Indexing

Indexing is a data structure technique which allows you to quickly retrieve records from a database file.

For ex: Author catalog in Library

An Index is a small table having only two columns.

Search key	Pointer

Search Key - attribute to set of attributes used to look up records in a file.

An **index file** consists of records (called **index entries**) of the form

The first column comprises a copy of the primary or candidate key of a table. Its second column contains a set of pointers for holding the address of the disk block where that specific key value stored.

Usually Indexes are smaller than the original file.

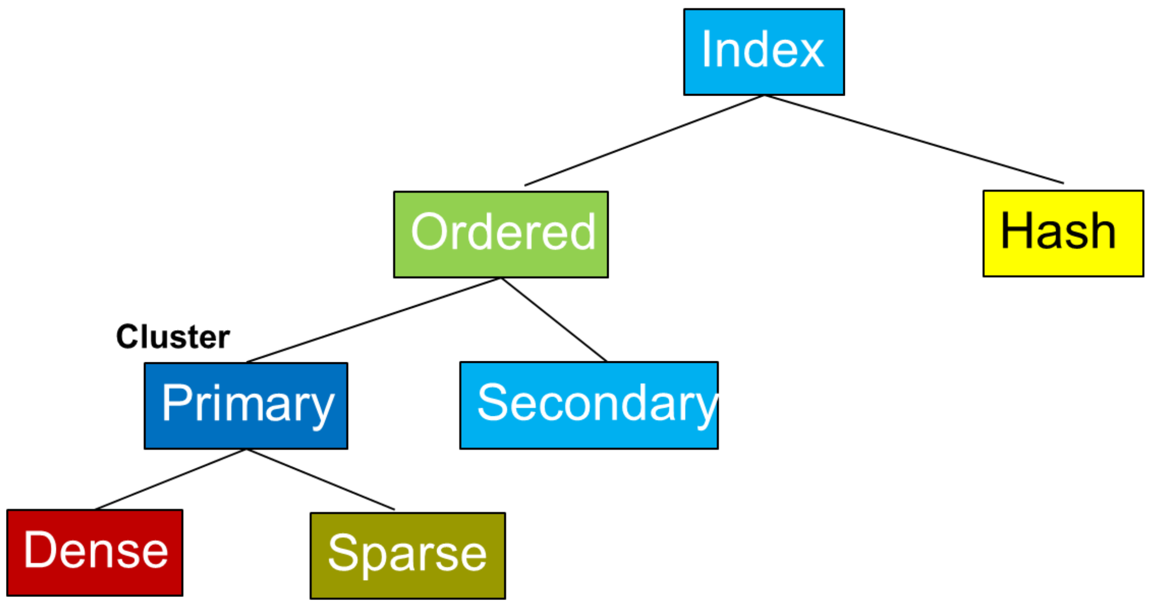
An index -

- Takes a search key as input
- Efficiently returns a collection of matching records.

In this DBMS Indexing tutorial, you will learn:

- Types of Indexing
- Primary Index
- Secondary Index
- Clustering Index
- What is Multilevel Index?
- B-Tree Index

Types of Indexing



Type of Indexes in Database

Two basic kinds of indices:

1. **Ordered indices:** search keys are stored in sorted order
2. **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

The following metrics are used to evaluate indexing methods:

- Access types
- Access time
- Insertion time
- Deletion time
- Space Overhead

Ordered Indices:

In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.

Primary index: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.

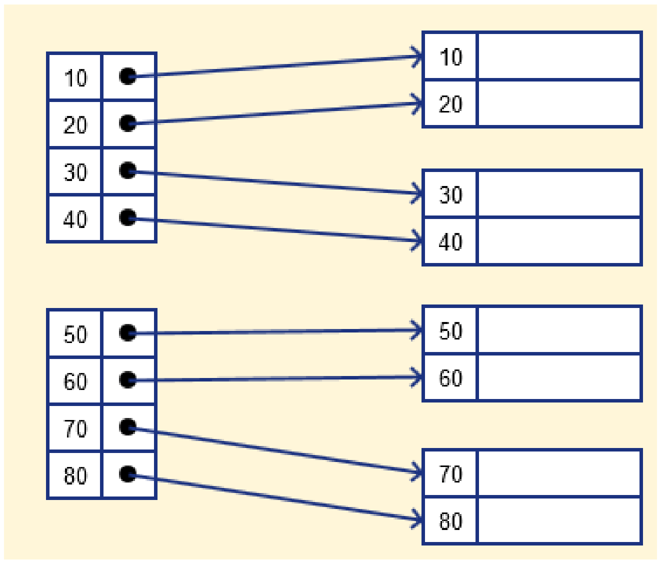
1. Also called **clustering index**
2. The search key of a primary index is usually but not necessarily the primary key.

Secondary index: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**

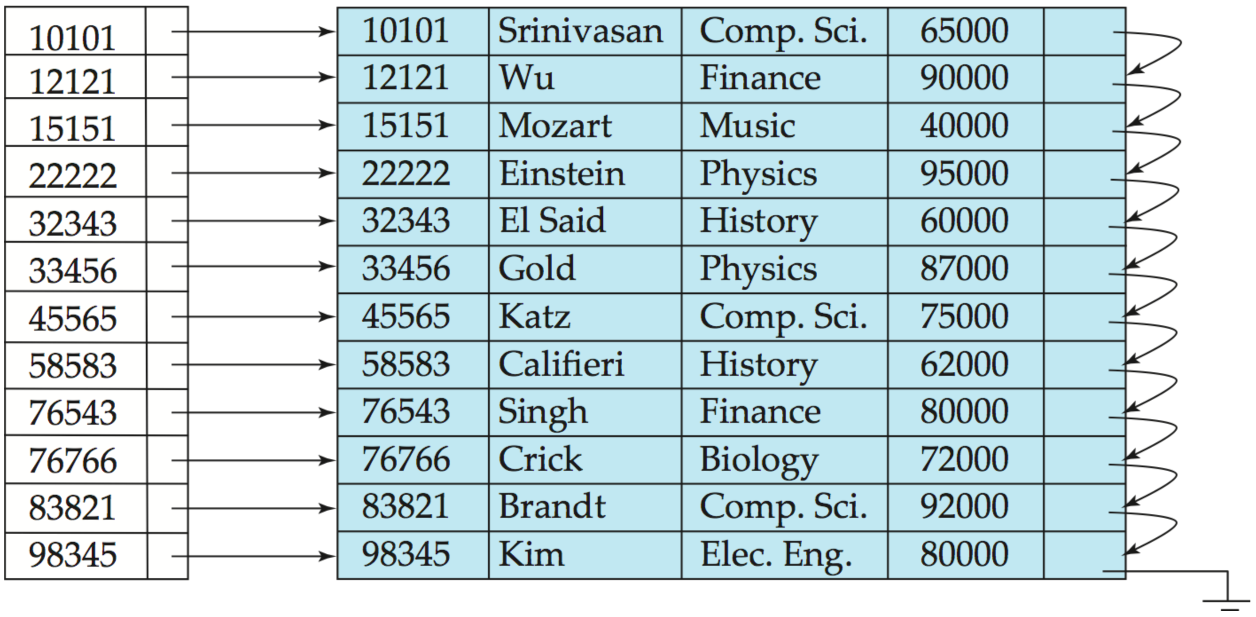
Index-sequential file: ordered sequential file with a primary index

Dense Index:

In a dense index, a record is created for every search key valued in the database. This helps you to search faster but needs more space to store index records. In this Indexing, method records contain search key value and points to the real record on the disk.



