Chapter-6,
File Structure and Hashing

6.1 Records Organizations
6.2 Disks and storage
6.3 Remote Backup System
6.4 Hashing concepts, static and Dynamic Hashing
6.5 Order Indices
6.6 B+ tree index

Overview of Physical storage Media.

Several types of data storage exist in most computer systems. These storage media are classified by the speed with which data can be accessed by the cost per unit of data to buy the medium, and by the medium's reliability. Among the media typically available are these.

i. Cache
ii. Main memory
iii. Flash memory
iv. Magnetic disk-storage
v. Optical storage
vi. Tape storage

Cache:

↳ The cache is the fastest and most costly form of storage.

↳ Cache memory is small, its use is managed by the

computer system hardware.

Main Memory:

↳ The storage medium used for data that are available to be operated on is main memory.

↳ The general-purpose machine instructions operate on main memory.

↳ Although main memory may contain many megabytes of data, or even gigabytes of data in large server systems, it is generally to small (or too expensive) for storing the entire database.

↳ The contents of main memory are usually lost if a power failure or system crash occurs.

↳ Fast access (10s to 100s of nanoseconds)

Flash memory:

↳ Data survives powes failure.

↳ Also known as EEPROM (Electrically Erasable Programmable Read Only Memory)

↳ Reading data from flash memory takes less than 100 nanoseconds, which is roughly

as fast as reading data from main memory.

↳ writing data to flash memory is more complicated data can be written once, which take about 4 to 10 miliseconds, but cannot be overwritten disectly.

↳ To overwrite memory that has been written already, we have to erase an entire bank of the memory at once, it is then ready to be written again.

↳ Flash memory can support only a limited number of erase cycles, ranging from 10,000 to 1 million.

↳ Cost per unit of storage roughly similar to main memory.

↳ Widely used in embedded devices such as digital cameras.

Magnetic-disk.

↳ The primary medium for the long-term online storage of data is the magnetic disk, typically stores entire database.

↳ The system must move the data from disk to main memory so that they can be accessed.

↳ After the system has performed the designated operations, the data that have been modified must

be written to disk.

↳ The size of magnetic disks ranges from few gigabytes to 1 terabyte

↳ Both lower and upper end of this range have been growing at about 50 percent per year, & we can expect much larger capacity disks every year.

↳ Disk storage survives power failures and system crashes

Optical storage:
Magnetic Tapes:
Storage Hierarchy.

RAID (Redundant Arrays of Independent Disks)

↳ Disk organization techniques that manage a large number of disks, providing a view of a single disk of high capacity and high speed by using multiple disks in parallel, and high reliability by storing data redundantly, so that data can be recovered even if a disk fails.

↳ The chance that at least one disk out of a set of N disks will fail is much higher than the chance that a single disk will fail.

↳ Suppose that the mean time to failure of a disk is 100,000 hours, or slightly over 11 years. Then, the mean time to failure of somedisk inan array of 100 disks will be 100,000/100 = 1000 hours, or around 42 days, which is not long at all! If we store only one copy of the data, then each disk failure will result in loss of significant amount of data. Such a high frequeny of data loss is unacceptable.

RAID Levels:

↳ schemas to provide redundancy at lowes cost by using disk striping combined with pasity bits.

↳ Different RAID organizations or RAID levels

have different cost, performance and reliability

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

RAID 0

↳ 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.

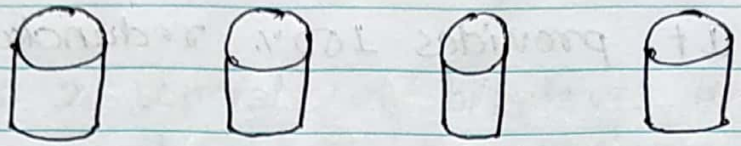
↳ This level doesn't provide fault tolerance but increases the system performance.

↳ Used in high-performance applications where data lost is not critical.

Example:

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

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



a. RAID 0: nonredundant striping

Pros of RAID 0:

↳ In this level, throughput is increased because multiple data request 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.

↳ Refers to disk mirroring with block striping

↳ It copies the data from drive 1 to drive 2. It provides 100% redundancy in case of 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 drive is used to store the data. The other half of the drive is just a mirror to the already stored data.

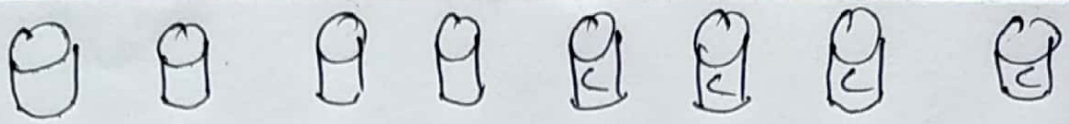
Pros of RAID 1:

↳ 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 1: mirrored disks

RAID 2:

↳ RAID 2 consists of bit-level striping using hamming code pasity. In this level, each data bit in a word is recorded on a separate disk & ECC (Error-correcting codes) code of data words is stored on diffesent set disks.

↳ Due to high cost and complex structure, this level is not commercially used. This same performance can be achieved by RAID 3 at a lower cost.

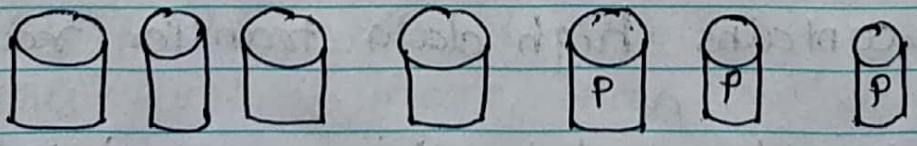
Pros of RAID 2:

↳ This level uses one designated drive to store pasity.

↳ It uses the hamming code for error detection.

Cons of RAID 2:

It requires an additional drive for error detection.



RAID 2: Memory-style error-correcting codes.

RAID 3.

↳ RAID 3 consists of byte-level striping with dedicated parity information is stored for each disk section and written to a dedicated pasity 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 pasity drive.

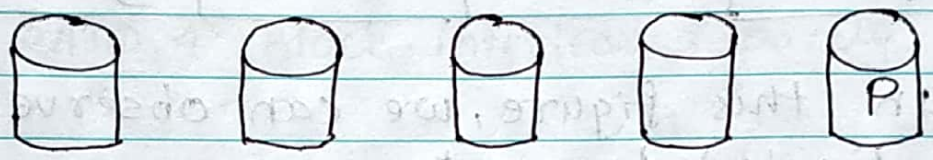
↳ It contains high data transfer rates

↳ In this level, data is accessed in pasallel.

Cons of RAID 3:

↳ It requires an additional drive for pasity.

↳ It gives a slow performance for operating on small sized files.



RAID 3: bit-interleaved parity.

RAID level 4:

↳ Block-interleaved parity organization, uses block-level stripping, like RAID 0, and in addition keeps a pasity block on a separate disk for corresponding blocks from N other disks.

↳ Instead of duplicating data, the RAID 4 adopts a pasity-based approach.

↳ This level allows recovery of at most 1 disk failure due to the way pasity works. In this level, if more than one disk fails, then these is no way to recover data.

↳ level 3 and level 4 both are requised at least three disks to implement RAID

Disk 0	Disk 1	Disk 2	Disk 3
A	B	C	P_0
D	E	F	P_1
G	H	I	P_2
J	K	L	P_3

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 (0,0,0,1) = 1.

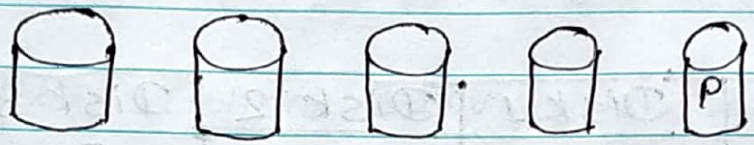
↳ If the parity bits are 0,0,1,1 then parity bit is XOR (0,0,1,1) = 0

↳ That means, even number of one results in parity 0 and an odd number of one results in parity 1.

C_1	C_2	C_3	C_4	Parity
0	1	0	0	1
0	0	1	1	0

↳ Suppose that in the above figure, C_2 is lost

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



RAID 4: Block-interleaved parity

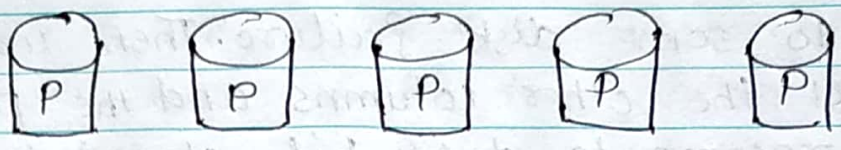
RAID 5:

↳ Block Interleaved Distributed Parity: partitions data and parity among all N+1 disks, rather than storing data in N disks and parity in 1 disk.

↳ RAID 5 is 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.



RAID 5 : block-interleaved distributed parity

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P_1	4
10	11	P_2	8	9
15	P_3	12	13	14
P_4	16	17	18	19

↳ This figure shows that horo 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 bettes.

Cons of RAID 5:

↳ In this level, disk failure recovery takes longer time as parity has to be calculated from all available drives.

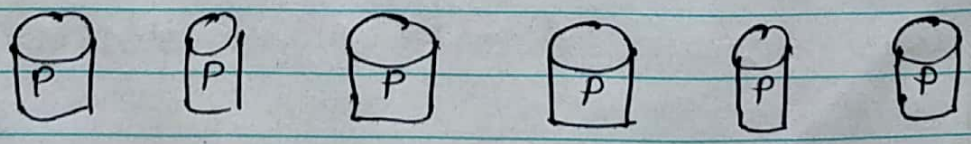
↳ This level cannot survive in concurrent dsive failure.

RAID 6.

↳ P+Q Redundany scheme; similar to level 5, but stores extra redundant information to guard against multiple disk failures.

↳ In RAID 6, you can survive 2 concurrent disk failures.

↳ Suppose you are using RAID 5, and RAID1. When your disks fail, you need to replace the failed disk because if simultaneously another disk fails then you wont 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



RAID 6: P+Q redundany.

Disk 1	Disk 2	Disk 3	Disk 4
A_0	B_0	Q_0	P_0
A_1	Q_1	P_1	D_1
Q_2	P_2	C_2	D_2
P_3	B_3	C_3	Q_3

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.

Con's of RAID 6:

↳ 100 % disk capability is not utilized as half is used for mirroring.

↳ It contains very limited scalability

File organization.

↳ A database is mapped into a number of different files that are maintained by the underlying OS. These files reside permanently on disks.

