block stripe are **distributed among all the data disks** rather than storing them on a different dedicated disk.



### RAID : Level 6

- RAID 6 is an 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



## **4.15 Query Optimization using Heuristics and Cost Estimation**

**AU : Dec. -13, 16, May-15, Marks 16**

### **4.15.1 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 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 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

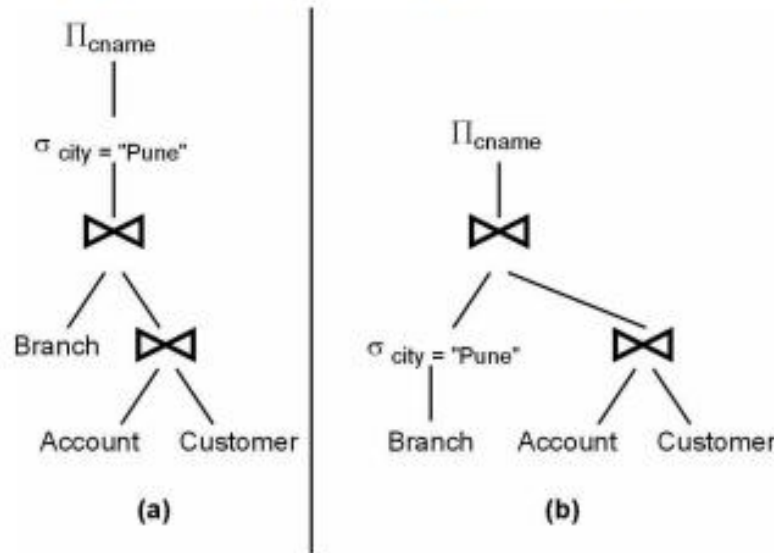
---

**For example :** Suppose there are two relational algebra -

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

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

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



**Fig. 4.15.1 Query evaluation plan**

Out of the above given query evaluation plans, the Fig. 4.15.1 (b) is much faster than Fig. 4.15.1 (a) because – in Fig. 4.15.1 (a) the join operation is among Branch, Account and Customer, whereas in Fig. 4.15.1 (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.

#### **4.15.2 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



## 4.5 Ordered Indices

AU : Dec.-04, 08, 15, May-06, Marks 10

### 4.5.1 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.

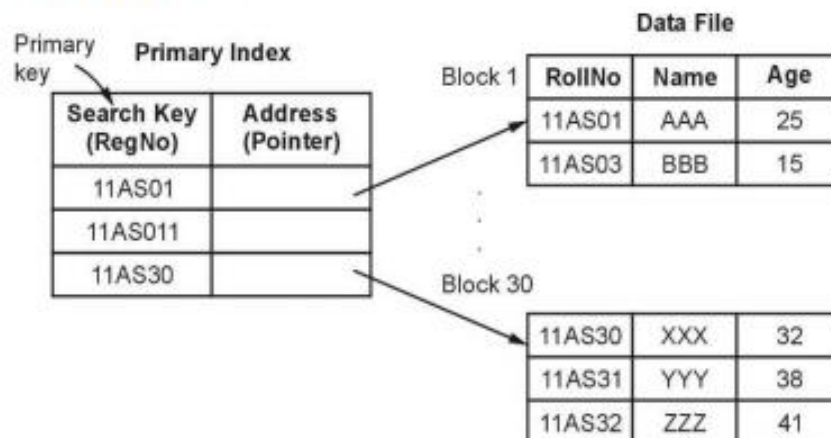


Fig. 4.5.1 : Example of primary index

- 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 RegNo 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 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  $\log_2 n + 1 = (\log_2 3) + 1 \approx 3$



- 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.; 1<sup>st</sup> semester students, 2<sup>nd</sup> semester students, 3<sup>rd</sup> semester students etc. are grouped.

