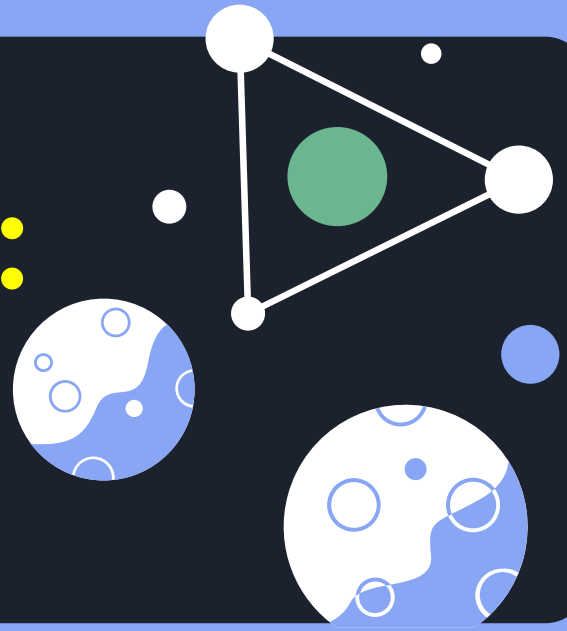
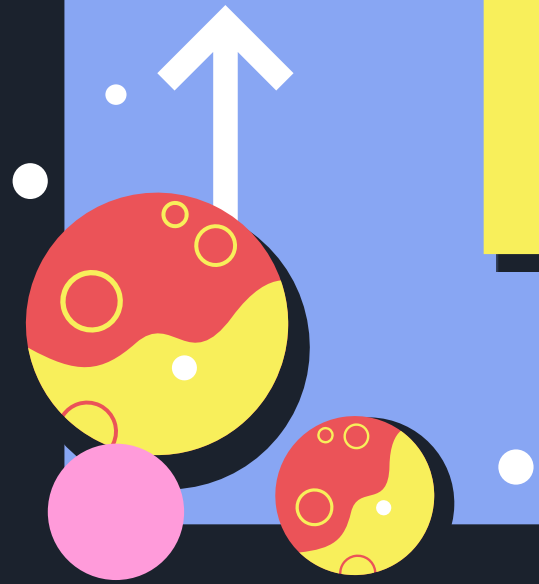
STORING DATA: DISKS & FILES



<Himanshu Kumar Singh>
<Kapil>

A memory is what is left when something happens and does not completely un-happen.