↳ A file is organized logically as a sequence of records. These records are mapped onto disk blocks.

↳ Each file is also logically partitioned into fixed-length storage units called blocks, which are the units of both storage allocation and data transfer.

↳ Most databases use block size of 4 to 8 kilobytes by default, but many databases allow the block size to be specified when database instance is created.

↳ A block may contain several records, the exact set of records that a block contains is determined by the form of physical data organization being used.

There are two types of Records:

i. Fixed length Records
ii. Variable length Records

Fixed - Length Records.

↳ As an example, let us consider a file of instructor records for our university database. Each record of this file is defined (in psudocode) as:

```
type instructor = record
        ID varchar(5);
        name varchar(20);
        dept_name varchar(20);
        salary numeric(8,2);
    end
```

- Assume that each character occupies 1 byte and that numeric(8,2) occupies 8 bytes.

Then the instructor record is 53 bytes long. A simple approach is to use first 53 bytes for the first record, the next 53 bytes for the second record, and so on. However, there are two problems with this simple approach.

i. Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.

ii. It is difficult to delete a record from this

structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

record 0	10101	Srinivasan	Comp. Sci	65000
record 1	12121	Arjun	Finance	90000
record 2	15151	Nagarajan	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	Shahid	History	60000
record 5	33456	Ganesh	Physics	87000
record 6	45565	Karthick	Comp. Sci	75000

File containing instructor records.

↳ To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block (this number can be computed easily by dividing the block size by the record size, and discarding the fractional part). Any remaining bytes of each block are left unused.

↳ When record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on until every record following the deleted record has been moved ahead.

↳ Such an approach requires moving a large number

of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record.

↳ It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Since insertion tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space.

record 0	10101	Srinivasan	Comp. Sci	65000
record 1	12121	Arjun	Finance	90000
record 2	15151	Nagrajan	Music	40000
record 4	32343	Shahid	History	60000
record 5	33456	Ganesh	Physics	87000
record 6	45565	Karthick	Comp. Sci	75000

Instructor records with record 3 deleted and all records moved.

record 0	10101	Srinivasan	Comp. Sci	65000
record 1	12121	Arjun	Finance	90000
record 2	15151	Nagarajan	Music	40000
record 6	45565	Karthick	Comp. Sci	75000
record 4	32343	Shahid	History	60000
record 5	33456	Ganesh	Physics	87000

Instructor records with record 3 deleted and final record moved.

↳ At the beginning of the file, we allocate a certain number of bytes as a file header. The header will contain a variety of information about the file.

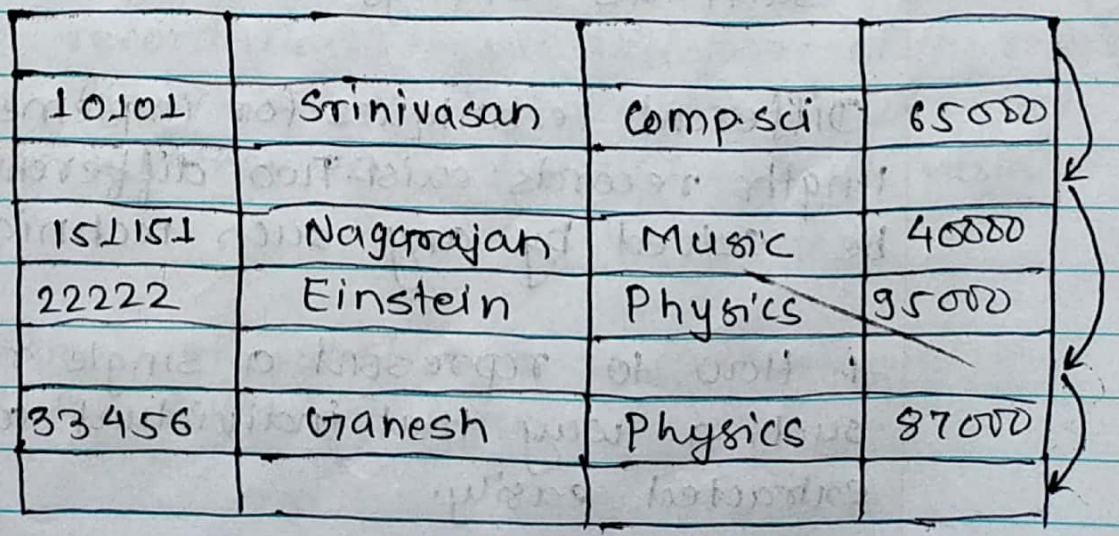
↳ For now all we need to store there is the address of the first record whose content are deleted.

↳ We use this first record to store the address of the second available record and so on.

↳ We can think of these stored addresses as pointers, since they point to the location of a record.

↳ The deleted record thus form a linked list, which is often referred to as a free list.

↳ The figure shows the file of instructor, with the free list after records 1, 4, 6 have been deleted.



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

↳ Insertion and deletion for files of fixed length records are simple to implement, because the space made available by the deleted record is exactly the space needed to insert a record.

Variable-Length Records.

Variable length records arise in database systems 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.

Different techniques for implementing variable length records exist. Two different problems must be solved by any such technique:

i. How to represent a single record in such a way that individual attributes can be extracted easily.

ii. How to store variable-length records within a block, such that records in a block can be extracted easily.

↳ The representation of a record with variable-length attributes typically has two parts: an initial part with fixed length attributes, followed by data for variable length attributes.

↳ Fixed-length attributes, such as numeric values, dates or fixed length character strings are allocated as many bytes are required to store theis value.

↳ Variable length attributes such as varchar types, are represented in the initial part of the record by a pair (offset, length), where offset denotes where the data for that attribute begins within the record, and length is the length in bytes of the vasiable-sized attribute.

↳ The values for these attributes are stored consecutively, after the initial fixed-length past of the record. Thus, the initial part of the record stores a fixed size of information about each attribute, whether it is fixed length or vasiable length.

↳ An example of such record representation is shown in the following figure.

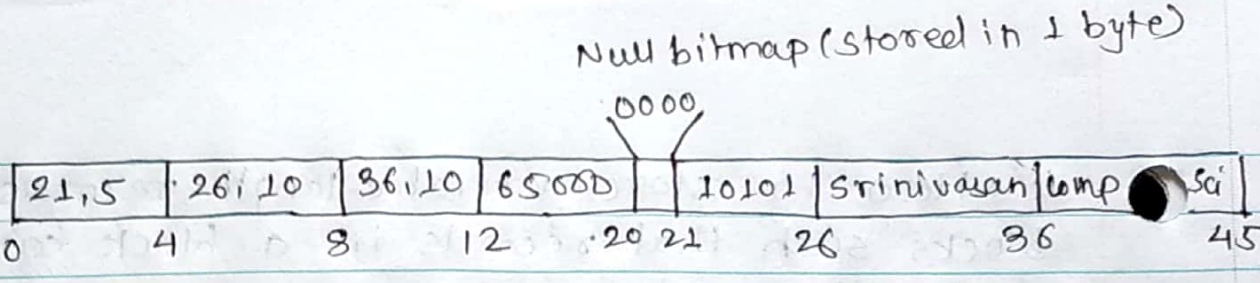


Fig: Representation of variable-length record.

↳ The figure shows an instructor record, whose first three attributes ID, name, and dept-name are variable-length strings, and whose fourth attribute salary is fixed sized number.

↳ We assume that the offset and length values are stored in two bytes each, for a total of 4 bytes per attribute. The salary attribute is assumed to be stored in 8 bytes and each string takes as many bytes as it has characters

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

↳ In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the salary value stored in bytes 12 through 19 would be ignored.

↳ The slotted page structure is commonly used for organizing records within a block. as shown below. There is a header at the

beginning of each block, containing the following information:

i. The number of record entries in the header.
ii. The end of the free space in the block.
iii. An array whose entries contain the location and size of each record.

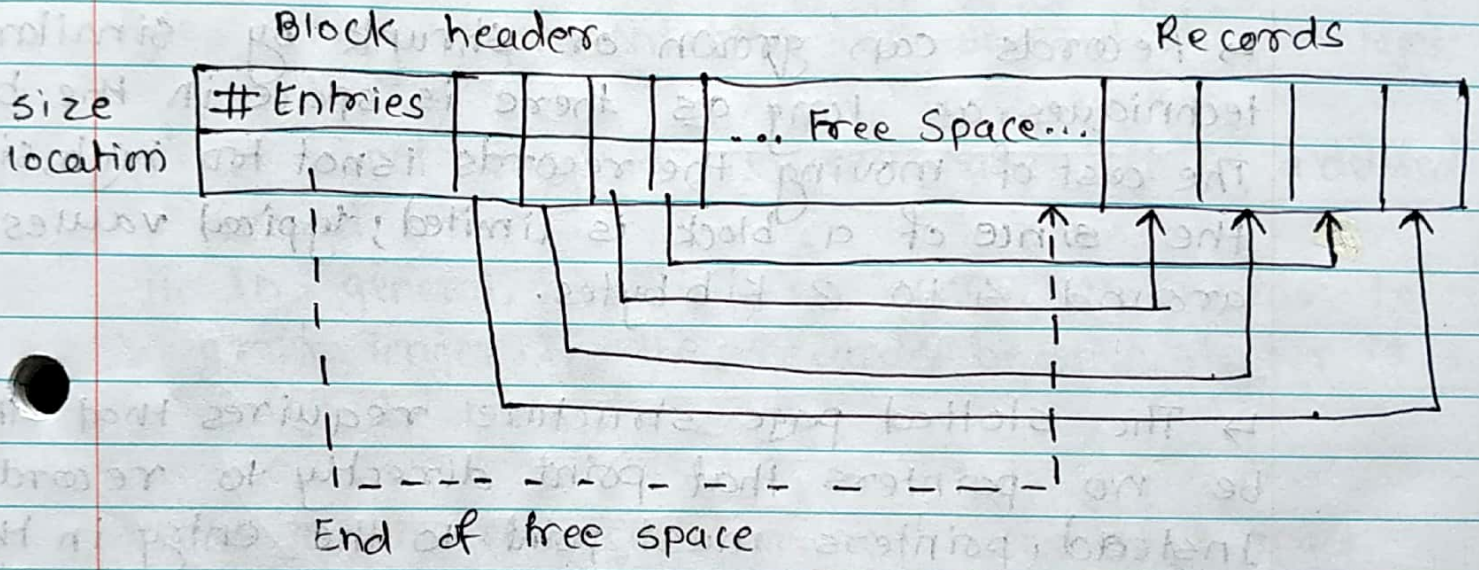


Fig: slotted-page structure.