	RollNo	Name	Marks	City	Age
	100				
	101				
	102				
	...	...	...	...	...
	200				
	201				
	202				
	...	...	...	...	...
	240				
	241				
	...	...	...	...	...
	300				

Semester	Address
1	
2	
3	
4	

Fig. 4.5.2 Clustered Index

#### 4.5.2 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.

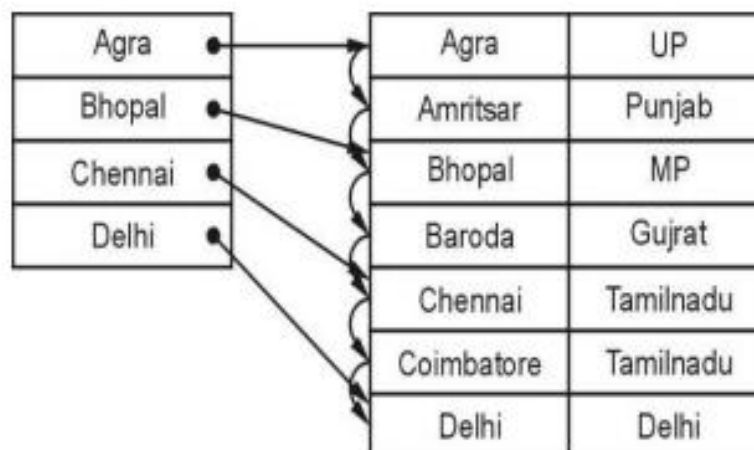
- For example :



**Fig. 4.5.3 : 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. 4.5.4 : Sparse index**

### **4.5.3** Single and Multilevel Indices

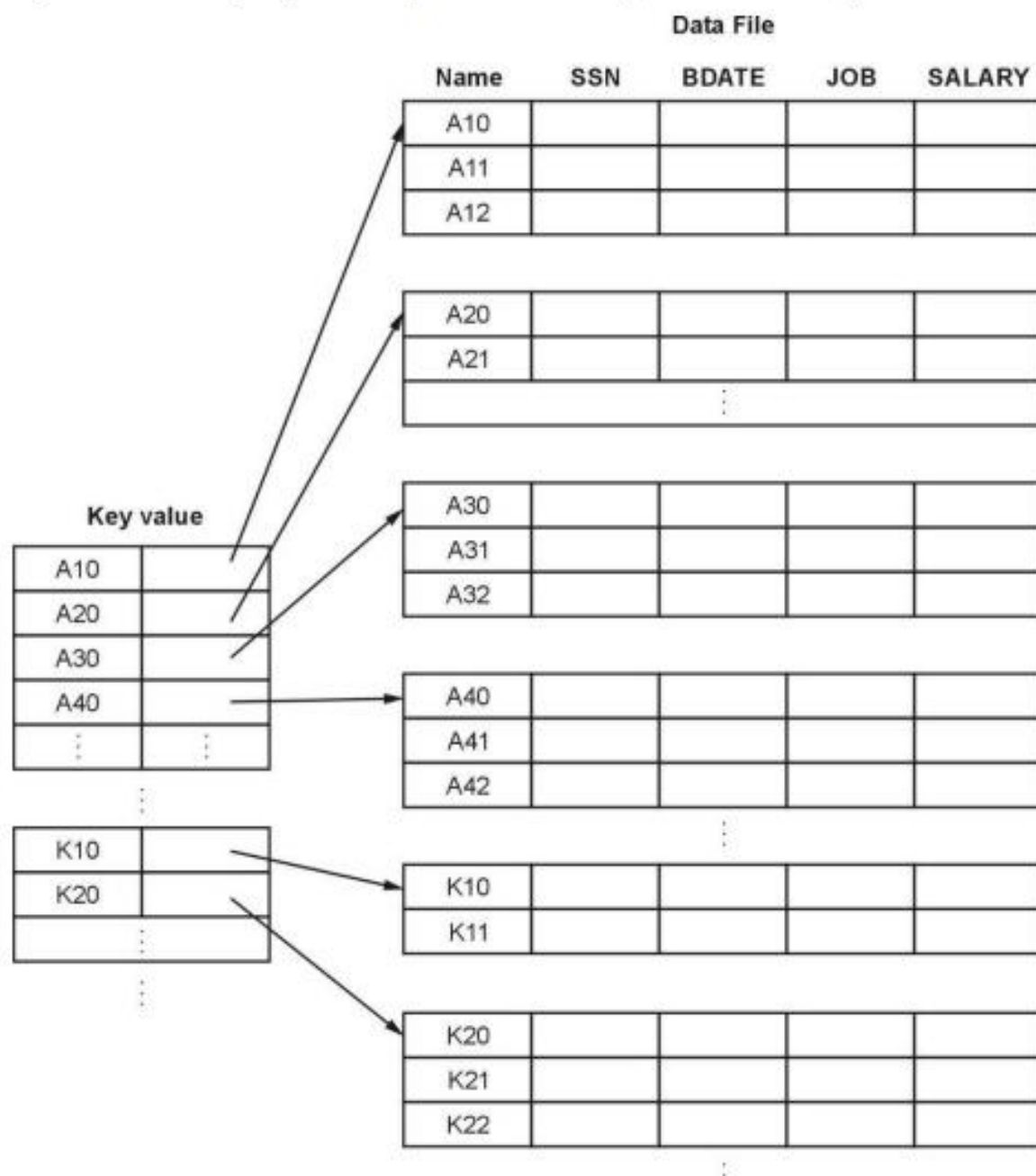
#### Single level indexing :