Table of contents

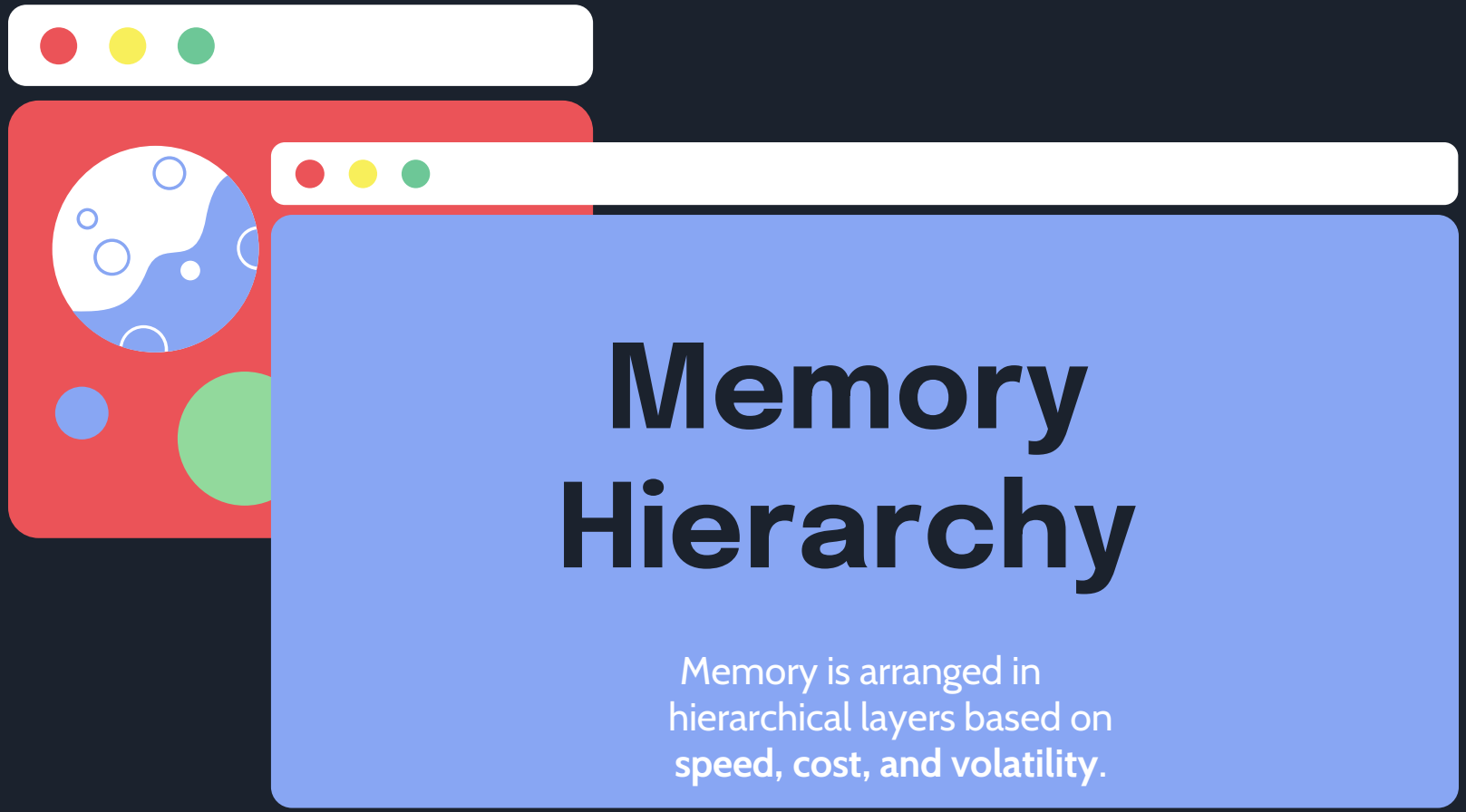


01

Data on External Storage

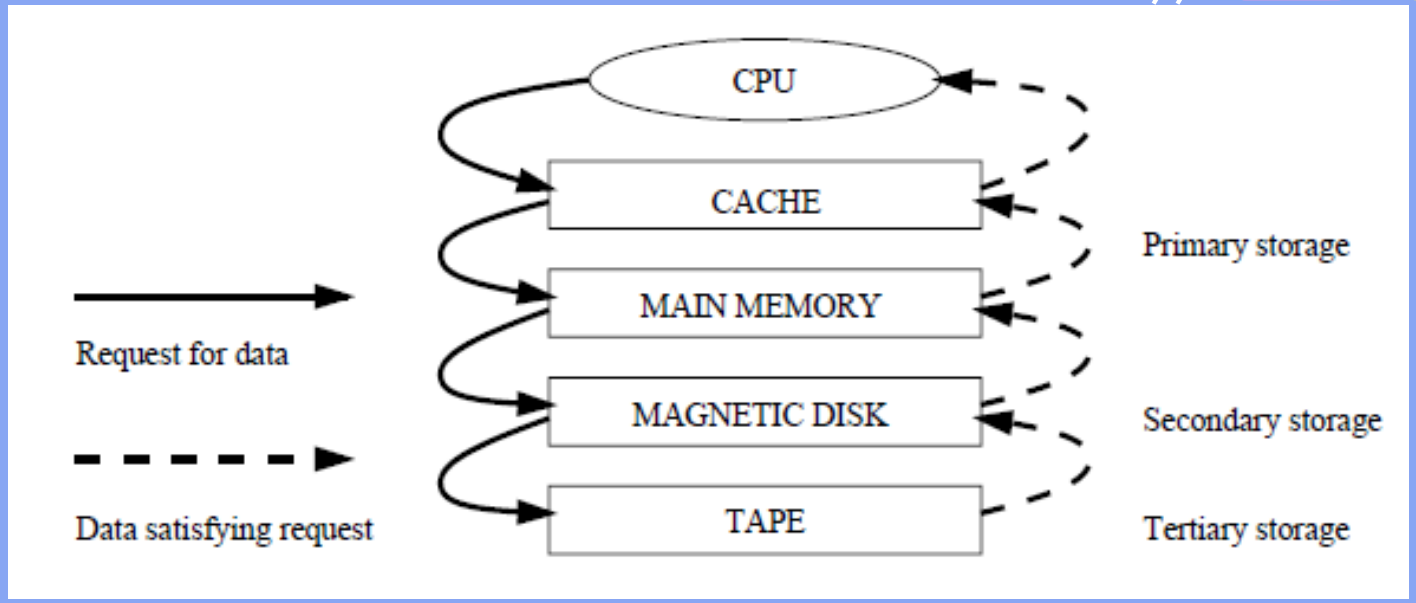
Used when data is too large for main memory.

- Data is stored on external storage devices such as disk , tapes , and fetched into main memory as needed for processing.
- The unit of information read from or written to disk is a page. (size = 4KB or 8KB)
- The cost of I/O dominates the cost of typical database operations, and database system are carefully optimized to minimize this cost.



Memory Hierarchy

Memory is arranged in hierarchical layers based on **speed, cost, and volatility**.


CPU
CACHE
MAIN MEMORY
MAGNETIC DISK
TAPE
Primary storage
Secondary storage
Tertiary storage
Request for data
Data satisfying request

Primary Storage

- Includes **cache** and **main memory**
- **Very fast** access
- **Volatile** (data lost on shutdown unless battery-backed)
- Expensive – ~100x the cost of the same disk space

Secondary Storage

- Consists of **magnetic disks**
- **Slower than main memory** but cheaper
- **Nonvolatile** – retains data after shutdown
- Essential due to:
- **High data volume**
- **Cost constraints** of buying large main memory

Tertiary Storage

- Includes **optical disks** and **tapes**
- **Very inexpensive**, used for **archival storage**
- Best for **rarely accessed** data
- Also **nonvolatile**

Why Secondary/Tertiary Storage?

Cost efficiency:

Main memory is too expensive for all data.

Addressing Limitations:

32-bit systems can only address **2^{32} bytes** (~4GB)

Persistence

Persistent storage across system restarts

Long-Term Storage:

Ideal for **long-term data maintenance**

Scalability

Easier to expand storage capacity using disks or tapes than with RAM

Tapes ?

Example: Quantum DLT 4000

Stores **20 GB** (up to **40 GB** with compression).
Uses **128 tracks.**
Transfer rate:
1.5 MB/sec (uncompressed)
3.0 MB/sec (compressed)
Can stack up to 7 tapes: max **~280 GB compressed.**

Advantages:

- **Large capacity**
- **Low cost**
- Great for **backup and archival**

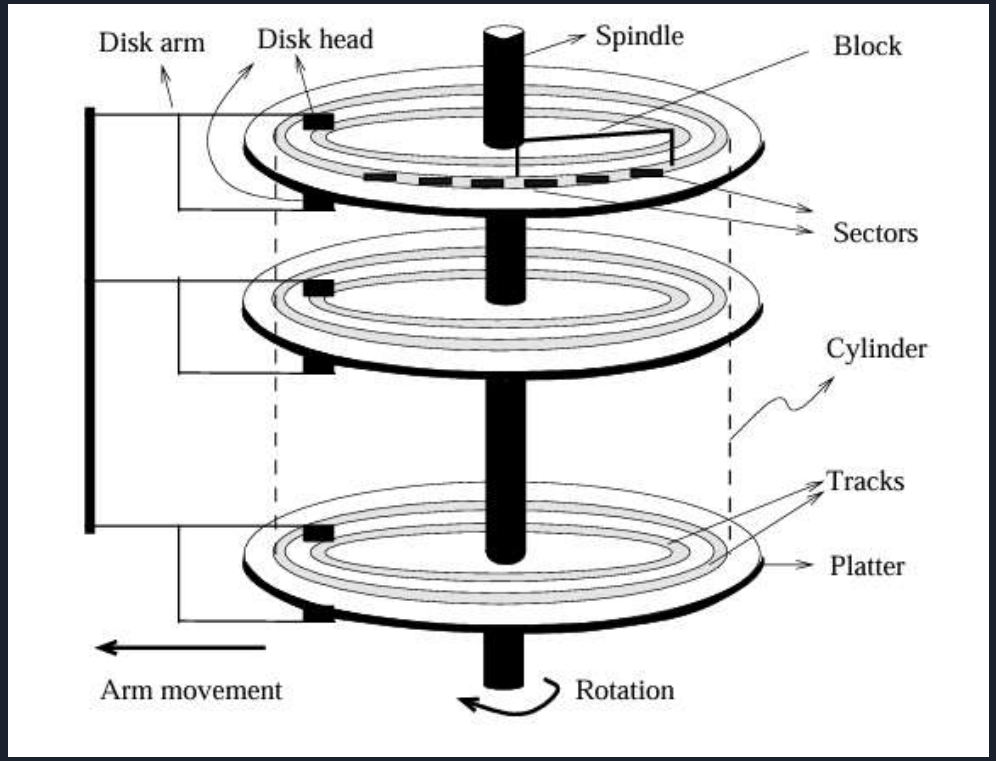
Disadvantages:

- **Sequential access** only.
- Very **slow for random access.**
- Not suitable for **frequent or operational data use.**

The main drawback of tapes is that they are sequential access devices. We must essentially step through all the data in order and cannot directly access a given location on tape. For example, to access the last byte on a tape, we would have to wind through the entire tape first. This makes tapes unsuitable for storing operational data, or data that is frequently accessed. Tapes are mostly used to back up operational data periodically.

Magnetic Disks

Magnetic disks support direct access to a desired location and are widely used for database applications.