↳ The actual records are allocated contiguously in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header array, and the first record.

↳ If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.

↳ If a record is deleted, the space that it

occupies is freed, and its entry is set to deleted (its size is set to -1, for example). Furthes, the records in the block before the deleted records are moved, so that the free space created by the deletion gets occupied, and all free space is again between the final entry in the header is appropriately updated as well.

↳ Records can grown or shrunk by similar techniques, as long as there is space in the block. The cost of moving the records isnot too high, since the size of a block is limited; typical values are around 4 to 8 Ki bytes.

↳ The slotted page structure requires that there be no pointers that point directly to records Instead, pointers must point to the entry in the header that contains the actual location of the record. This level of indirection allows records to be moved to prevent fragmentation of space inside a block, while supporting indirect pointers to the record.

Byte string Representation.

↳ Attach a special end of record (⊥) symbol to the end of record.

↳ Each record is stored as a string of sucessive bytes.

0	Perryridge	A-102	400	A-201	900	A-218	700	⊥
1	Round Hill	A-305	350	⊥				
2	Mianus	A-215	700	⊥				
3	Downtown	A-101	500	A-110	600	⊥		
4	Redwood	A-222	700	⊥				
5	Brighton	A-217	750	⊥				

Byte string representation has several disadvantages:

i. It is not easy to re-use space left by a deleted record

ii. In general, there is no space for records to grow longer. If the records become longer it must be moved.

Fixed length representation with reserved space.

can use fixed-length records of a known maximum length; unused space in shorter records filled with a null or end-of-record symbol

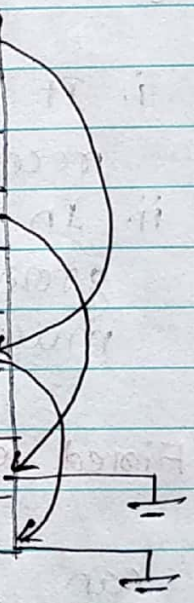
0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-101	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	brighton	A-217	750	⊥	⊥	⊥	⊥

Fixed length representation with pointer.

↳ A variable-length record is represented by a list of fixed-length records that are chained together via pointers

↳ can be used even if the maximum record length isnot known.

0	Perryride	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

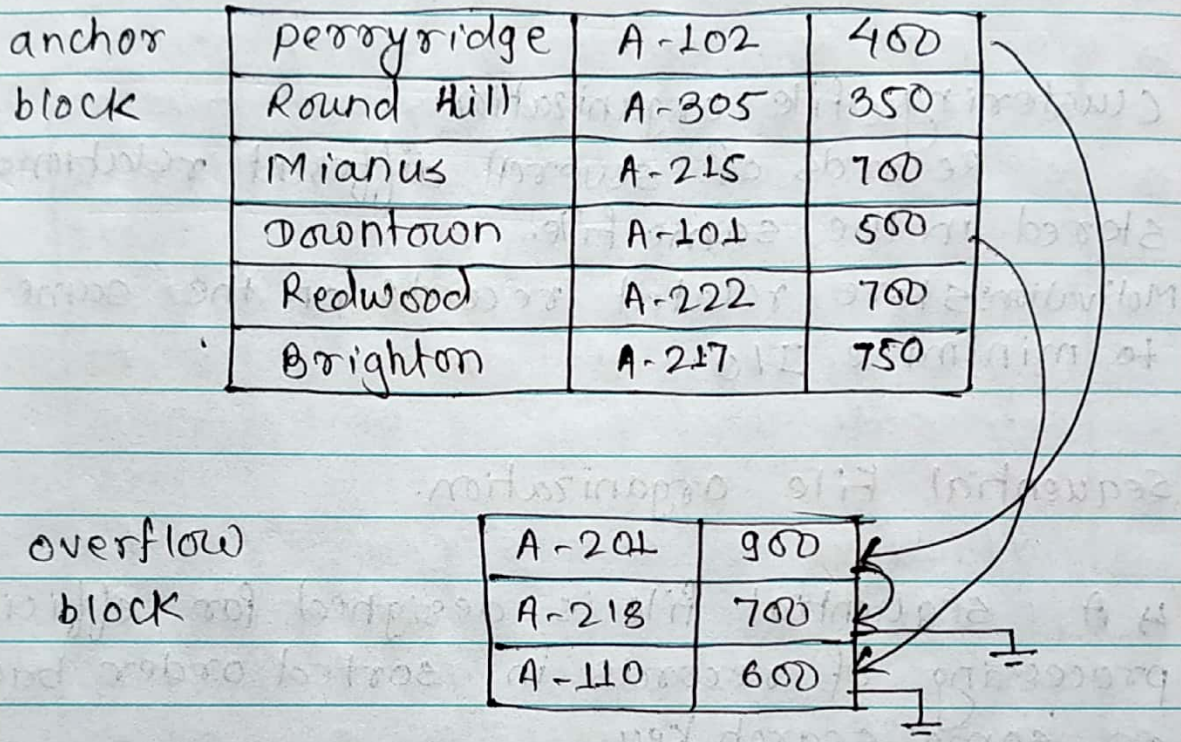


↳ The main disaduantage of the pointer structure is that space is wasted in all records except the first in a chain.

↳ Alternative solution (for fixed-length representation with pointer), which uses two different kinds of blocks in a file.

Anchor block : contains the first record of each chains.

overflow block: contains records other than those that are the first records of chains.



Organization of Records in Files.

Different ways to logically organize records in a file.

Heap file organisation.

A record can be placed anywhere in the file where there is space; there is no ordering in the file.

Sequential file organization.

Store records in sequential order, based on the value of the search key of each record.

Hashing file organization:

A hash function is computed on some attribute of each record; the result specifies in which block of the file, the record is placed.

Clustering file organization:

Records of several diffesent relations can be stored in the same file.

Motivation: store related records on the same block to minimize I/O.

Sequential File organization.

↳ A sequential file is designed for efficient processing of records in sorted order based on some search key.

↳ A search key is any attribute or set of attributes; it need not be the primary key, or even superkey.

↳ Suitable for applications that require sequential processing of the entire file.

↳ To permit fast retrieval of records in search-key order, we chain together records by pointers

↳ The pointer in each record points to the next record in search-key order.

↳ To minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search key order as possible.

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	

32222	Verdi	Music	480000	

Fig: Sequential file after an insertion.

↳ The sequential file organization allows records to be read in sorted order; that can be useful for display purpose, as well as for certain query processing algorithms.

↳ It is difficult however, to maintain physical sequen-

trial order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion.

↳ We manage deletion by using pointer chains, as we saw previously. For insertion, we apply the following rules:

i. Locate the record in the file that comes before the record to be inserted in search-key order.

ii. If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an overflow block. In either case, adjust the pointers so as to chain together the records in search-key order.

Clustering File Organization.

↳ Simple file structures stores each relation in a separate file.

↳ Clustering file organization stores several relations in one file.

Example: Clustering file of the two relations customer (name, street, city) and depositor

(name, account_num)

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

↳ Good for queries involving a join of depositor and customer, and for queries involving one single customer and his account.

↳ Bad for queries involving only customer.

↳ Results in variable size records

Indexing and Hashing.

Indexing.

↳ Indexing is used to optimize the performance of a database by minimising the number of disk accesses required when a query is processed.

↳ The index is a type of data structure. It is used to locate and access the data in database table quickly.

Index structure:

Indexes can be created using some database

columns.

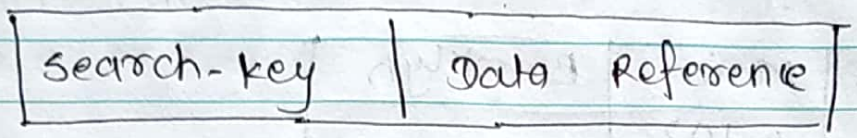


Fig: structure of Index.

↳ The first column of the database is the search key that contains a copy of the primary key or candidate key of the table.

↳ The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.

↳ The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particulas key can be found.

Index Evaluation Metrices.

Access type:
Finding records with specified attribute value.
Finding records whose attribute value falls in a specified range.

Access time.
Time to find particulas value.

Insestion time:
Time to insert new value as well as to update the index structure.

Deletion Time:
Time to delete data item as well as to update the index structure.

Space overhead:
Additional space occupied by an index structure

Indexing methods:

i. ordered indices
ii. Primary index
iii. Clustering index
iv. Secondary index.

ordered indices:

↳ In an ordered index, the indices are usually sorted to make seaching faster.

↳ The indices which are sorted are known as ordered indices.

Example:

Suppose we have an employee table with thousands of record and each of which is 10 bytes long. If their ids start with 1, 2, 3, ... and

so on and we have to search student with ID-543.

↳ In case of database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading 543*10 = 5430 bytes.

↳ In case of an index, we will search using indexes and the DBMS will read the record after reading 542*2 = 1084 bytes which are very less compared to previous case.

Primary Index.

↳ If the index is created on the basis of the primary key of the table, then it is known as primary indexing. These primary keys are unique to each and contain 1:1 relation between the records.

↳ As primary key are stored in sorted order, the performance of the searching operation is quite efficient.

↳ The primary index can be classified into two types: Dense index and sparse index.

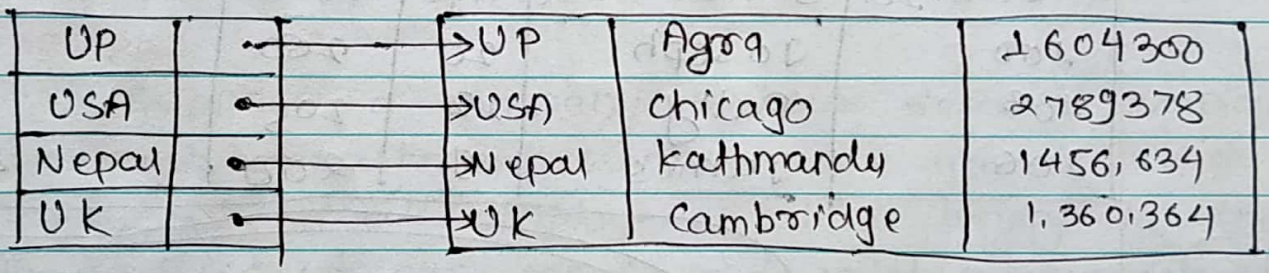
Dense Index:

The dense index contains an index record for

every seasch key value in the data file. It makes seaschipg faster.