E.g. index on *ID* attribute of *instructor* relation



10101	10101	Srinivasan	Comp. Sci.	65000
12121	12121	Wu	Finance	90000
15151	15151	Mozart	Music	40000
22222	22222	Einstein	Physics	95000
32343	32343	El Said	History	60000
33456	33456	Gold	Physics	87000
45565	45565	Katz	Comp. Sci.	75000
58583	58583	Califieri	History	62000
76543	76543	Singh	Finance	80000
76766	76766	Crick	Biology	72000
83821	83821	Brandt	Comp. Sci.	92000
98345	98345	Kim	Elec. Eng.	80000

Sparse Index

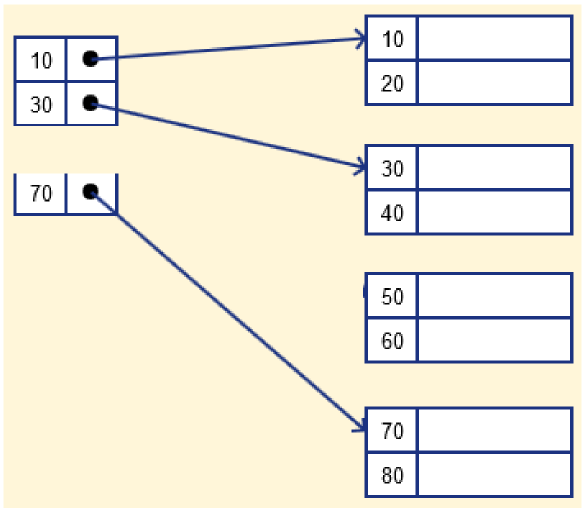
It is an index record that appears for only some of the search key values in the file. Applicable when records are sequentially ordered on search-key.

Sparse Index helps you to resolve the issues of dense Indexing in DBMS. In this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrieved, the block address will be fetched.

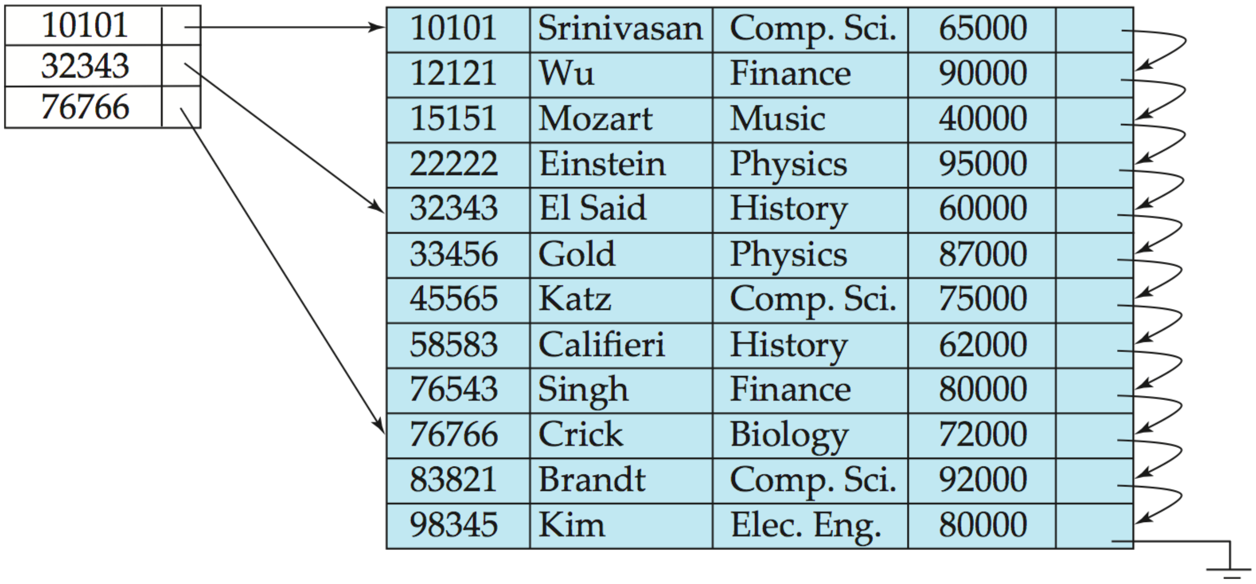
Comparisons:

Sparse Index stores index records for only some search-key values. It needs less space, less maintenance overhead for insertion, and deletions but It is slower compared to the dense Index for locating records.

Below is an database index Example of Sparse Index



Another Example:



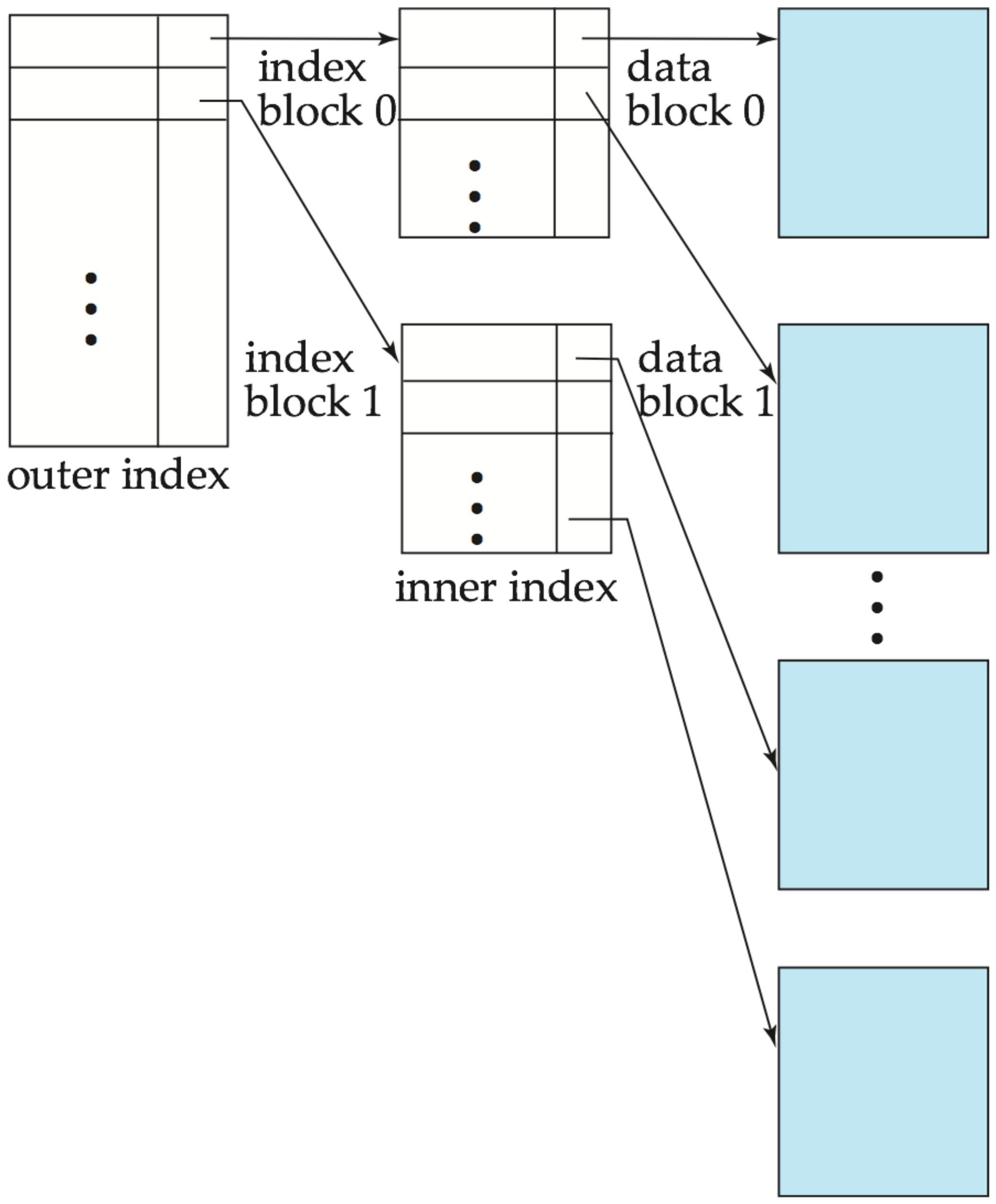
10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Multi-level Indexing

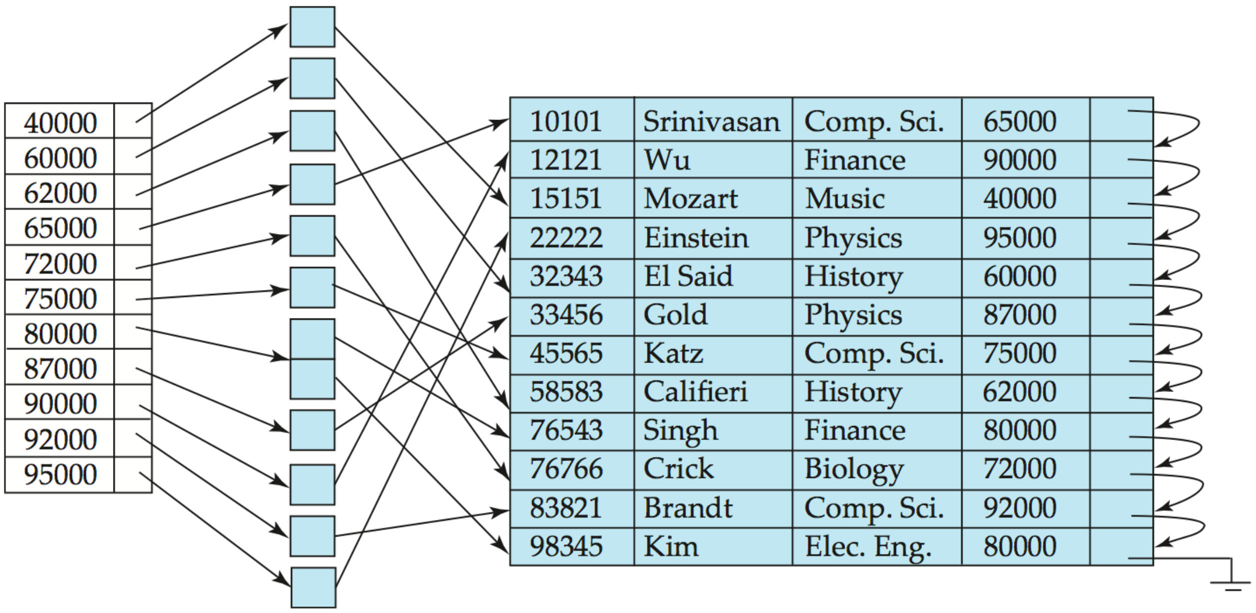
- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
    1. outer index – a sparse index of primary index
    2. inner index – the primary index file

- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.



Secondary Indices:

1. Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
   - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
   - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
2. We can have a secondary index with an index record for each search-key value



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

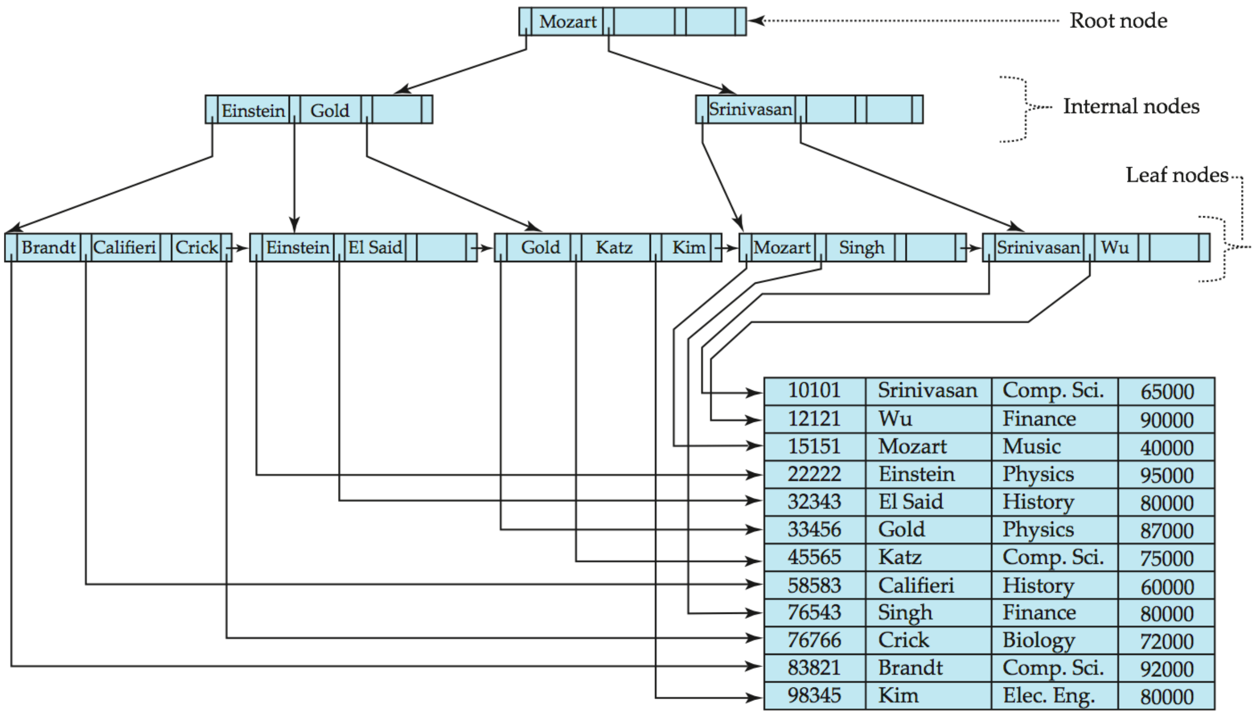
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