Disk arm
Disk head
Spindle
Block
Sectors
Cylinder
Tracks
Platter
Arm movement
Rotation

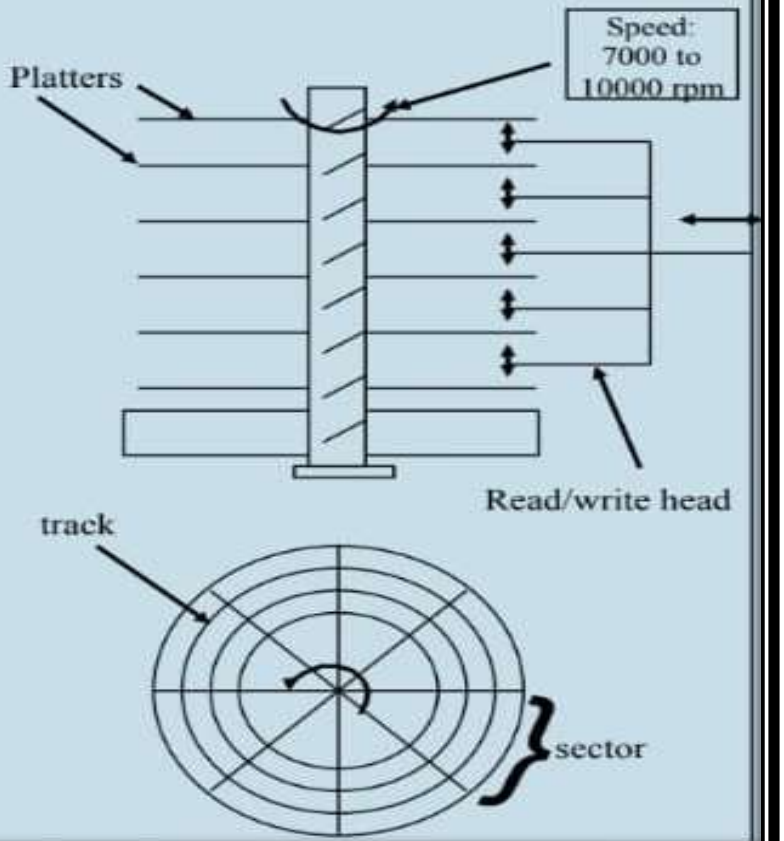
Disk Structure & Terminology

- **Disk Block**: Basic unit of data storage and transfer on disk (a contiguous sequence of bytes).
- **Tracks**: Concentric rings on the platter where blocks are stored.
- **Sectors**: Fixed-size arcs within a track; smallest addressable unit.
- **Platter**: Circular disk where data is stored (can be single or double-sided).
- **Cylinder**: All tracks aligned vertically across platters.
- **Disk Head**: Reads/writes data; one per surface.
- **Disk Arm**: Moves the heads across tracks.
- **Spindle**: Rotates the platters.

Structure of Disks

Disk

- several platters stacked on a rotating spindle
- one read / write head per surface for fast access
- platter has several tracks
  - ~10,000 per inch
- each track - several sectors
- each sector/track - blocks
- unit of data transfer - block
- cylinder i - track i on all platters
- sectoring is optional
- block – ½ KB to 8KB
  - fixed; set at initialization time



Data Transfer from Disk

Address of a block: Surface No, Cylinder No, Block No

Data transfer:

Move the r/w head to the appropriate track

- time needed - seek time – ~ 12 to 14 ms

Wait for the appropriate block to come under r/w head

- time needed - rotational delay - ~3 to 4ms (avg)

Access time: Seek time + rotational delay

Blocks on the same cylinder - roughly close to each other

- access time-wise

- cylinder i, cylinder (i + 1), cylinder (i + 2) etc.

Access Mechanics

- **Direct Access**: Supports random access to any disk location.
- **Disk Controller**: Interfaces between the disk and the computer.
- **Data Access Steps**:
  - **Seek Time**: Move head to the correct track.
  - **Rotational Delay**: Wait for block to rotate under the head.
  - **Transfer Time**: Read/write block data.

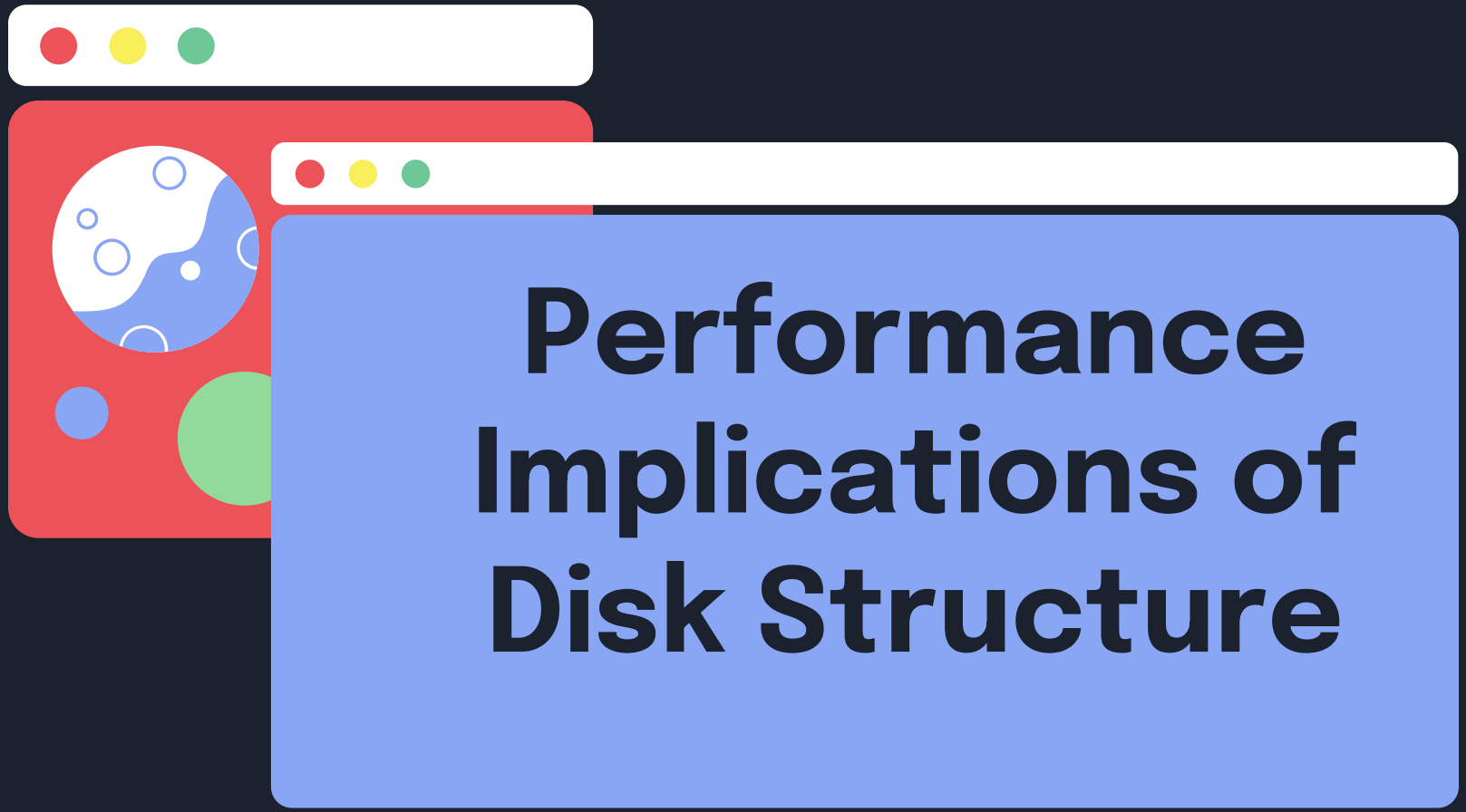
Parallelism Limitation

- All disk heads move together, but typically **only one head is active at a time**.
- Full parallel reading/writing is rare due to alignment difficulties.