↳ The number of records in the index table is same as the number of records in main table

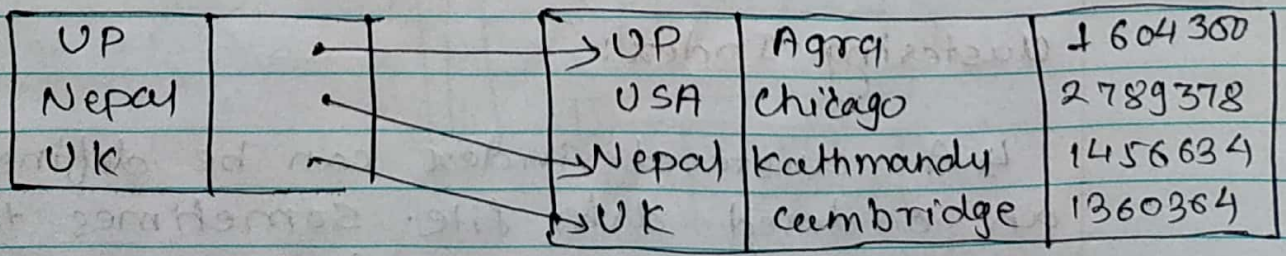
↳ It needs more space to store index record itself. The index records have the seasch key and a pointer to the actual record on the disk.



Sparse index:

↳ In the data file, index record appears only for few items. Each item points to a block.

↳ In sparse index, instead of pointing to each record in the main table, the index points to the records in the main table in a gap.



↳ In this method of indexing, range of index columns store the same data block address. And when data is to be retrieved, the block address will be fetched linearly till we get the requested data.

↳ The main goal of this method should be more efficient search with less memory space.

Student_Id	Student_Name	Age	Course	Index Address.
100	Joseph	20	200	
103	Patty	22	205	
106	James	19	200	

100	Joseph
101	Alien
102	Chris
103	Patty
104	Jack
106	James
107	Antony
108	Jacob

Clustering Index:

↳ A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns

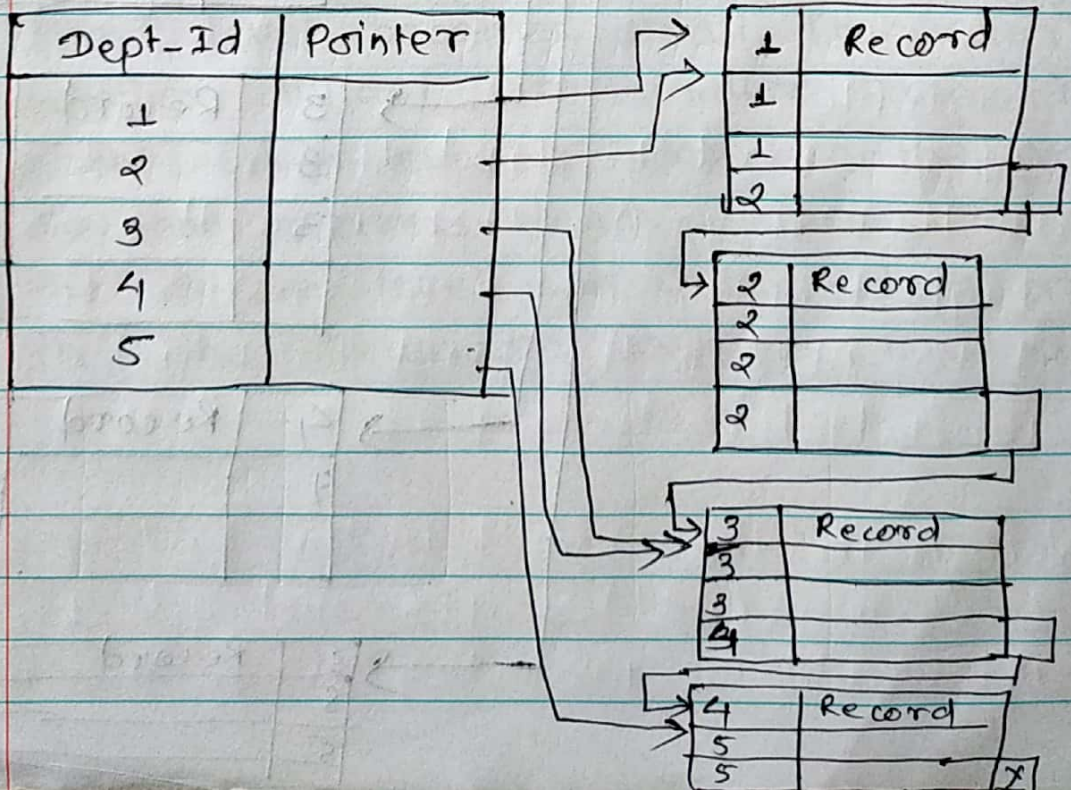
which maynot be unique for each record.

↳ In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.

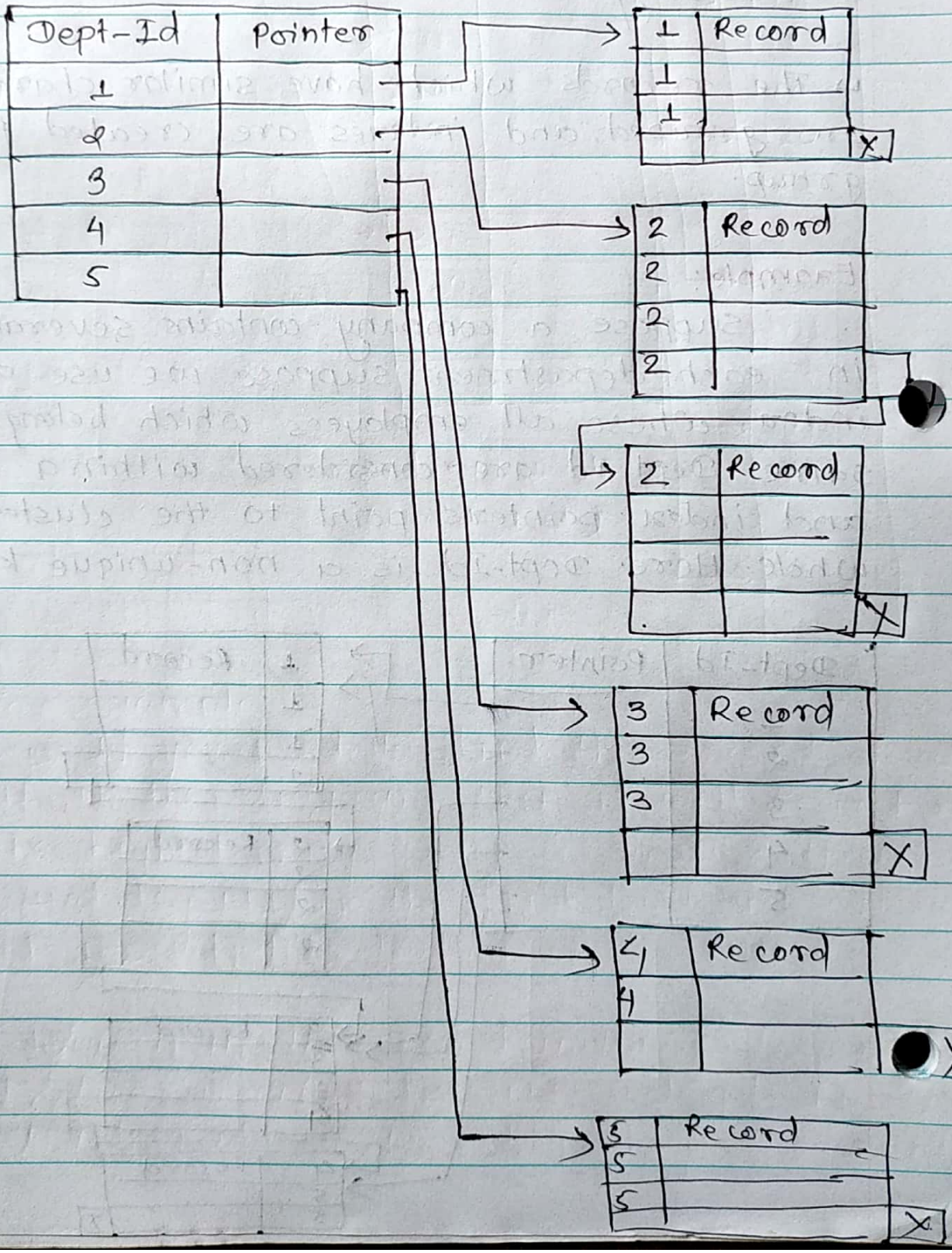
↳ The records which have similar characteristics are grouped, and indexes are created for these group.

Example:

Suppose a company contains several employees in each department. Suppose we use a clustesing index, where all employees which belong to the same Dept-Id are considered within a single cluster and index pointers point to the cluster as a whole. Here Dept-Id is a non-unique key.



The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate cluster, then it is called better technique.