B$^+$-Tree Index Files

B$^+$-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files

  - l performance degrades as file grows, since many overflow blocks get created.
  - l Periodic reorganization of entire file is required.
- n Advantage of B^+-tree index files:
  - l automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - l Reorganization of entire file is not required to maintain performance.
- n (Minor) disadvantage of B^+-trees:
  - l extra insertion and deletion overhead, space overhead.
- n Advantages of B^+-trees outweigh disadvantages
  - l B^+-trees are used extensively



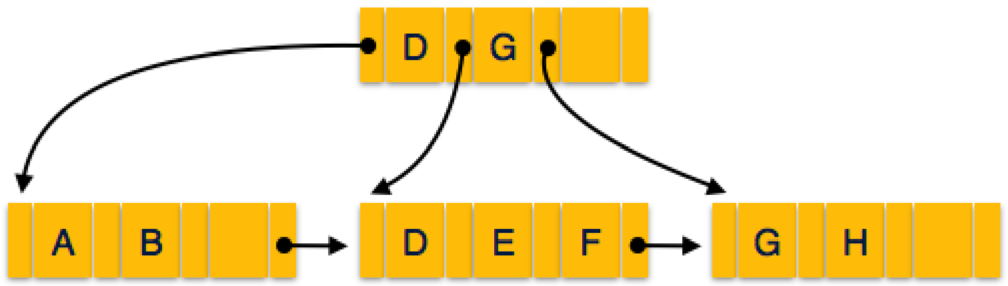
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Structure of B^+ Tree

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

- o The B+ tree is a balanced binary search tree. It follows a multi-level index format.

- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.



Internal nodes −

- Internal (non-leaf) nodes contain at least ⌈n/2⌉ pointers, except the root node.
- At most, an internal node can contain **n** pointers.
- Each node can have at most **m** keys and **m+1** pointers.

Leaf nodes −

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

Properties of B^+ Tree:

1. All leaves are at the same level.
2. The root has at least two children.
3. Each node except root can have a maximum of `m` children and at least `m/2` children.
4. Each node can contain a maximum of `m` `- 1` keys and a minimum of `⌈m/2⌉` `-` `1` keys

B^+ Tree Insertion

- B^+ trees are filled from bottom and each entry is done at the leaf node.

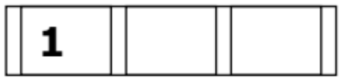
- If a leaf node overflows −
    - Split node into two parts.
    - Partition at **i = ⌊(m+1)$_{/2}$⌋.**
    - First **i** entries are stored in one node.
    - Rest of the entries (i+1 onwards) are moved to a new node.
    - **i**th key is duplicated at the parent of the leaf.
- If a non-leaf node overflows −
    - Split node into two parts.
    - Partition the node at **i = ⌈(m+1)$_{/2}$⌉**.
    - Entries up to **i** are kept in one node.
    - Rest of the entries are moved to a new node.