- 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
------------	-------------------

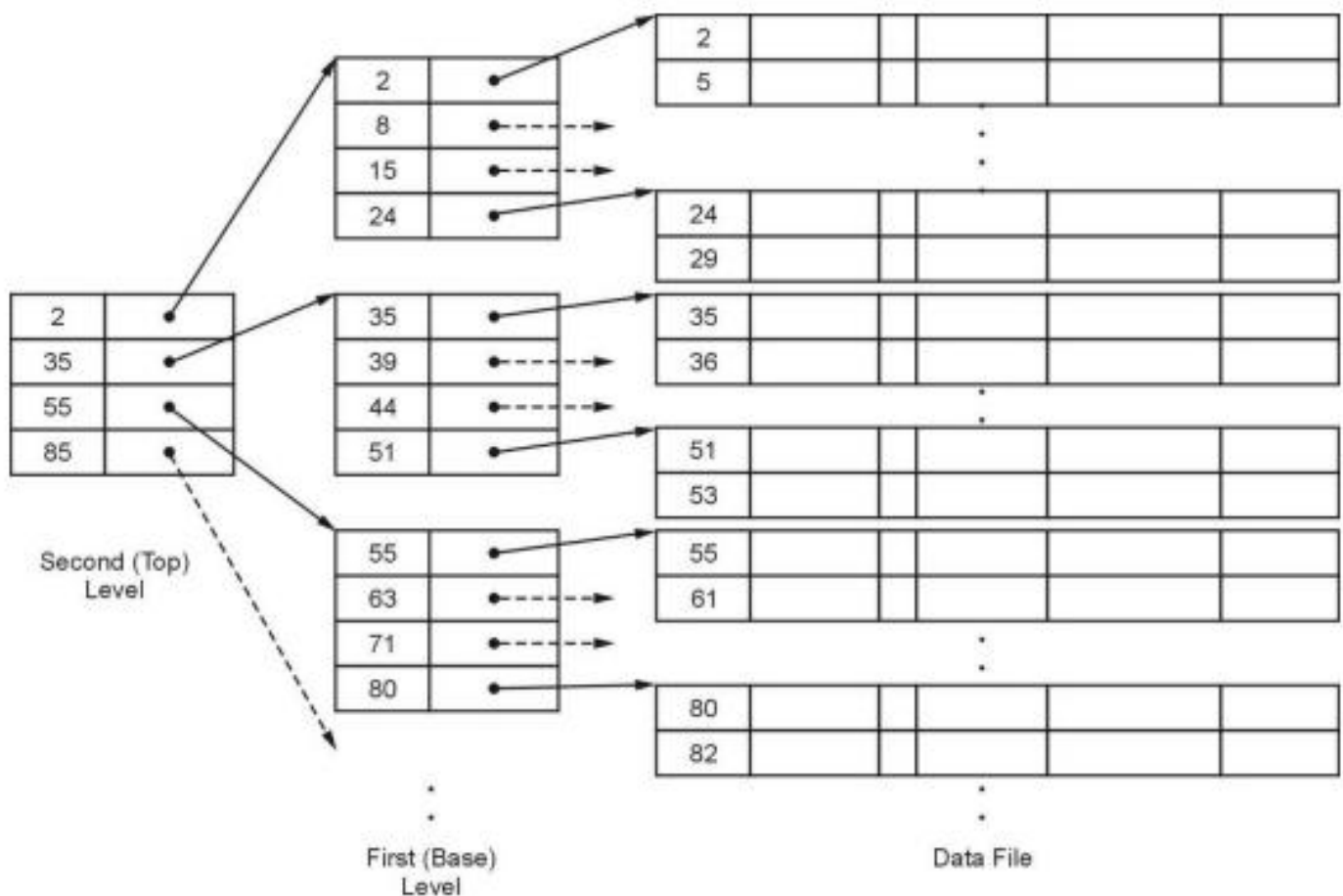
- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller.
- A binary search on the index yields a pointer to the file record.
- The types of single level indexing can be primary indexing, clustering index or secondary indexing.
- Example : Following Fig. 4.5.5 represents the 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. 4.5.6