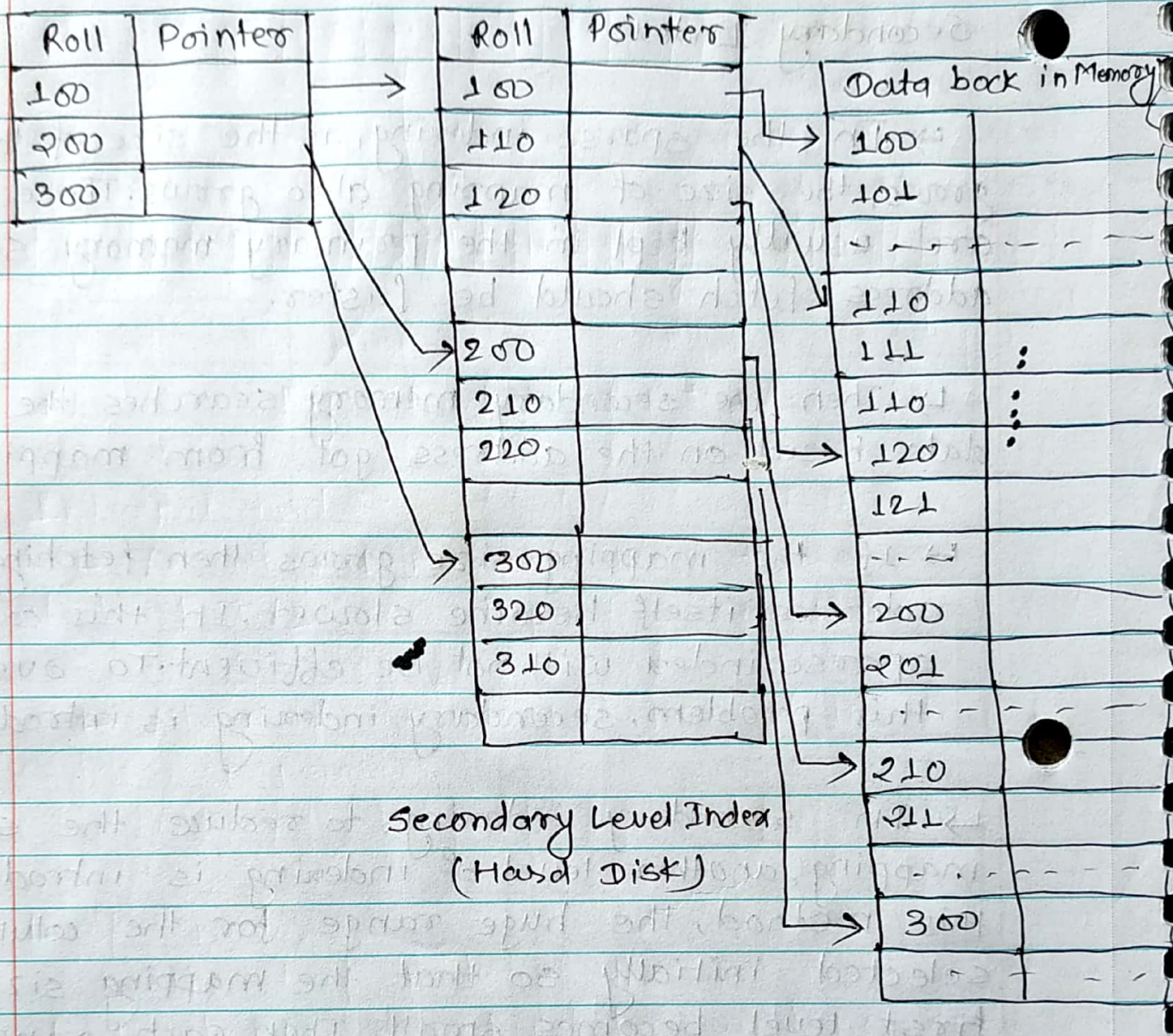
Secondary Index:

↳ In the sparse indexing, as the size of the table grows, the size of mapping also grows. These mapping are usually kept in the primary memory so that address fetch should be faster.

↳ Then the secondary memory searches the actual data based on the address got from mapping.

↳ If the mapping size grows then fetching the address itself become slower. In this case, the sparse index will not be efficient. To overcome this problem, secondary indexing is introduced.

↳ In secondary indexing, to reduce the size of mapping, another level of indexing is introduced. In this method, the huge range for the columns is selected initially so that the mapping size of the first level becomes small. Then each range is further divided into smaller ranges. The mapping of the first level is stored in the primary memory, so that address fetch is faster. The mapping of the second level and actual data are stored in the secondary memory (hard disk)



For example:

↳ If you want to find the record of all 111 in the diagram, then it will search the highest entry which is smaller than or equal to 111 in the first level index. It will get 100 at this level.

↳ Then in the second index level, again it does max(111) <= 111 and gets 110. Now using the address 110, it goes to the data

block and starts searching each record till it gets 111.

↳ This is how a search is performed in this method. Insesting, updating or deleting is also done in same manner.

B+ tree Index.

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

↳ In B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that leaf nodes remain at the same height; thus balanced.

↳ In B+ tree, the leaf nodes are linked using a linked list. Therefore, a B+ tree can support random access as well as sequential access.

↳ In B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.

↳ Supports equality and range searches, multiple attribute key and partial key searches.

↳ Responds to dynamic changes in the table.

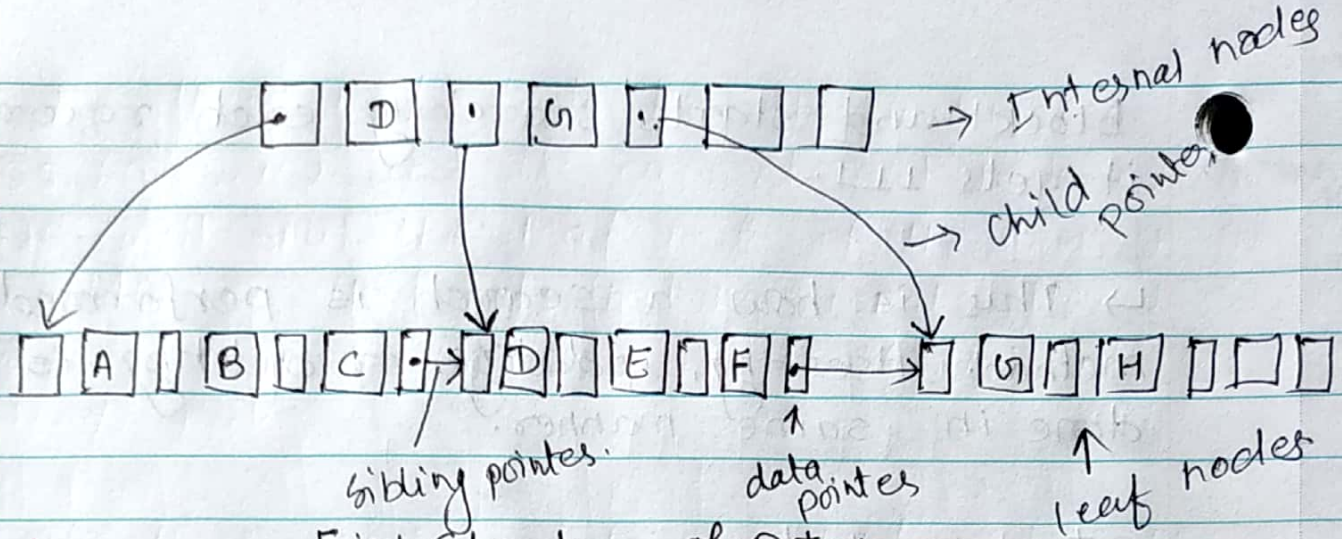


Fig: Structure of B+ tree

Internal nodes:

↳ An internal node of the B+ tree can contain at least n/2 record pointers except the root node

↳ At most, an internal node of the tree contains n pointers

Leaf node.

↳ The leaf node of the B+ tree can contain at least n/2 record pointers and n/2 key values

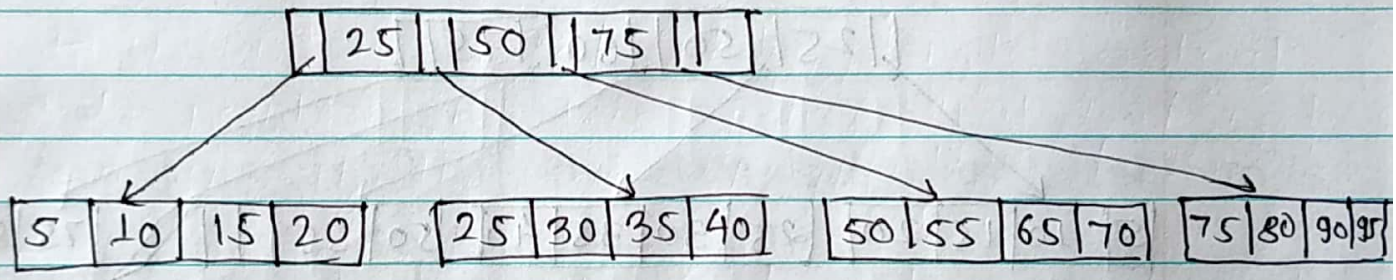
↳ At most, a leaf node contains n record pointer and n key values.

↳ Every leaf node of the B+ tree contains one block pointer P to point to next leaf node

Searching a record in B+ tree.

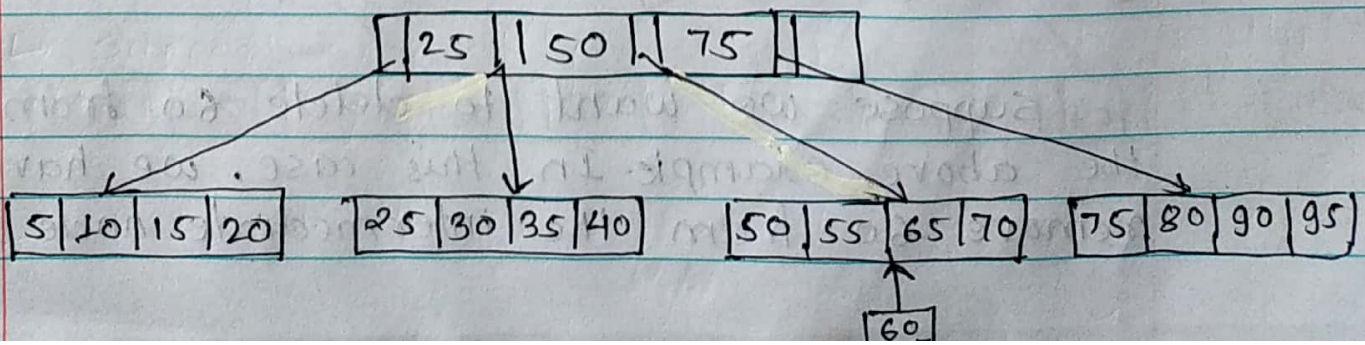
Suppose we have to search 55 in the below B+ tree structure. First we will fetch for the intermediary node which direct to the leaf node that can contain a record for 55.

So in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.



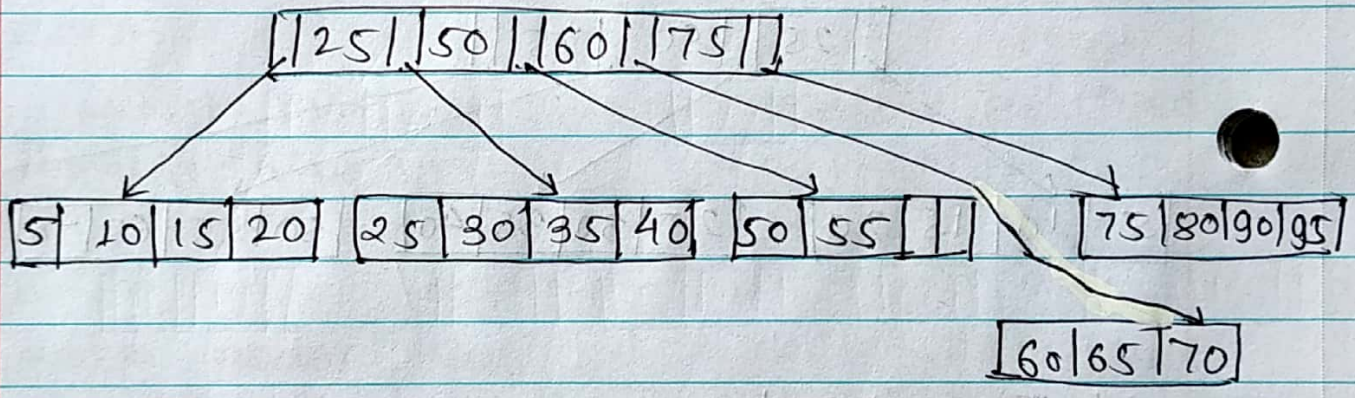
B+ Tree insertion

Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.