Suppose that B+ Tree node can hold upto 4 pointers and 3 keys

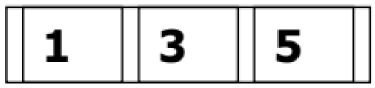
M=3 and d=1

Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

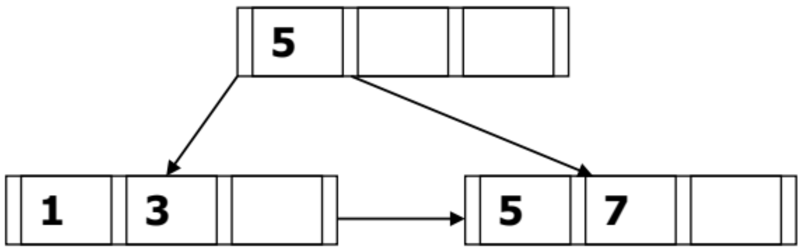
Insert 1



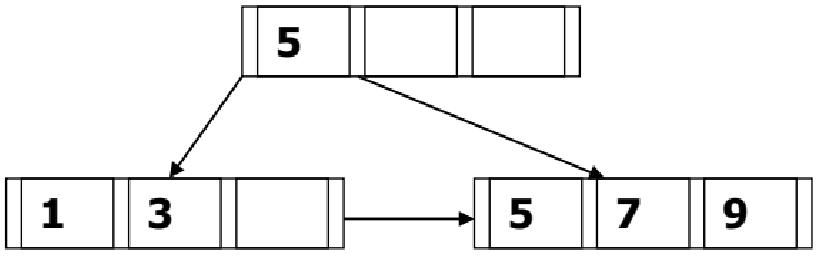
Insert 3, 5



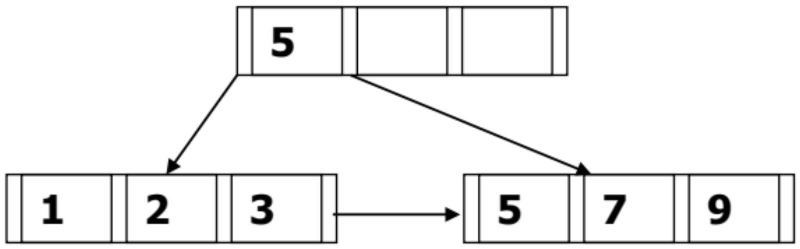
Insert 7



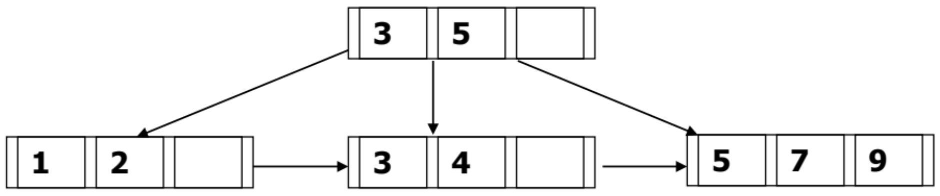
Insert 9



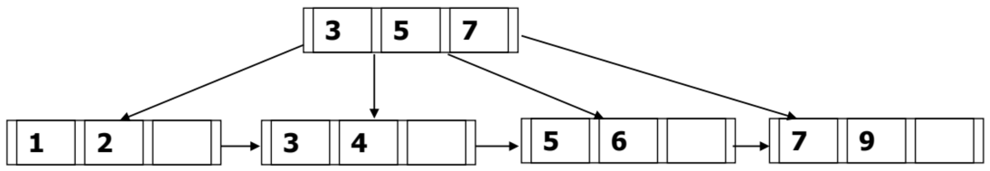
Insert 2



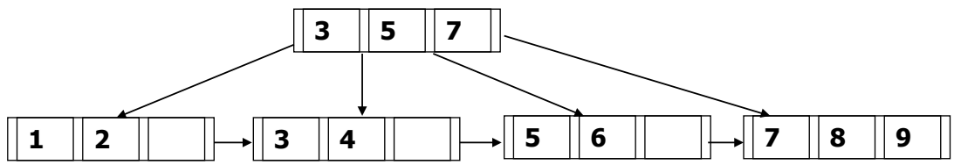
Insert 4



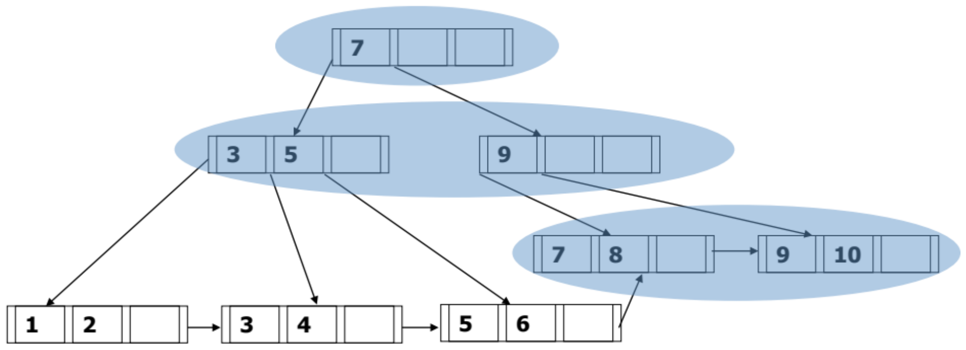
Insert 6



Insert 8



Insert 10

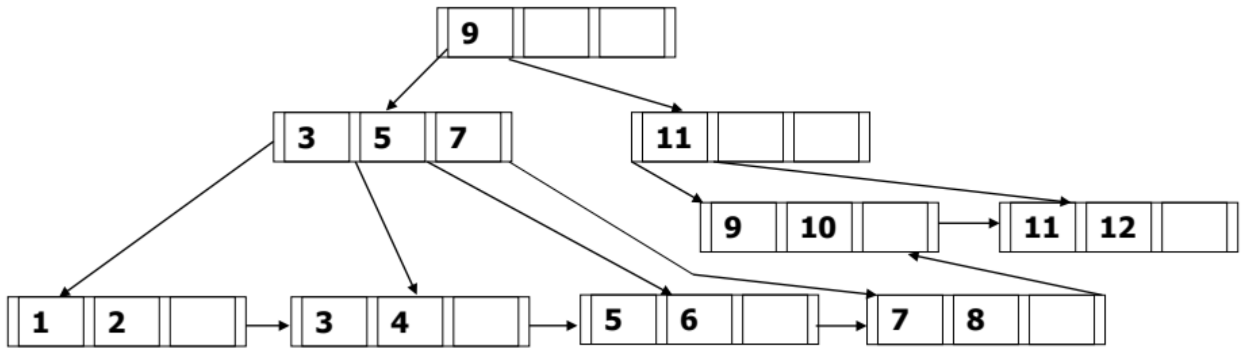