**Fig. 4.5.6 Multilevel indexing**

#### 4.5.4 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.

TECHNICAL PUBLICATIONS™ - An up thrust for knowledge

- For example -

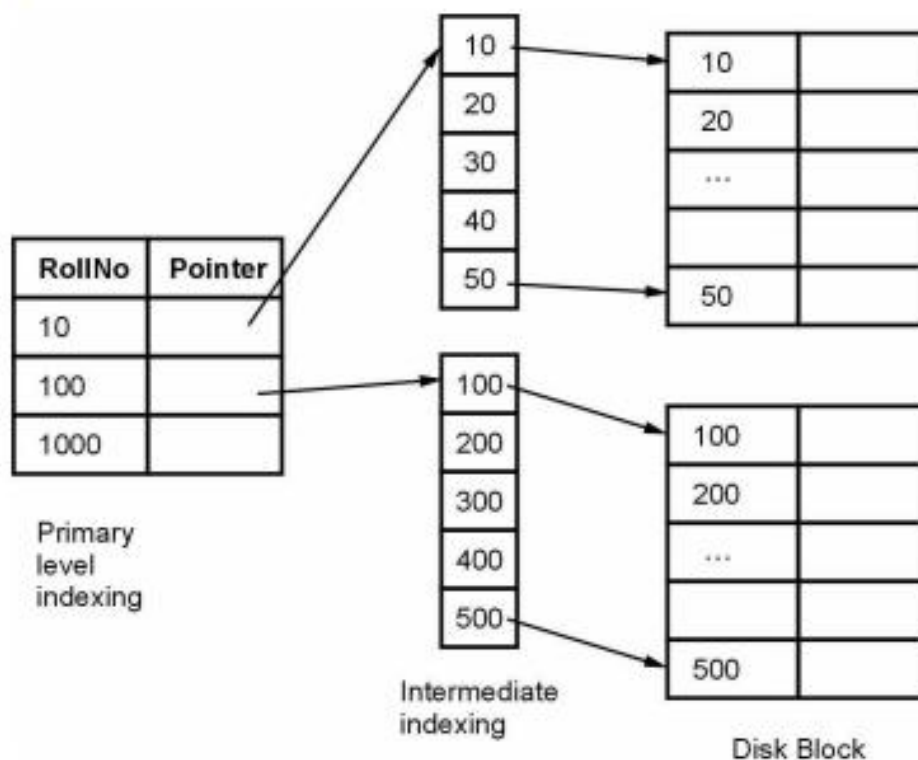


Fig. 4.5.7 Secondary Indexing





## 4.11 Query Processing Overview

AU : May-14, 16, 18, 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 :