The third leaf node has the values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node of the tree in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes.

If these two has to be leaf nodes, the intermediate node cannot branch from 50. It should have 60 added to it, and then we can have pointers to new leaf node.



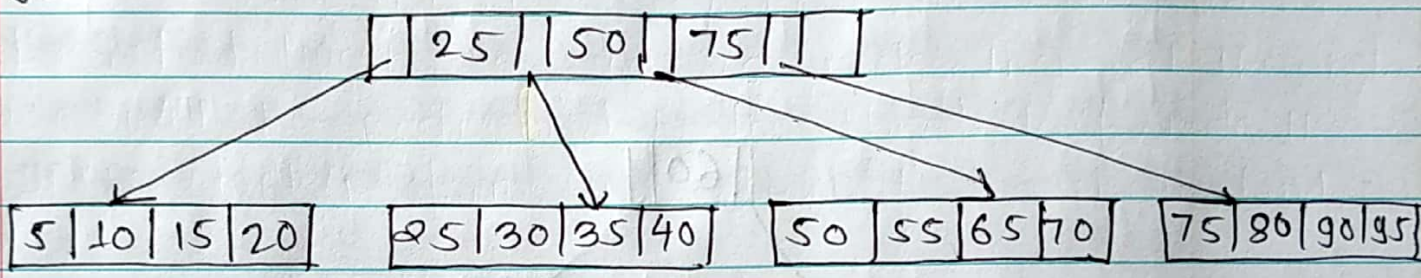
This is how we can insert an entry when there is overflow. In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node.

B+ Tree deletion

Suppose we want to delete 60 from the above example. In this case, we have to remove 60 from the intermediate node as

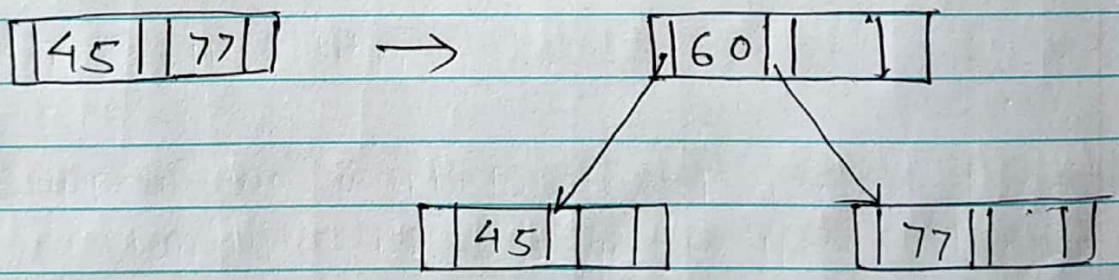
well as from 4th leaf node too. If we remove it from the intesmediate node, then the tree will not satisfy the rule of the B+ tree. So we need to modify it to have a balanced tree.

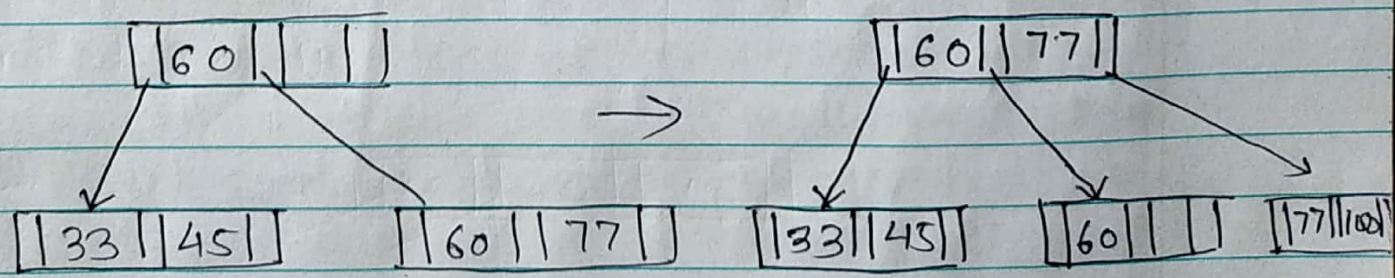
After deleting node 60 from above B+ tree and re-arranging the nodes, it will show as follows:

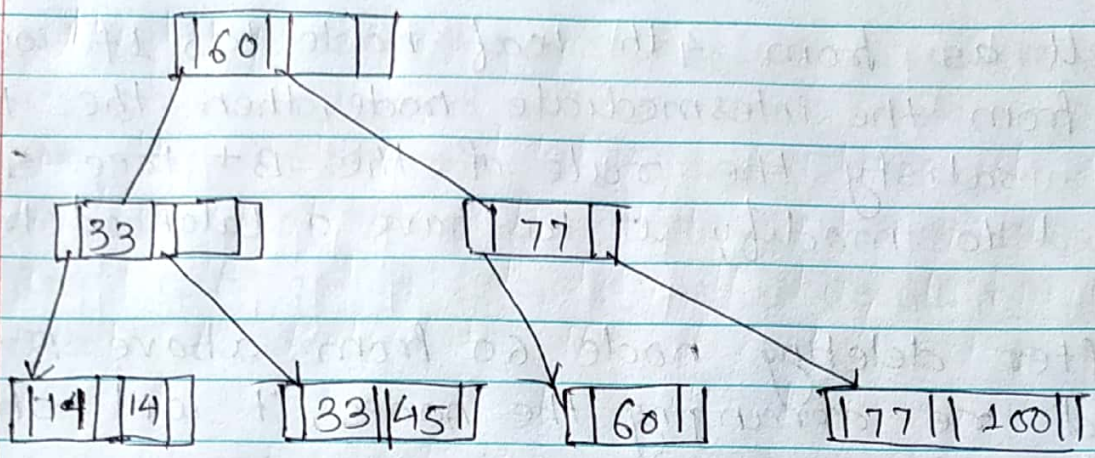


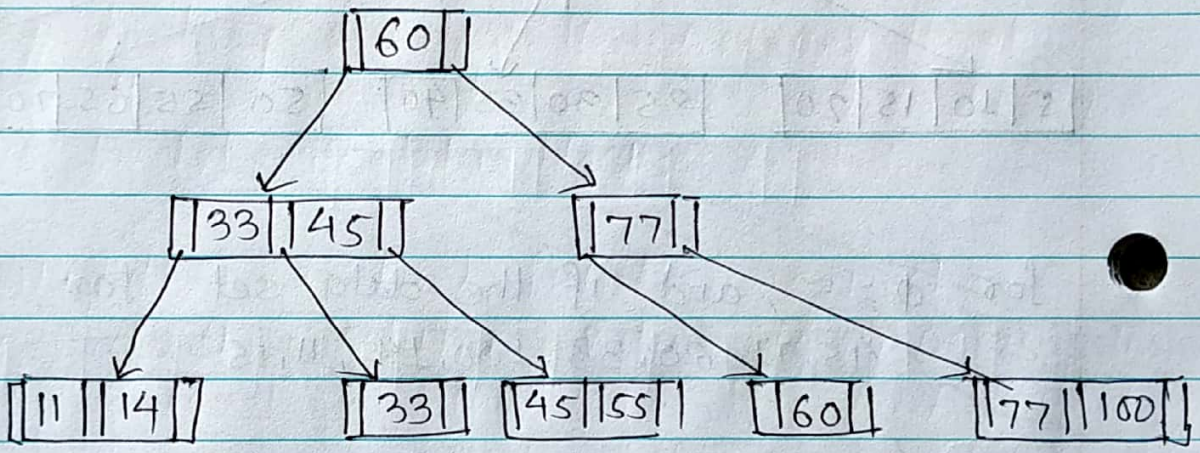
for $\phi = 3$ and if the data set for insestion is