Data Integrity

- **Checksums** used to detect sector corruption.
- Controller retries reads if checksum mismatch is detected.

An example of a current disk: The IBM Deskstar 14GPX. The IBM Deskstar 14GPX is a 3.5 inch, 14.4 GB hard disk with an average seek time of 9.1 milliseconds (msec) and an average rotational delay of 4.17 msec. However, the time to seek from one track to the next is just 2.2 msec, the maximum seek time is 15.5 msec. The disk has five double-sided platters that spin at 7,200 rotations per minute. Each platter holds 3.35 GB of data, with a density of 2.6 gigabit per square inch. The data transfer rate is about 13 MB per second. To put these numbers in perspective, observe that a disk access takes about 10 msecs, whereas accessing a main memory location typically takes less than 60 nanoseconds!


Performance
Implications of
Disk Structure

- **Data must be in main memory** for the DBMS to process it.
- **Disk ↔ Memory transfers happen in blocks**, even if only one item is needed.
- Each block transfer = **I/O operation (input/output)**.
- **Total Access Time = Seek Time + Rotational Delay + Transfer Time**
- Disk I/O time is often the **dominant cost** in DB operations.

Optimizing Data Placement :

- **Goal**: Reduce access time by **placing related data close together**.
- **Best case**: Related records in the **same block** (read in a single I/O).
- **Next best** (in decreasing closeness):
  - Same **track**
  - Same **cylinder**
  - **Adjacent cylinder**

Access Time

Access time depends on 3 things:

Access Time = Seek Time + Rotational Delay + Transfer Time

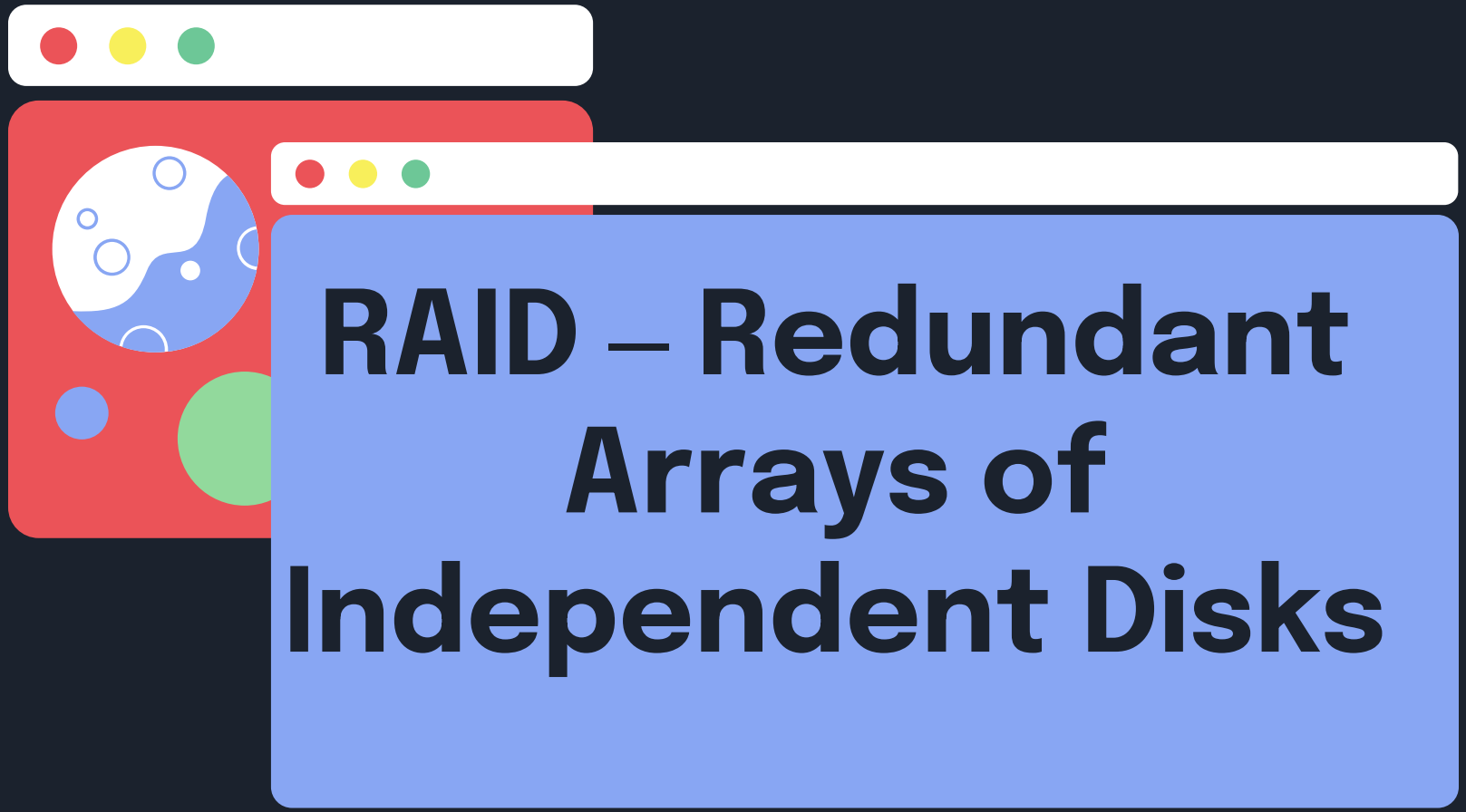
- **Seek Time**: Time to move the disk head to the right track.
- **Rotational Delay**: Time waiting for the block to rotate under the head.
- **Transfer Time**: Time to actually read/write the block.