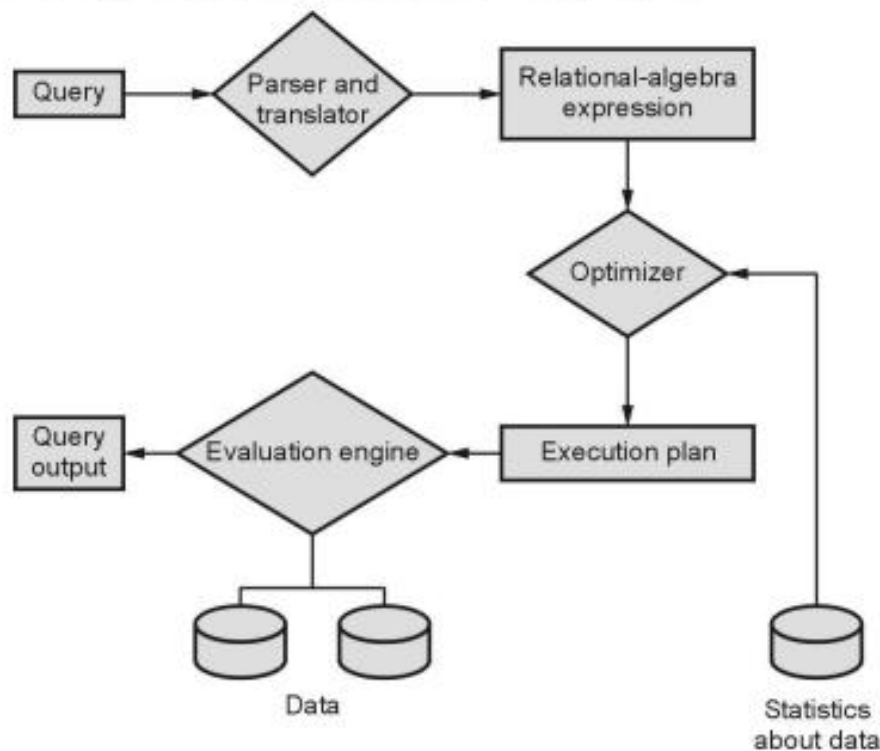
- During this process the query evaluation plan is prepared from all the relational algebraic expressions.
- The query cost for all the evaluation plans is calculated.
- Amongst all equivalent evaluation plans the one with lowest cost is chosen.

- Cost is estimated using statistical information from the database catalog, such as the 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.

The above describe steps are represented by following Fig. 4.11.1 –



**Fig. 4.11.1 Query processing**

For example - If the SQL query is,

```

SELECT balance
FROM account
WHERE balance < 1000
  
```

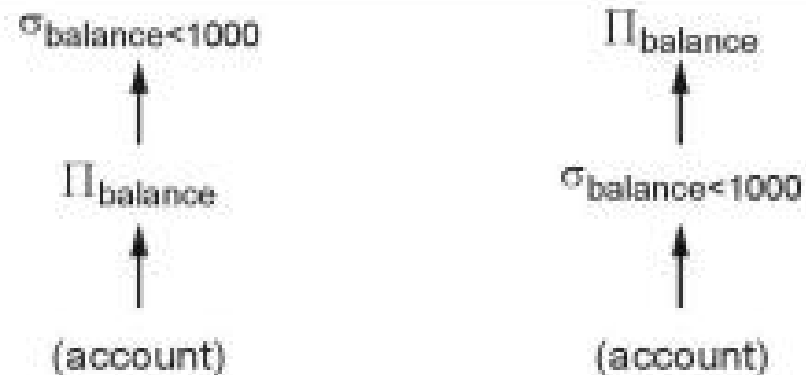
**Step 1 :** This query is first verified by the **parser and translator** unit for correct syntax. If so then the relational algebra expressions can be obtained. For the above given queries there are two possible relational algebra

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

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

**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. 4.11.2 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**.

## 4.10 Dynamic Hashing

AU : May-04,07,18, Dec.-08,17, Marks 13

- 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.