45, 77, 60, 33, 100, 14, 11, 15









Hashing

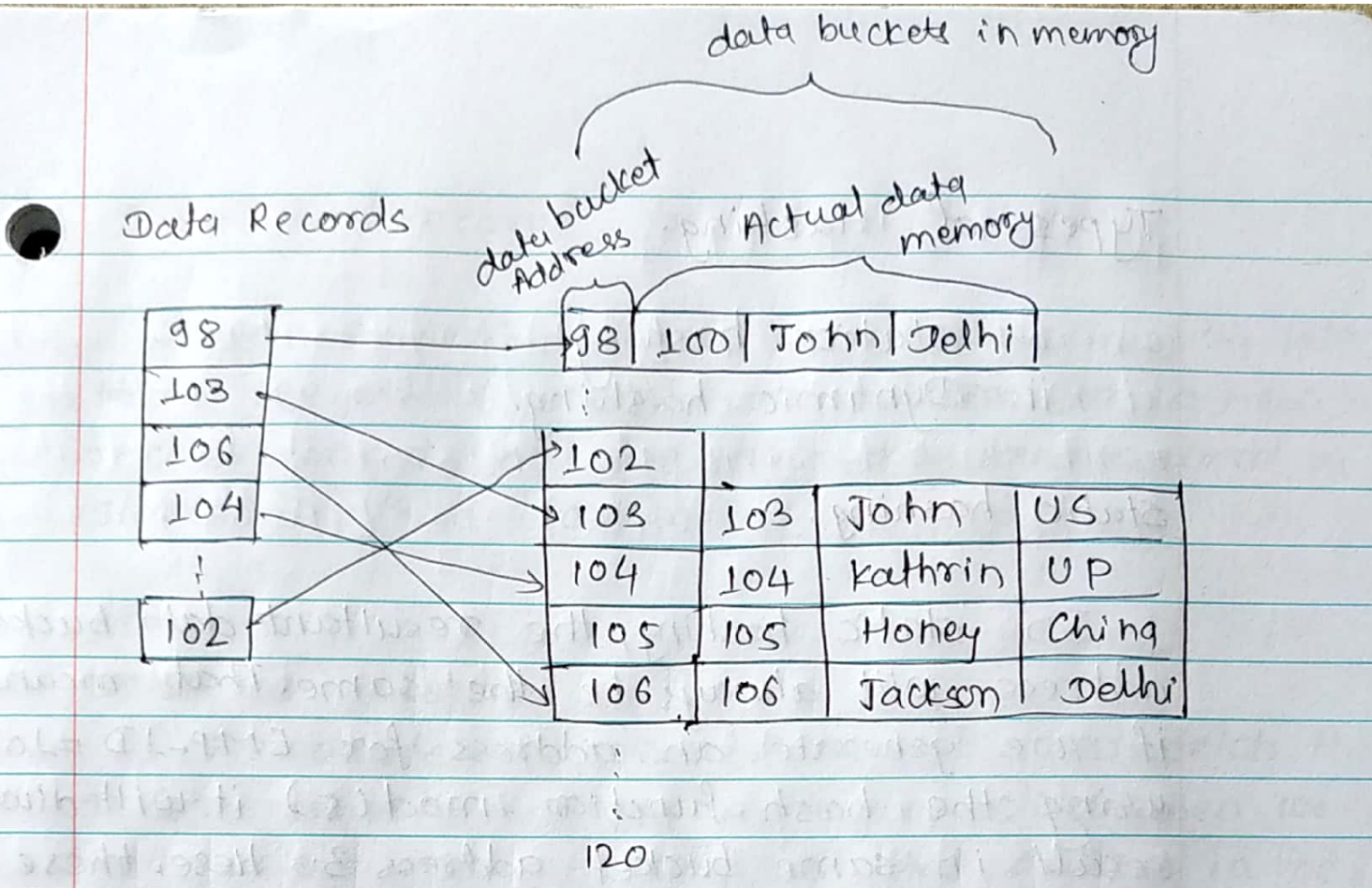
↳ In a huge database structure, it is very inefficient to search all the index values and reach the desired data. Hashing technique is used to calculate the direct location of a data record on the disk without using index structure.

↳ In this technique, data is stored at the data blocks whose address is generated by using hashing function. The memory location where these records are stored is known as data bucket or data blocks.

↳ In this, a hash function can choose any of the column value to generate the address. Most of the time, the hash function uses the primary key to generate the address of the data block.

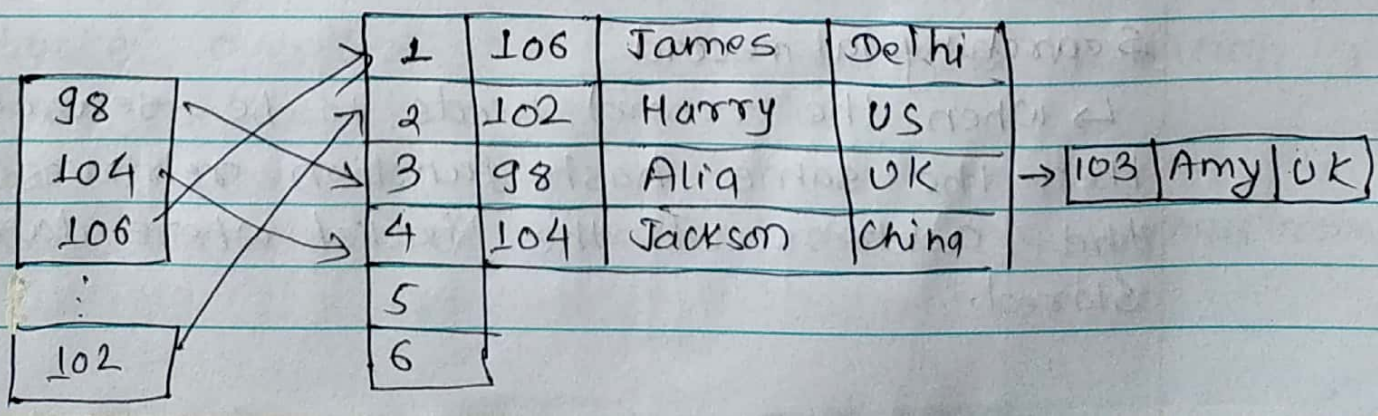
↳ A hash function is a simple mathematical function to any complex mathematical function.

↳ We can even consider the primary key itself as the address of the data block. That means each row whose address will be the same as a primary key stored in the data block.



↳ The above diagram shows data block addresses as primary key value. This hash function can also be a simple mathematical function like exponential, mod, cos, sin etc.

↳ Suppose we have mod(5) has function to determine the address of the data block. In this case, it applies mod(5) hash function on the primary key and generates 3, 3, 1, 4 and 2 respectively, and records are stored in those data block addresses



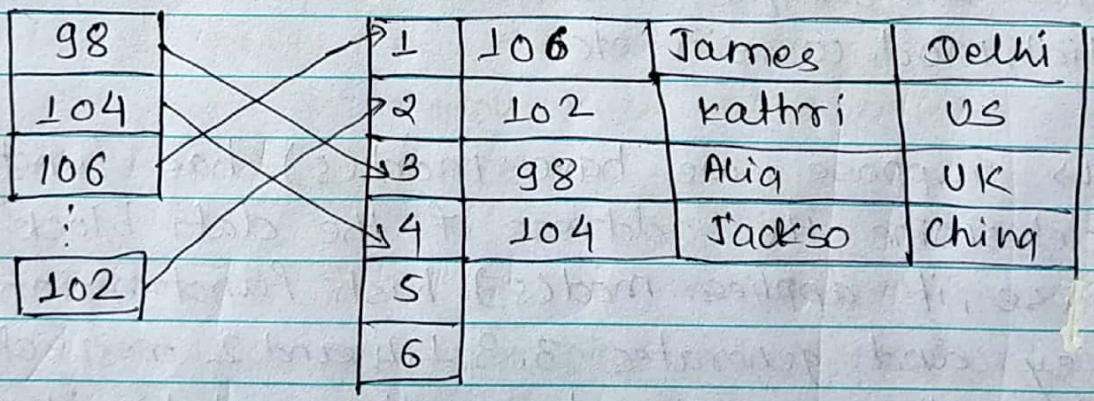
Types of Hashing.

i. static hashing
ii. Dynamic hashing.

static hashing.

In static hashing, the resultant data bucket address will always be the same. That means if we generate an address for EMP_ID = 103 using the hash function mod(5) it will always result in same bucket address 3. Hese, these will be no change in the bucket address.

Data Records



Operations of Static Hashing.

Searching a record.

↳ When the record needs to be seasched, then the same hash function retrieves the address of the bucket where data is stored.

Insert a Record.

When a new record is inserted into the table, then we will generate an address for a new record based on the hash key and record is stored in that location.

Delete a record.

To delete a record, we will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in the memory.

Update a Record.

To update a record, we will first search it using a hash function. and then the data record is updated.

If we want to insest some new record into the file but the address of a data bucket genesated by the hash function is not empty, or data alseady exists in that address. This situation in the static hashing is known as bucket ovesflow. This is a critical situation in this method.

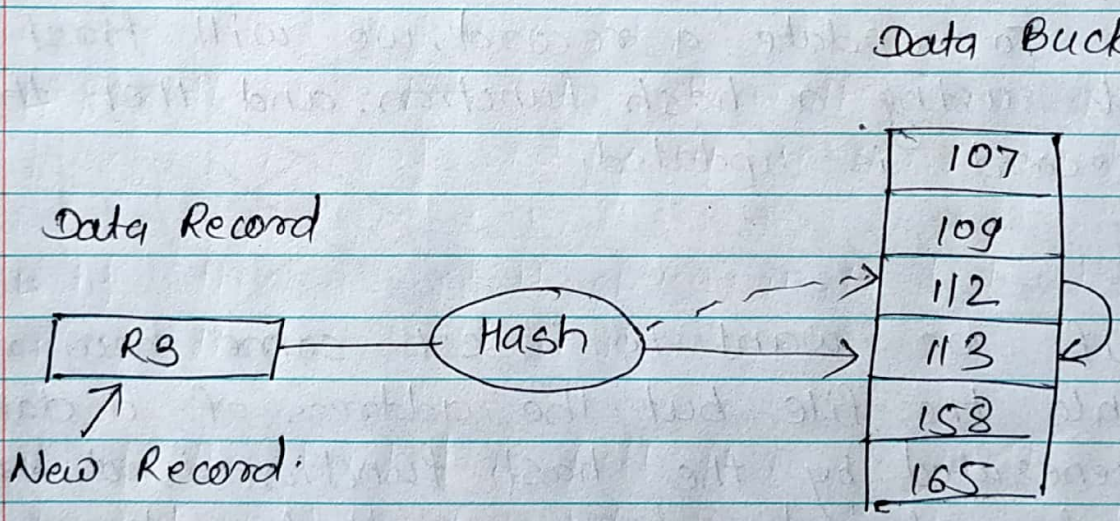
To overcome this situation, there are various methods

i. Open Hashing.

When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as Linear probing.

For example:

Suppose R3 is new address which needs to be inserted, the hash function generates address 112 for R3 But the generated address is already full. So system seaches next available data bucket, 113 and assigns R3 to it.



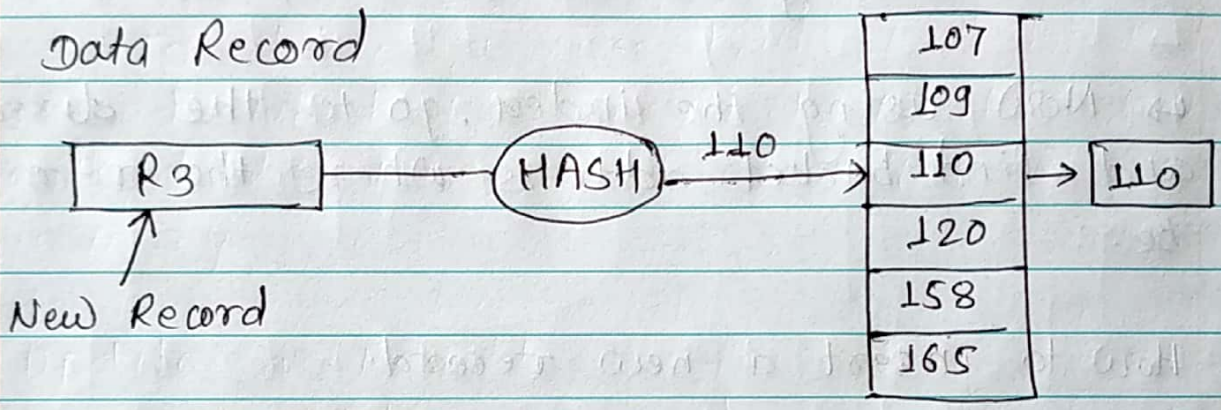
ii. Close hashing.