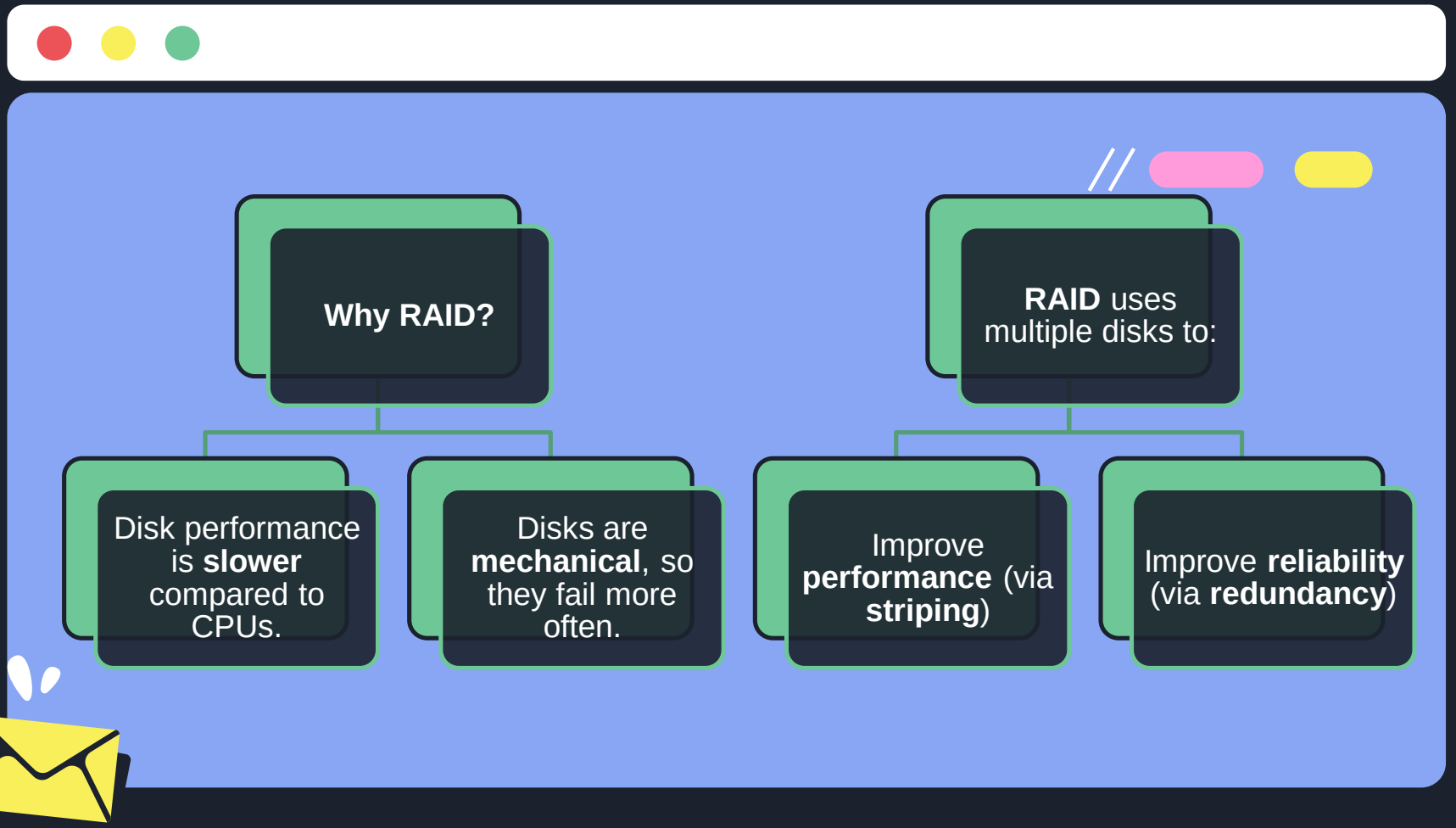
Disk Access Mechanics

•Entire **track can be read in one revolution**.
•Heads switch to read **other tracks in the same cylinder** (no arm movement needed).
•After a cylinder is processed, the arm moves to the **next cylinder**.

Sequential vs. Random Access

•**Sequential access** is **much faster** than random access.
    Minimizes **seek time** and **rotational delay**.
    Ideal for scans and ordered retrievals.
•Organizing data **sequentially on disk** greatly improves performance.


RAID – Redundant Arrays of Independent Disks


Why RAID?
Disk performance is **slower** compared to CPUs.
Disks are **mechanical**, so they fail more often.
RAID uses multiple disks to:
Improve **performance** (via **striping**)
Improve **reliability** (via **redundancy**)

Data Striping

- Data is split into **equal-sized parts** (striping units) and spread across multiple disks.
- **Small striping unit (bit/byte)** = good for large requests but poor for small ones.
- **Large striping unit (block)** = allows **parallel reads/writes**, better for small requests.

Redundancy

- More disks = higher failure chances.
- **Redundant data** is stored to recover from disk failures.
- Two main choices:
    - **Where** to store redundant data (one or all disks).
    - **How** to compute it (parity, Hamming codes, etc.).
- **Parity**: Stores 1 bit per group of data bits to detect and recover a single disk failure.

RAID Levels

RAID 0 – Striping Only (No Redundancy)

- High performance, **no fault tolerance**.
- 100% space utilization.
- Best write performance, but if one disk fails → all data lost.

RAID 1 – Mirroring

- Each disk has an **identical copy** (mirror).
- **Very reliable**, can serve **parallel reads**.
- Costly (only 50% space used).

RAID 0+1 or 10 – Striping + Mirroring

- Combines **performance of RAID 0** + **reliability of RAID 1**.
- Still 50% space utilization.
- Good for write-heavy applications.

RAID 2 – Bit Striping + Hamming Code

- Uses **error-correcting codes**.
- Good for **large reads**, bad for small ones.
- Rarely used.

RAID 3 – Bit Striping + Single Parity Disk

- One **parity disk** to recover from 1 failure.
- Works well for **large sequential reads**, not small ones.

RAID 4 – Block Striping + Single Parity Disk

- Read small blocks from one disk.
- Writes are slower due to **check disk bottleneck**.

RAID 5 – Block Striping + Distributed Parity

- **Most popular** RAID level.
- Parity is spread across all disks.
- **No single bottleneck**, supports parallel reads/writes.
- Good for **general use**.

RAID 6 – Block Striping + Double Parity

- Uses **Reed-Solomon codes**.
- Can survive **2 disk failures**.
- Slightly slower small writes due to double parity update.
- Ideal for **large, critical systems**.

Choosing a RAID Level

RAID	Use When...
RAID 0	Speed is more important than safety
RAID 1 / 10	High reliability is needed, cost is not an issue
RAID 5	Best overall balance (speed + reliability)
RAID 6	You need to tolerate 2 failures
RAID 3/4	Mostly used in older or specific systems


Disk Space
Management

- DBMS uses a **Disk Space Manager** to manage disk storage.
- Works with **pages** (same size as a disk block).
- Supports:
- Allocate / Deallocate a page
- Read / Write a page

Why Use Pages?

Transfers between memory and disk happen in **blocks**.
So DBMS uses **pages = blocks** for efficient I/O

Sequential Access Optimization

- Frequently accessed data is stored in **contiguous blocks**.
- Improves **performance** during scans or bulk reads.
- Disk Space Manager handles this for upper DBMS layers.

Keeping Track of Free Blocks

When data is added or removed, the DBMS needs to manage **which disk blocks are free or in use**. Here are two main methods to track this:

1. Free List

- A **linked list** of free blocks.
- When a block is freed (e.g., deleted record), it's added to the list.
- A **pointer to the first free block** is saved on disk.
- **Simple**, but not efficient for finding **contiguous free blocks**.

2. Bitmap

- Uses **1 bit per block**:
  - 1 = used
  - 0 = free
- Fast to find a group of free blocks **together**.
- Better for **efficient allocation** of large continuous space.

Disk Space Management :

- Should OS service be used ?
- Should RDBMS manage the disk space by itself ?

•2nd Option is preferred as RDBMS requires complete control over when a block or page in main memory buffer is written to the disk.