### 4.10.1 Extendible Hashing

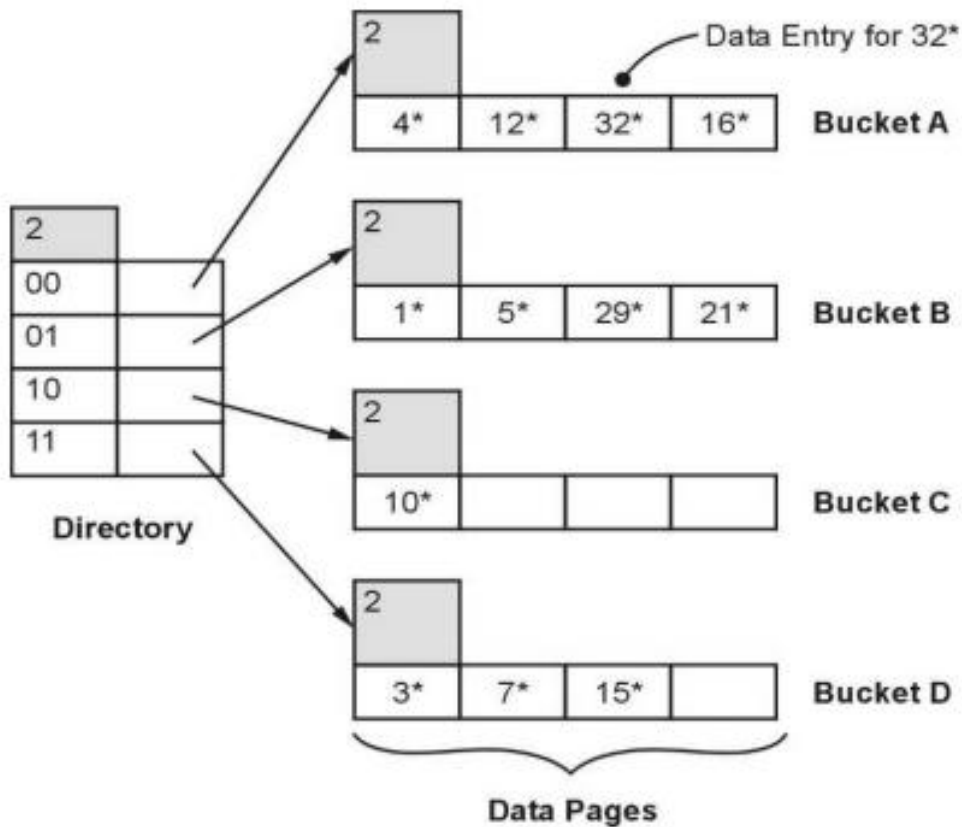
The extendible hashing is a dynamic hashing technique in which, if the bucket is overflow, then the number of buckets are doubled and data entries in buckets are re-distributed.



---

### Example of extendible hashing :

In extendible hashing technique the **directory** of pointers to bucket is used. Refer following Fig. 4.10.1