When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism

is also known as Overflow chaining.

For example:

Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store new data. In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.



Dynamic Hashing:

↳ The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.

↳ In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.

↳ This method makes hashing dynamic i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key.

↳ First, calculate the hash address of the key.

↳ Check how many bits are used in the directory, and these bits are called as i.

↳ Take the least significant i bits of the hash address. This gives an index of the directory.

↳ Now using the index, go to the directory and find bucket address where the record might be.

How to insert a new record.

↳ Firstly, we have to follow the same procedure for retrieval, ending up in some bucket.

↳ If there is still space in that bucket, then place the record in it.

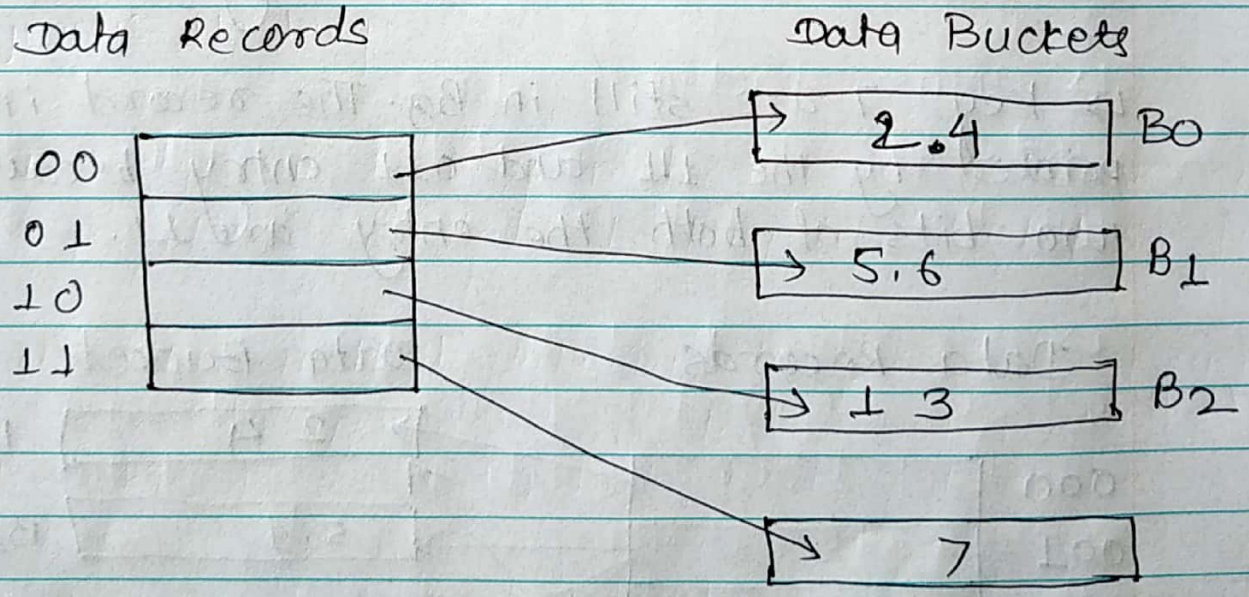
↳ If the bucket is full, then we will split the bucket and redistribute the records.

For example:

Consider the following grouping of keys into buckets, depending on the prefix of their hash address.

Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

The last two bits of 2 and 4 are 00. So it will go into bucket B0. The last two bits of 5 and 6 are 01, so it will go into bucket B1. The last two bits of 1 and 3 are 10, so it will go into bucket B2. The last two bits of 7 are 11, so it will go into B3.



Insert key with hash address 10001 into the above structure.

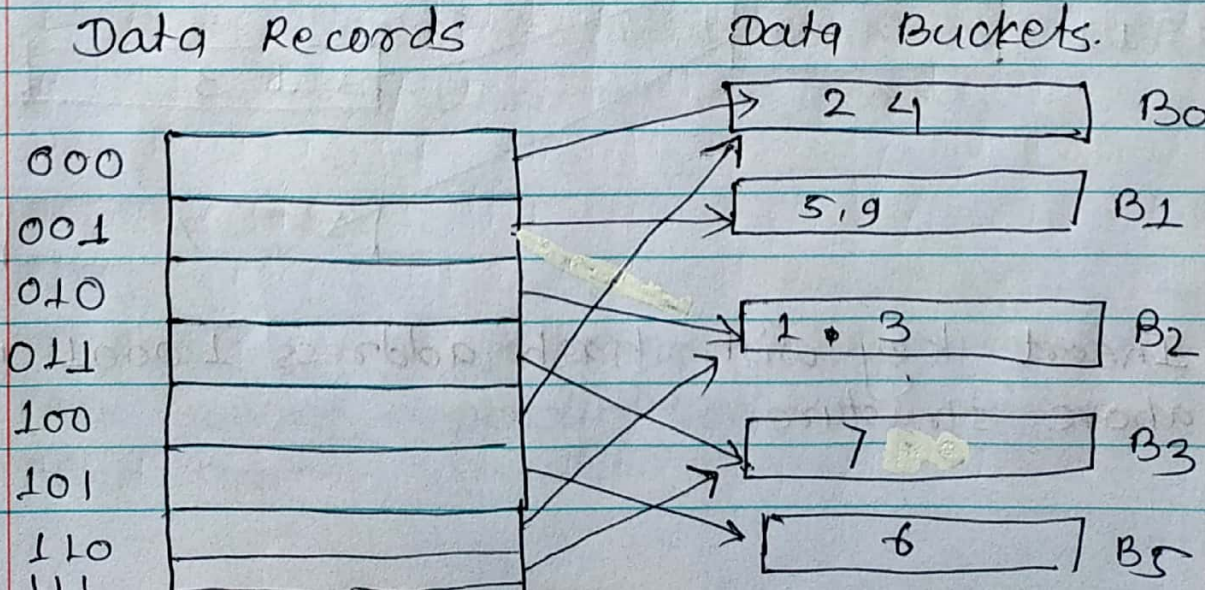
↳ Since key 9 has has address 10001, it must go into first bucket B_1 is full. So it will get split.

↳ The spliting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B_1, and the last three bits of 6 are 101, so it will go into Bucket B_5.

↳ keys 2 and 4 are still in B_0. The record in B_0 pointed by the 000 and 100 entry because last two bits of the both entry are 00.

↳ keys 1 and 3 are still in B_2. The record in B_2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.

↳ key 7 are still in B_3. The record in B_3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.



Advantages of dynamic hashing.

↳ In this method, the performance doesnot decrease as the data grows in the system. It simply increases the size of memory to accomodate the data.

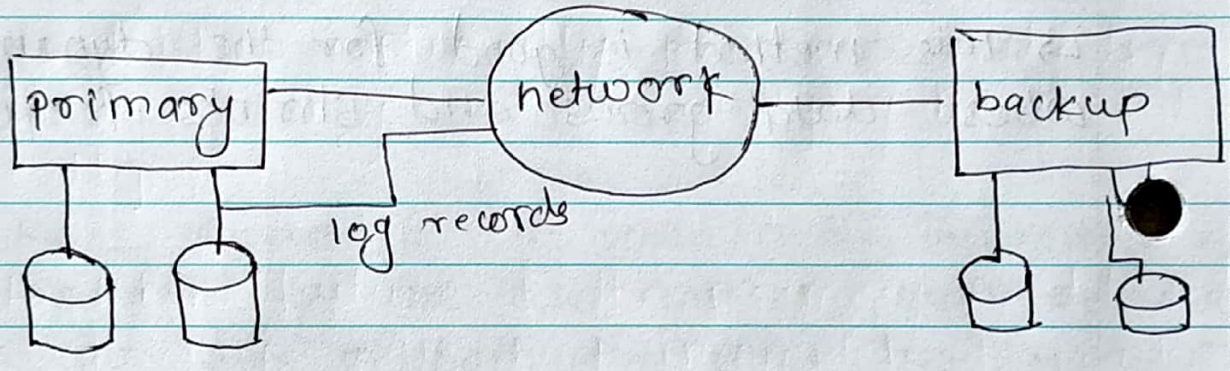
↳ In this method, memory is well utilized as it grows and shrink with the data. There will not be any unused memory lying.

↳ This method is good for the dynamic database whose data grows and shrinks frequently.

Remote Backup Systems.

↳ Remote backup provides a sense of security in case the primary location where the database is located gets destroyed.

↳ Remote backup can be offline or real-time or online. In case it is offline it is maintained manually.



↳ Online backup systems are more real-time and lifesavers for database administrators & investors.

↳ An online backup system is a mechanism where every bit of the real time data is backed up simultaneously at two distant places. One of them is directly connected to the system and the other one is kept at a remote place as backup.

↳ As soon as primary database storage

fails, the backup system senses the failure and switches the user system to the remote storage. Sometimes this is so instant that the users can't even realize a failure.

Detection of failure.

↳ Backup site must detect when primary site has failed.

↳ To distinguish primary site failure from link failure maintain several communication links between the primary and remote backup.

Transfer of control

↳ To take over control backup site first perform 'recovery using its copy of the database and all the long records it has received from the primary.

↳ Thus completed transactions are redone and incomplete transactions are rolled back.

↳ When the backup site takes over processing it becomes the new primary.

↳ To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.

Time to recover:

To reduce delay in takeoves, backup site periodically processes the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.

Hot-spare:

↳ Hot spare configuration permits very fast takeover.

↳ Backup continually processes redo log record as they arrive, applying the updates locally.

↳ When failure of the primary is detected the backup rolls back incomplete transactions and is ready to process new transactions.

Alternative to remote backup:

↳ Distributed database with replicated data.

↳ Remote backup is faster & cheaper but less tolerant to failure.

Ensure dusability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degree of dusability.

One safe:

↳ Commit as soon as transaction's commit log record is written at primary.

↳ Problem: updates maynot arrive at backup before it takes over.

Two-very-safe

↳ Commit when transactions' commit log record is written at primary and backup.

↳ Reduces availability since transactions cannot commit if either site fails.

Two-safe.

↳ Proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as it commit log record is written at the primary.

↳ Bettes availability than two-very safe; avoids problem of lost transactions in one-safe.