•This is important for recovering data when the system crash occurs.

Why Not Always Use OS Files?

1. Portability:
- DBMS must work on different OS platforms → can't rely on **OS- specific features**.

2. File Size Limits:
- Some OS (especially **32-bit systems**) limit files to **4 GB** → DBMS may need larger files.

3. Multi-disk Support:
- OS files usually **can't span multiple disks**, but DBMS often needs that.

Thank-You

02

File Organization

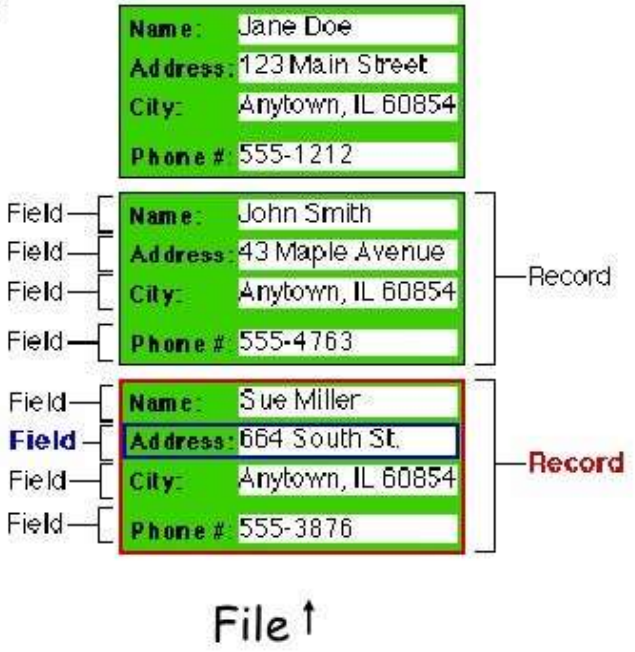
How records are stored in file.

Databases are made up of

-files

-records

-fields



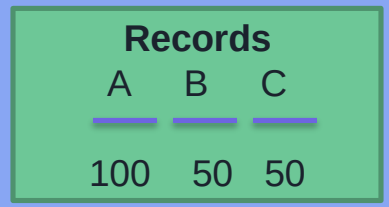
•Db is a collection of files. (corresponds to a table or a data segment in a DBMS)
•Each file is a collection of records. (Represents a single entry or row in a table.)
•Each records is a sequence of fields. (representing a single data item (e.g., name, age, ID).)

•**Database is divided into blocks (also called pages)**
These are the units of data read from or written to disk.
•**Each block contains records**
The number of records per block depends on the block size and record size.
•**Record storage**
Records are stored sequentially or using pointers (depending on the file organization method).

File Organization :

Data Records can be :

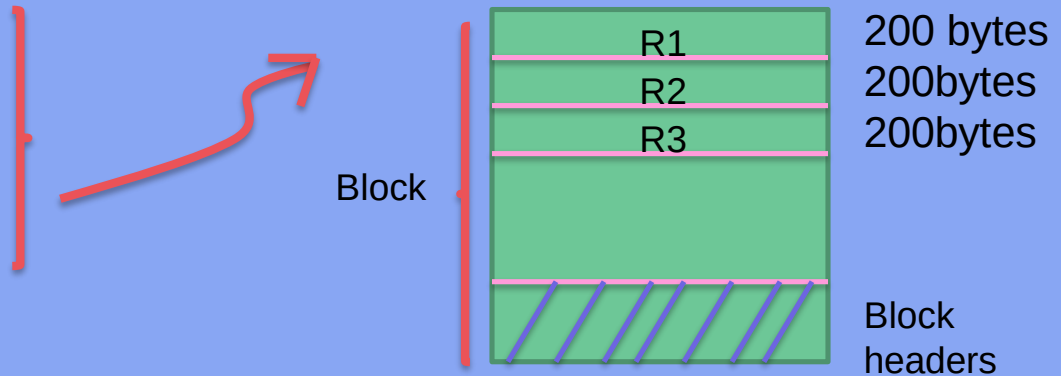
1) Fixed length Records
2) Variable length Records



200 Bytes (for each Records)

1) Fixed length Records

```
Create table R
( A char(100),
  B char(50),
  C char(50));
```



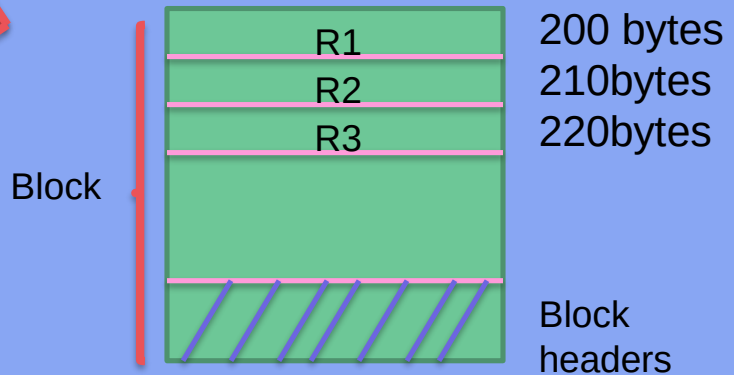
200 bytes
200bytes
200bytes

Block headers

1) Variable length Records

```
Create table S
( D char(100),
  E char(50),
  F text
);
```

Size(D) + Size(E) + Size(F)
100 50 not fixed



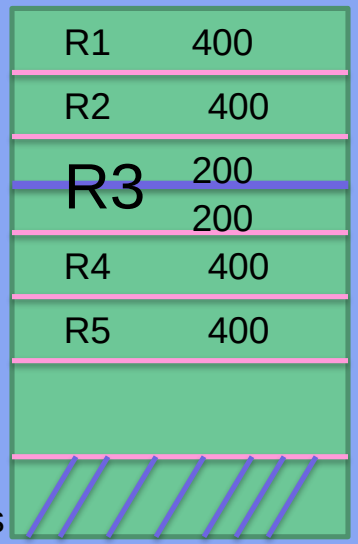
Record Organization :

a) Spanned Organization

Record allow to spin in more than one block.

Example :- Block size 1000B ; Record size 400B

Block factor :- 1000/400 = 2.5 record/block



- Too complex to manage records.
- More access cost to access records.

Advantage :

Possible to allocate file with no internal fragmentation.

No memory wastage.

Suitable:

For variable length records.

a) Unspanned Organization

Complete record must be stored in one block.

Example :- Block size 1000B ; Record size 400B

Block factor :- floor(1000/400) = floor(2.5)=2 record/block

- Memory wastage or Internal fragmentation.
- Less access cost to access records.