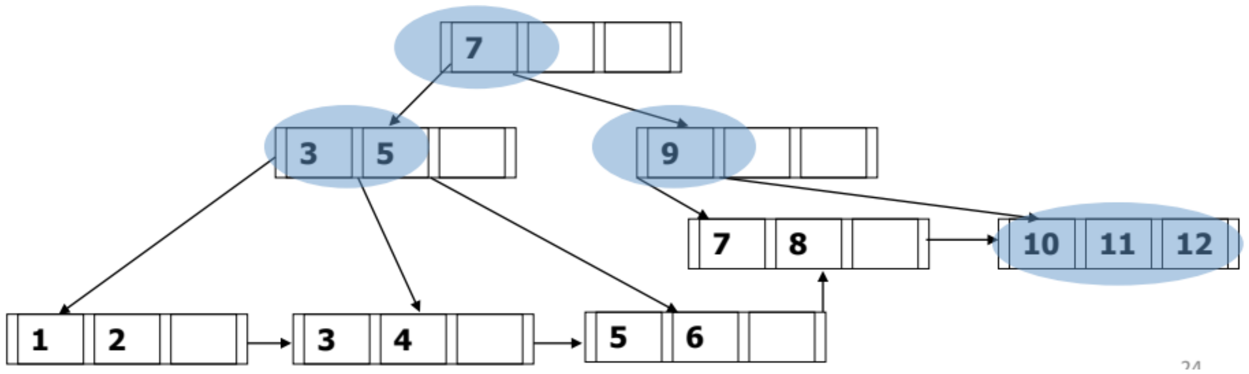
Remove 9, 7, 8

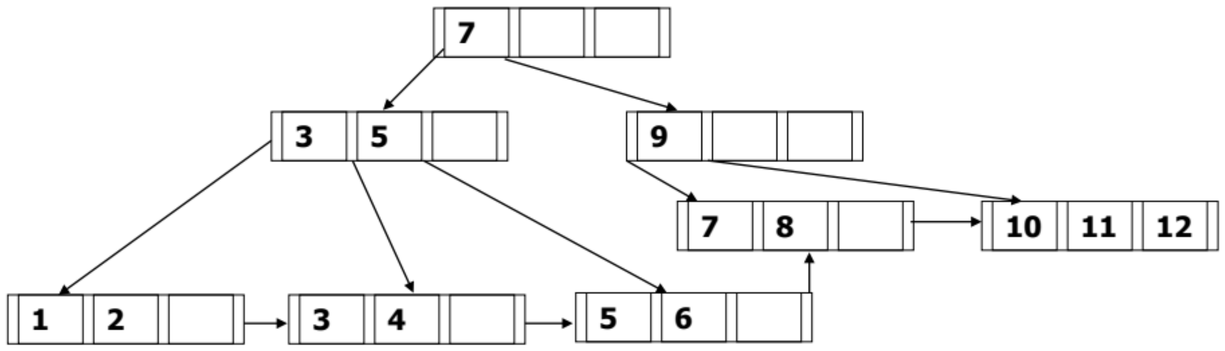
B+ Tree Deletion

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

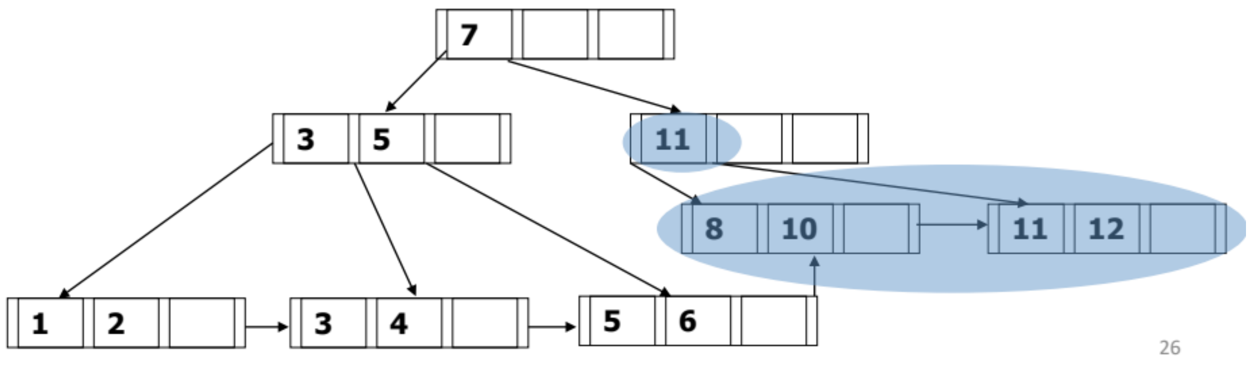
Remove 9, 7, 8

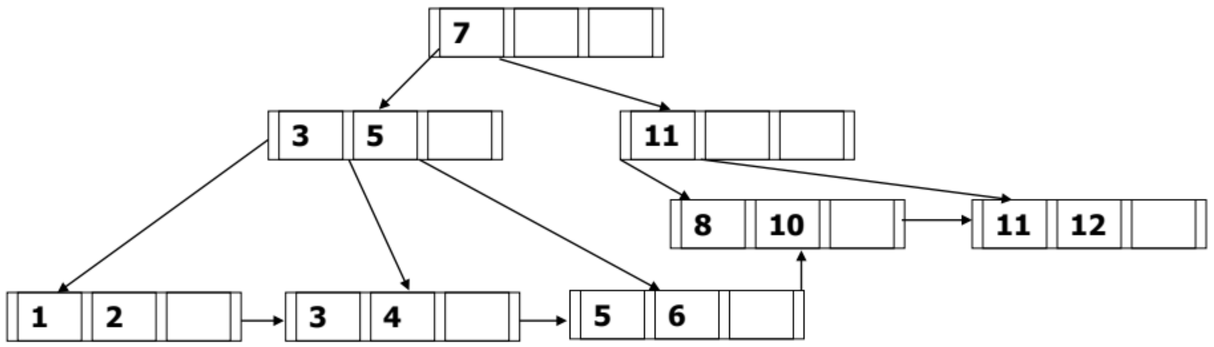




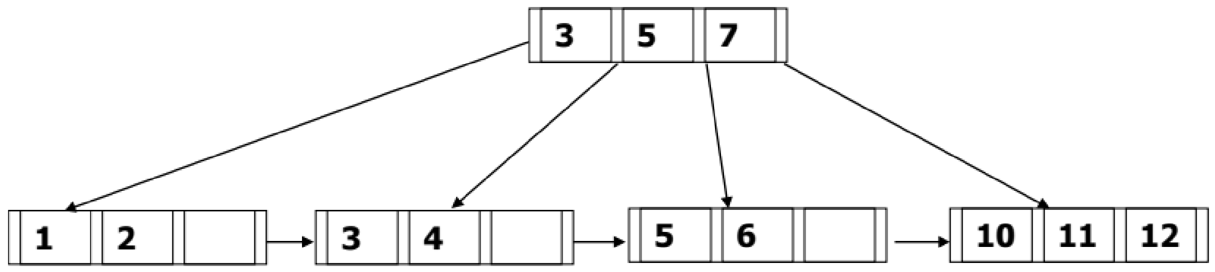


After removing 7



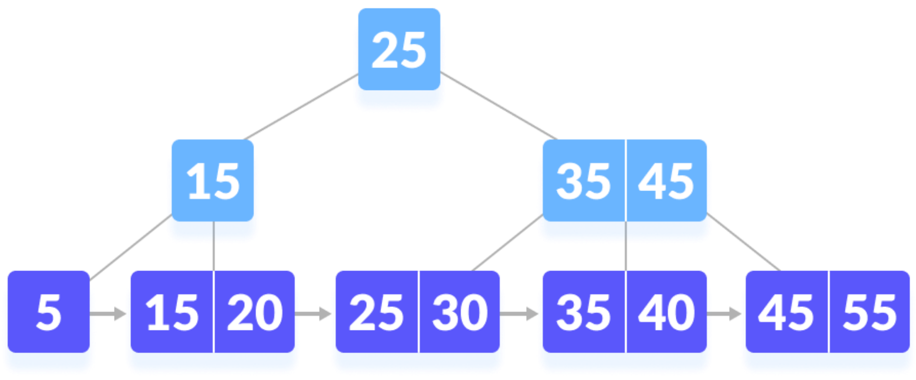


After removing 8

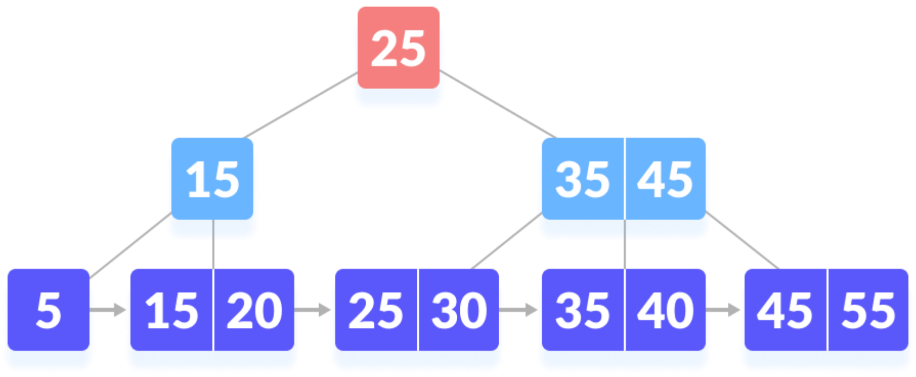


Searching

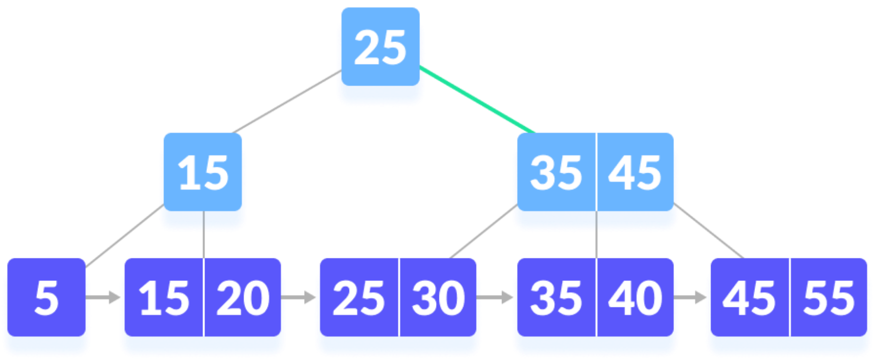
Let us say that we want to search value k is 45 from the following tree



Compare k with the root node.