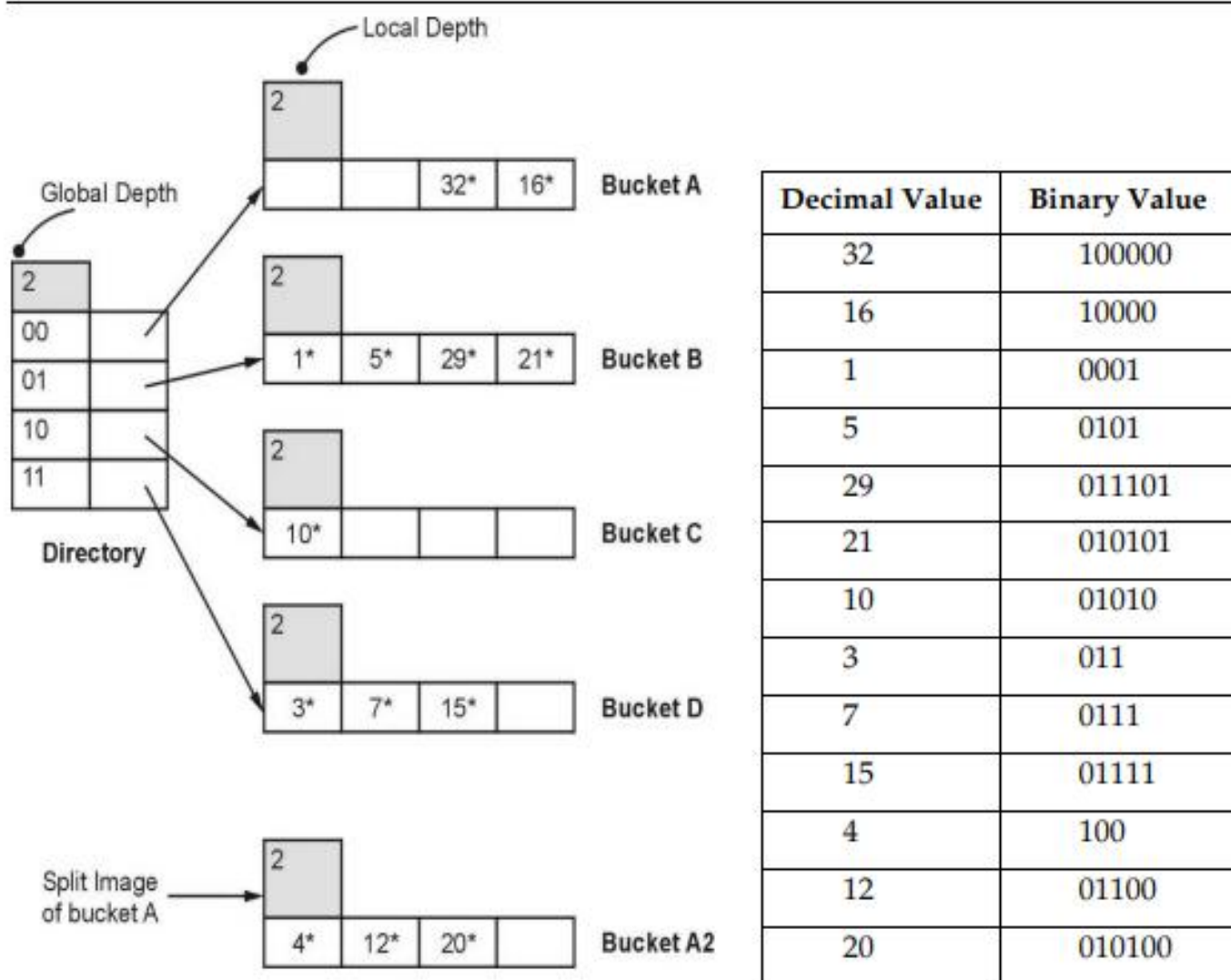
**Fig. 4.10.1 Extendible hashing**

To locate a data entry, we apply a hash function to search the data we use **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, 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. 4.10.2



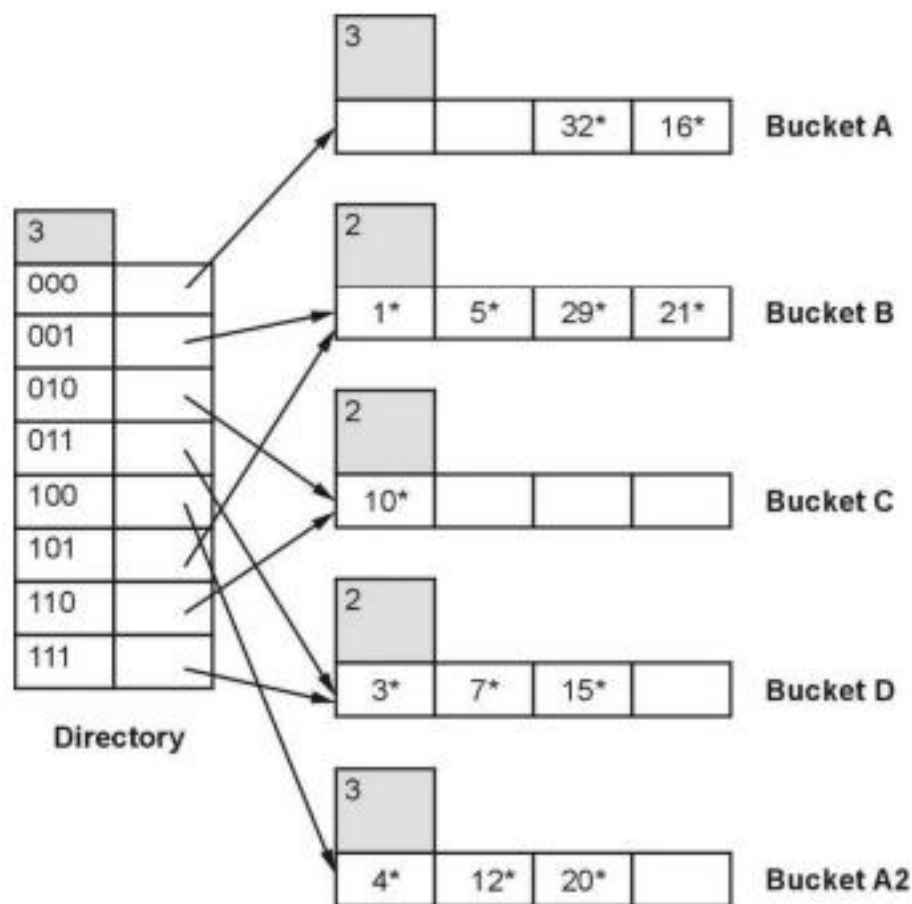


**Fig. 4.10.2 : During insertion process**

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. 4.10.3.