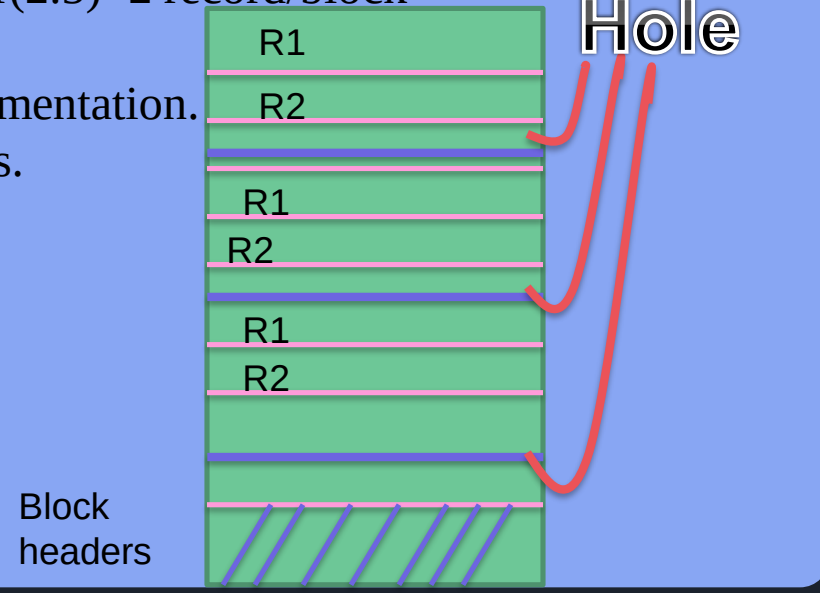
Advantage :

Block access reduces.

No memory wastage.

Suitable:

For fixed length records.



Default organization is Un-spanned.

As, I/O cost is less.

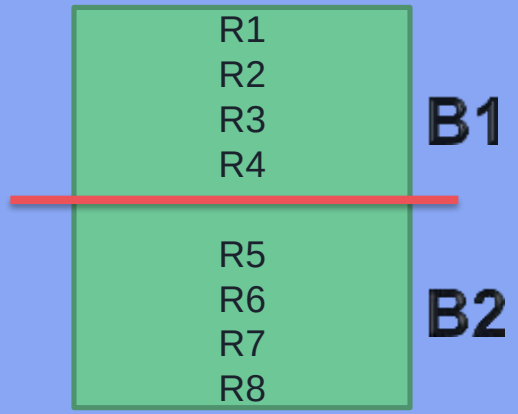
- To organized DB file with fixed length record un-spanned organization is preferred.
- To organized DB file with variable length record spanned organization is preferred.

Blocking Factor[Bf]

Average number of records per block.

Example :- Block factor = 4 (i.e.. 4 records per block)

$$\text{Block Factor} = \frac{\text{Block Size}}{\text{Record Size}}$$



Note :- Block size is always greater than Record size.

Block size : B bytes
Block Header size : H bytes
Record size : R bytes

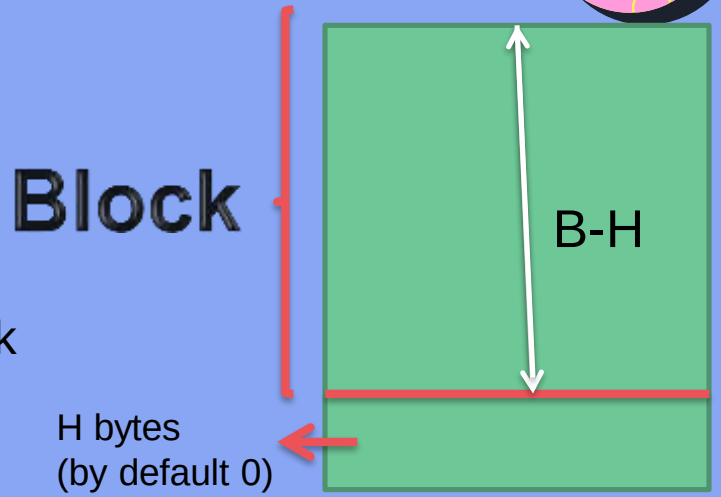
Block factor for un-spanned :

Floor value
e.g. floor(2.5)
= 2

$$= \left\lfloor \frac{B - H}{R} \right\rfloor \text{ Record / Block}$$

Block factor for spanned :

$$= \frac{B - H}{R} \text{ Record / Block}$$

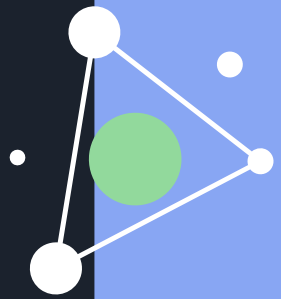


I/O Cost

I/O cost means number of blocks transferred from secondary memory to main memory in order to access some records.

Search key

Attribute used to access data from database.
e.g. Rollno , Sname , etc



Files of Ordered Records

- Also called Sorted or sequential file.
- Record sorted by ordering field.
- Searching easy.
- Insertion Expensive (Re organization of file again).
- Binary Search used.
- To access a record average number of block access = [log2B]

  B: Data block

Files of Un-ordered Records

- Also called Heap (or pile) file
- Record placed in file in order of insertion.
- Insertion easy.
- Searching expensive
- Linear search used.
  - Average = B/2
  - Worst = B
- Deleting technique:

  Rewrite the block

  Use deletion marker

Suppose that:

Record size R =150 bytes ; **Block size B** = 512 bytes ; **r** =30000 records

Then we get,

Blocking factor = floor(512/150) = 3 records/block

Number of file block b = 30000/3 = 10000 Data block

Ordered file:

To access a record average number of block access = [log2B]

log10000 = 14

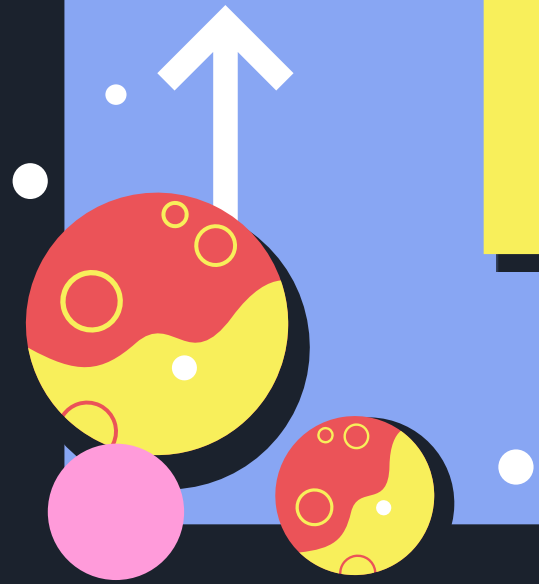
Un-ordered file :

To access a record average number of block access = B/2

10000/2 = 5000 (Average)

10000 (Worst)