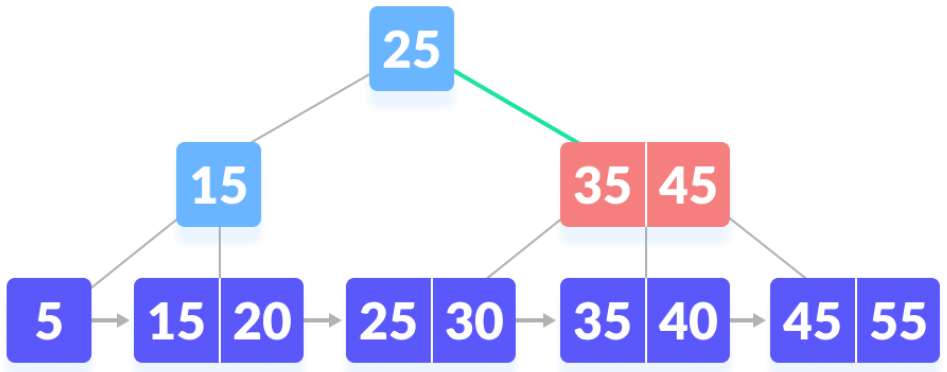


Since k > 25, go to the right child.



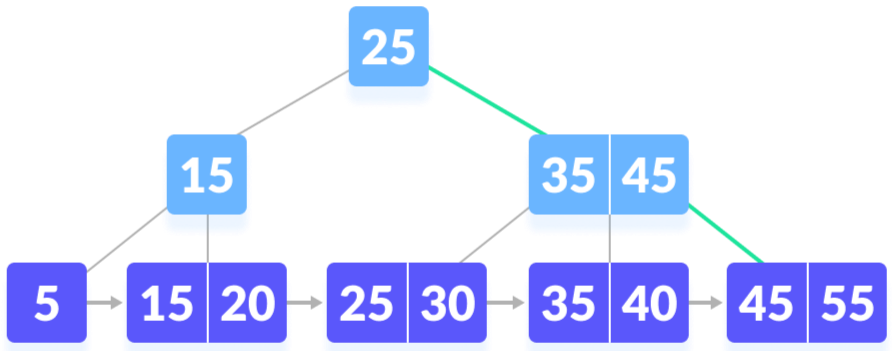
Go to the right root

Compare k with 35. Since k > 30, compare k with 45.



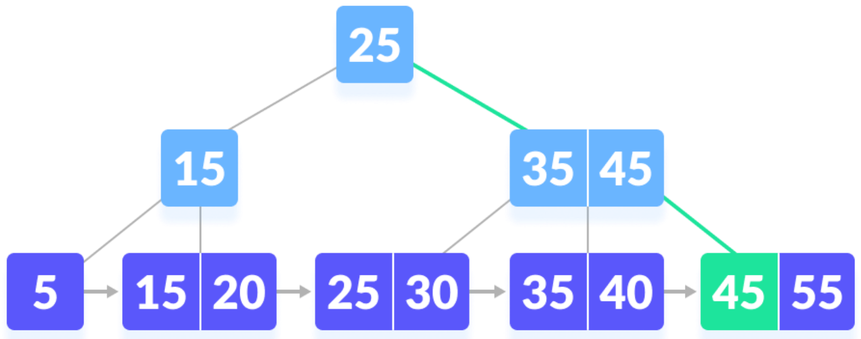
1. k not found

   Since k ≥ 45, so go to the right child.



1. go to the right

   k is found.



RAID (Redundant Array of Independent Disks)

RAID refers to redundancy array of the independent disk. It is a technology which is used to connect multiple secondary storage devices for increased performance, data redundancy or both. It gives you the ability to survive one or more drive failure depending upon the RAID level used.

It consists of an array of disks in which multiple disks are connected to achieve different goals.

RAID technology

There are 7 levels of RAID schemes. These schemas are as RAID 0, RAID 1, ..., RAID 6.

These levels contain the following characteristics:

- It contains a set of physical disk drives.
- In this technology, the operating system views these separate disks as a single logical disk.
- In this technology, data is distributed across the physical drives of the array.
- Redundancy disk capacity is used to store parity information.
- In case of disk failure, the parity information can be helped to recover the data.

RAID 0

-
-

Disk 0	Disk 1	Disk 2	Disk 3
20	21	22	23
24	25	26	27

28	29	30	31
32	33	34	35

-
- RAID level 0 provides data stripping, i.e., a data can place across multiple disks. It is based on stripping that means if one disk fails then all data in the array is lost.
- This level doesn't provide fault tolerance but increases the system performance.

Example:

In this figure, block 0, 1, 2, 3 form a stripe.

In this level, instead of placing just one block into a disk at a time, we can work with two or more blocks placed it into a disk before moving on to the next one.

Disk 0	**Disk 1**	**Disk 2**	**Disk 3**
20	22	24	26
21	23	25	27
28	30	32	34
29	31	33	35

In this above figure, there is no duplication of data. Hence, a block once lost cannot be recovered.

Pros of RAID 0:

- In this level, throughput is increased because multiple data requests probably not on the same disk.
- This level full utilizes the disk space and provides high performance.

- It requires minimum 2 drives.

Cons of RAID 0:

- It doesn't contain any error detection mechanism.
- The RAID 0 is not a true RAID because it is not fault-tolerance.
- In this level, failure of either disk results in complete data loss in respective array.

RAID 1

This level is called mirroring of data as it copies the data from drive 1 to drive 2. It provides 100% redundancy in case of a failure.

Example:

Disk 0	Disk 1	Disk 2	Disk 3
A	A	B	B
C	C	D	D
E	E	F	F
G	G	H	H

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.

Pros of RAID 1:

- The main advantage of RAID 1 is fault tolerance. In this level, if one disk fails, then the other automatically takes over.
- In this level, the array will function even if any one of the drives fails.

Cons of RAID 1:

- In this level, one extra drive is required per drive for mirroring, so the expense is higher.

RAID 2

- RAID 2 consists of bit-level striping using hamming code parity. In this level, each data bit in a word is recorded on a separate disk and ECC code of data words is stored on different set disks.
- Due to its high cost and complex structure, this level is not commercially used. This same performance can be achieved by RAID 3 at a lower cost.

Pros of RAID 2:

- This level uses one designated drive to store parity.
- It uses the hamming code for error detection.