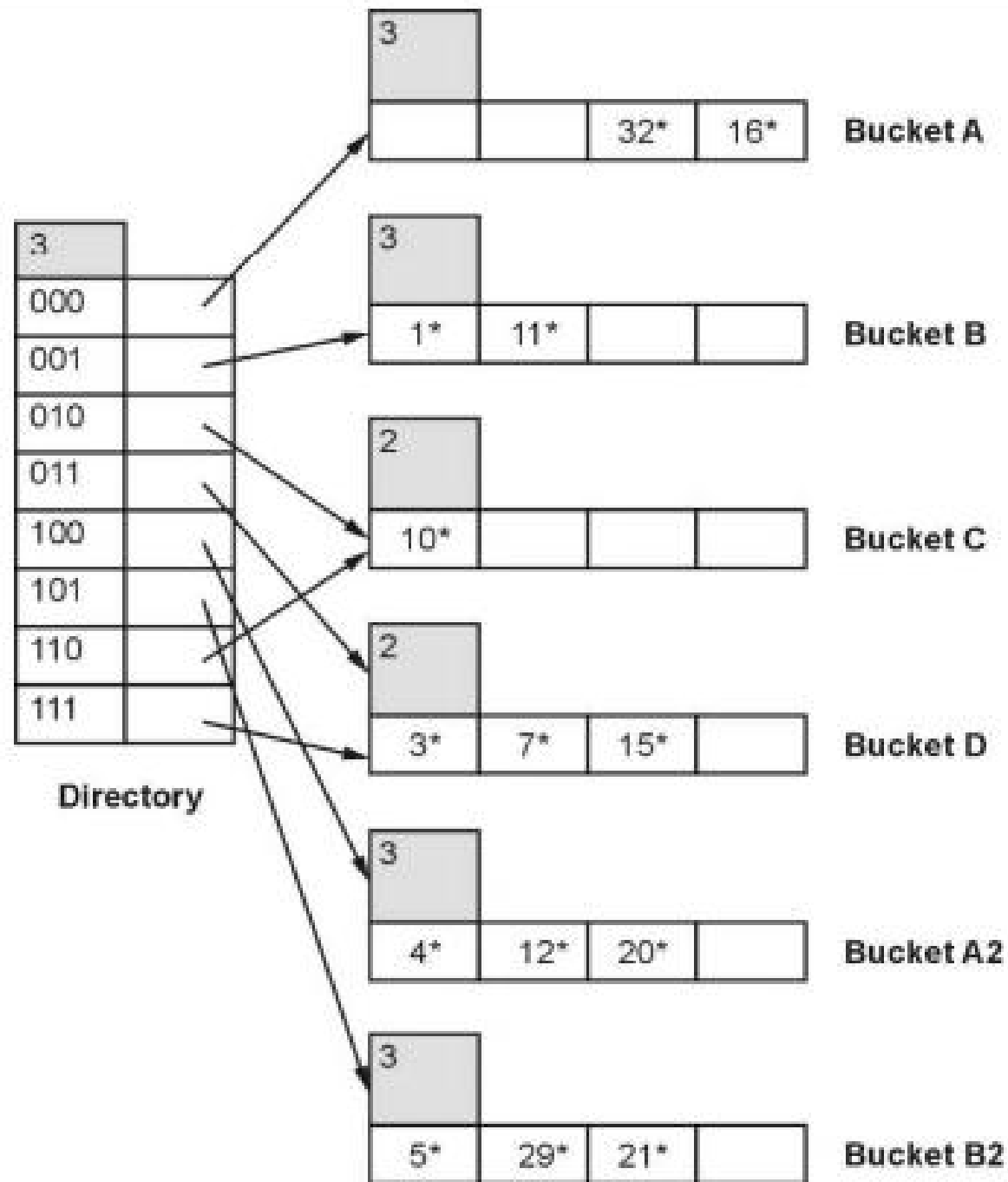
**(Note :** Student should refer binary values given in Fig. 4.10.2, for understanding insertion operation)



**Fig. 4.10.3 : 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  $1 = 001$   
 $11 = 10001$
- For bucket B2, we get  
 $5 = 101$   
 $29 = 11101$   
 and  $21 = 10101$

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



**Fig. 4.10.4 : After insertion of 11\***