Cons of RAID 2:

- It requires an additional drive for error detection.

RAID 3

- RAID 3 consists of byte-level striping with dedicated parity. In this level, the parity information is stored for each disk section and written to a dedicated parity drive.
- In case of drive failure, the parity drive is accessed, and data is reconstructed from the remaining devices. Once the failed drive is replaced, the missing data can be restored on the new drive.
- In this level, data can be transferred in bulk. Thus high-speed data transmission is possible.

Disk 0	Disk 1	Disk 2	Disk 3
A	B	C	P(A, B, C)
D	E	F	P(D, E, F)
G	H	I	P(G, H, I)
J	K	L	P(J, K, L)

Pros of RAID 3:

- In this level, data is regenerated using parity drive.
- It contains high data transfer rates.

- In this level, data is accessed in parallel.

Cons of RAID 3:

- It required an additional drive for parity.
- It gives a slow performance for operating on small sized files.

RAID 4

- RAID 4 consists of block-level stripping with a parity disk. Instead of duplicating data, the RAID 4 adopts a parity-based approach.
- This level allows recovery of at most 1 disk failure due to the way parity works. In this level, if more than one disk fails, then there is no way to recover the data.
- Level 3 and level 4 both are required at least three disks to implement RAID.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

In this figure, we can observe one disk dedicated to parity.

In this level, parity can be calculated using an XOR function. If the data bits are 0,0,0,1 then the parity bits is XOR(0,1,0,0) = 1. If the parity bits are 0,0,1,1 then the parity bit is XOR(0,0,1,1)= 0. That means, even number of one results in parity 0 and an odd number of one results in parity 1.

C1	C2	C3	C4	Parity
0	1	0	0	1
0	0	1	1	0

Suppose that in the above figure, C2 is lost due to some disk failure. Then using the values of all the other columns and the parity bit, we can recompute the data bit stored in C2. This level allows us to recover lost data.

RAID 5

- RAID 5 is a slight modification of the RAID 4 system. The only difference is that in RAID 5, the parity rotates among the drives.
- It consists of block-level striping with DISTRIBUTED parity.
- Same as RAID 4, this level allows recovery of at most 1 disk failure. If more than one disk fails, then there is no way for data recovery.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

This figure shows that how parity bit rotates.

This level was introduced to make the random write performance better.

Pros of RAID 5:

- This level is cost effective and provides high performance.
- In this level, parity is distributed across the disks in an array.
- It is used to make the random write performance better.

Cons of RAID 5:

- In this level, disk failure recovery takes longer time as parity has to be calculated from all available drives.
- This level cannot survive in concurrent drive failure.

RAID 6

- This level is an extension of RAID 5. It contains block-level stripping with 2 parity bits.

- In RAID 6, you can survive 2 concurrent disk failures. Suppose you are using RAID 5, and RAID 1. When your disks fail, you need to replace the failed disk because if simultaneously another disk fails then you won't be able to recover any of the data, so in this case RAID 6 plays its part where you can survive two concurrent disk failures before you run out of options.

Disk 1	Disk 2	Disk 3	Disk 4
A0	B0	Q0	P0
A1	Q1	P1	D1
Q2	P2	C2	D2
P3	B3	C3	Q3

Pros of RAID 6:

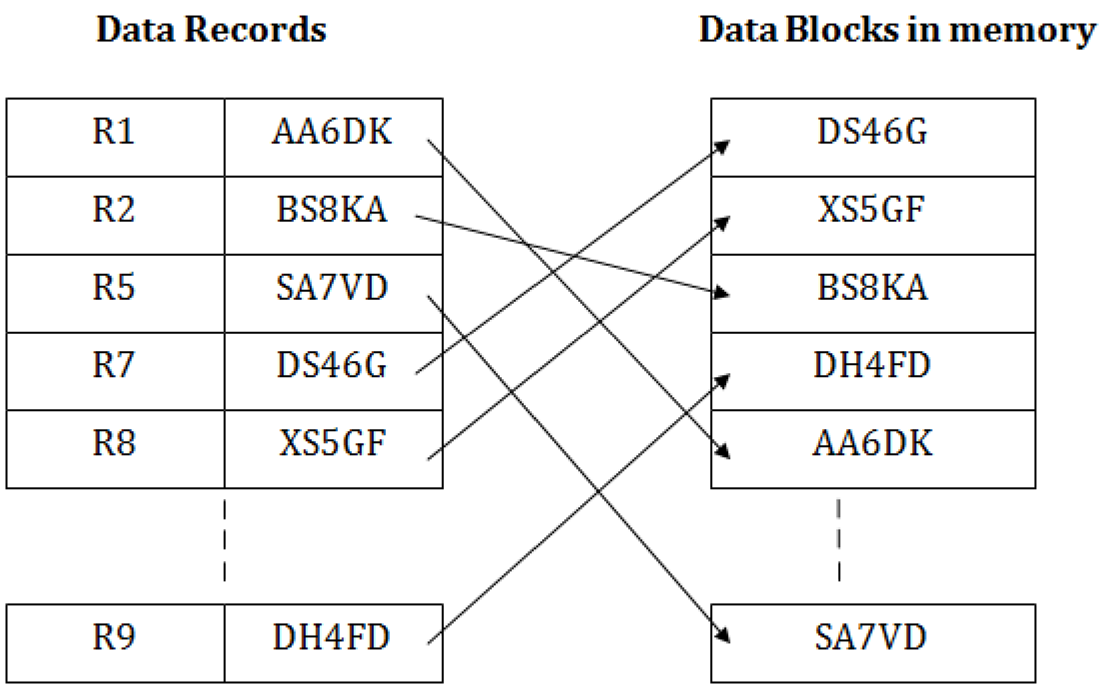
- This level performs RAID 0 to strip data and RAID 1 to mirror. In this level, stripping is performed before mirroring.
- In this level, drives required should be multiple of 2.

Cons of RAID 6:

- It is not utilized 100% disk capability as half is used for mirroring.
- It contains very limited scalability.

Indexed sequential access method (ISAM)

ISAM method is an advanced sequential file organization. In this method, records are stored in the file using the primary key. An index value is generated for each primary key and mapped with the record. This index contains the address of the record in the file.



If any record has to be retrieved based on its index value, then the address of the data block is fetched and the record is retrieved from the memory.

Pros of ISAM:

- In this method, each record has the address of its data block, searching a record in a huge database is quick and easy.
- This method supports range retrieval and partial retrieval of records. Since the index is based on the primary key values, we can retrieve the data for the given range of value. In the same way, the partial value can also be easily searched, i.e., the student name starting with 'JA' can be easily searched.

Cons of ISAM

- This method requires extra space in the disk to store the index value.
- When the new records are inserted, then these files have to be reconstructed to maintain the sequence.
- When the record is deleted, then the space used by it needs to be released. Otherwise, the performance of the database will slow down.