Thank-You



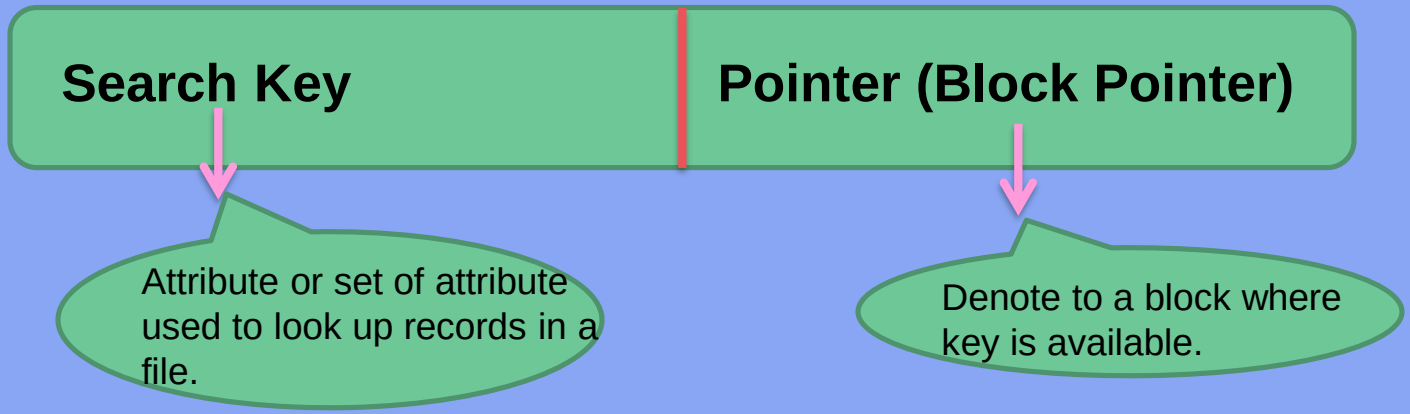
03

Indexing and Index Data Structure

Data structure to speed up search process.

- Indexing are used to improve searching efficiency.
- Reduce I/O cost.
- Provide a faster way to access the data
- Index is a ordered file.

- One record of index file contains 2 fields (called index entries):



- Index files are much smaller than the original file.
- One Index record size = size of search key + size of pointer.
- Index file block size is same as DB file block size.
- Two kinds of indices :

  Ordered indices : Search key are stored in sorted order.

  Hash indices : search key are distributed uniformly across "buckets" using hash function.

- To access a record average number of block access = $Log_2 B_i + 1$

B_i = Index block

Index block access

Data block access

Given the following data file

EMPLOYEE (NAME, SSN, ADDRESS, JOB, SAL, ...)

Suppose that:

- record size R=150 bytes, block size B=512 bytes r=30000 records
- For an index on the SSN field, assume the field size V_{SSN} =9 bytes, assume the record pointer size P_R=7 bytes. Then:

Solution:- One index record size = 9+7 = 16 bytes

Block factor of index file =

= floor(512/16)

= 32 index record per block

• Total number of index entries = 30,000

Total number of index blocks = floor(30000/32) = 938 index block

•To access a record average number of block access = Log_2B_i+1

= Log 938 + 1

= 10 + 1 = 11 Block access

•To access a index average number of block access = Log_2B_i

= Log 938

= 10 block access

Implementation of Indexes:

Dense Index

- Every **search key value** in the data file has an **entry** in the index.
- Example: For a table of 1000 records, you'll have 1000 index entries.

Number of index entries = Number of DB records

Sparse Index

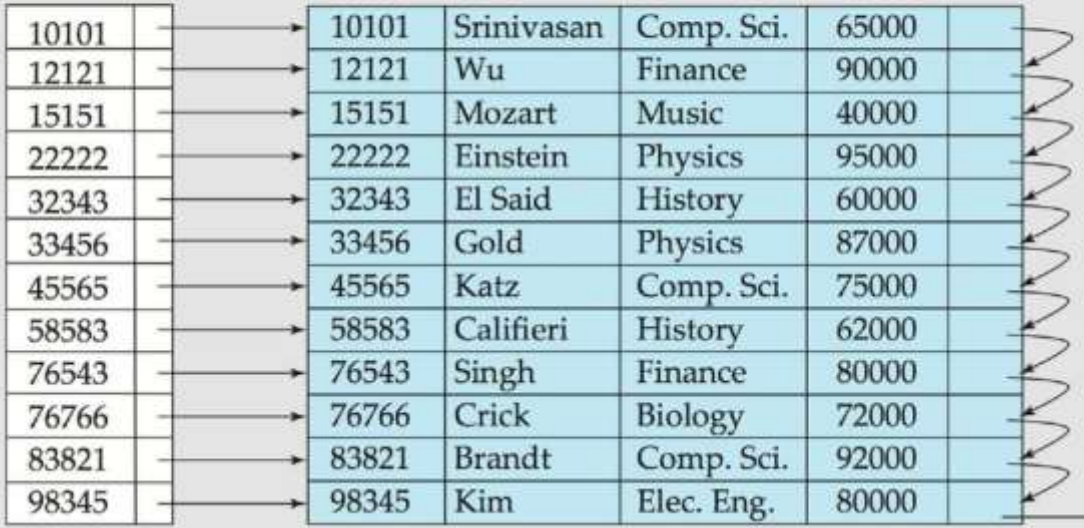
- Index only contains some **search key values**, typically one per **data block**.
- Saves space, but may require scanning inside the block.

Number of index entries = Number of Blocks

Anchor Record :

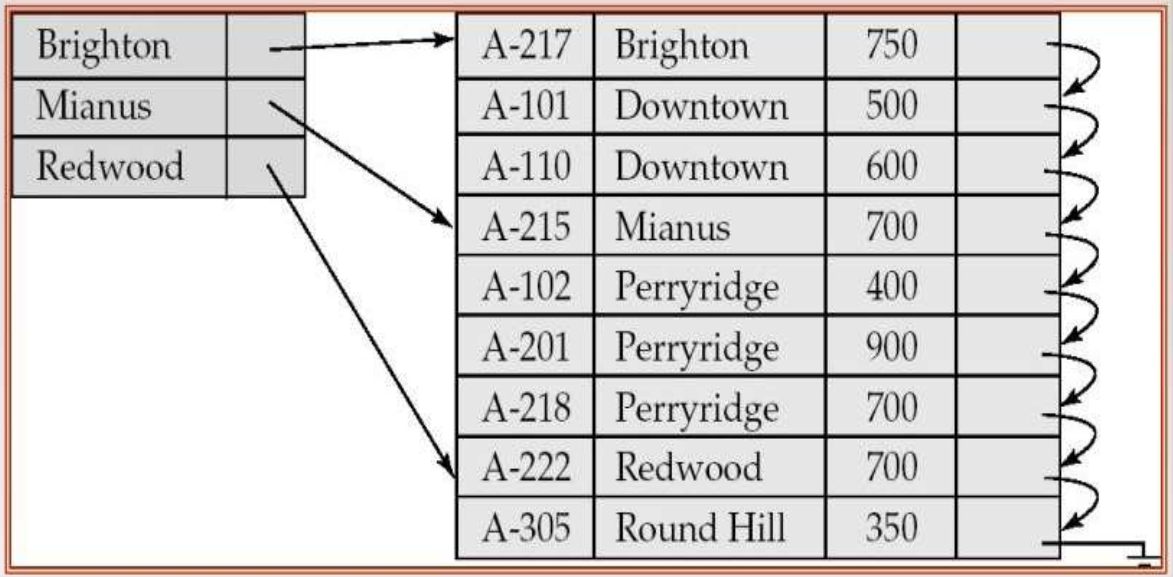
- A **representative record** in a **block** or a group of records.
- Used in **sparse indexing (**also or **spanned file organizations)** to point to the **first (or key) record** in each data block.
- Acts as an **“anchor point”** for searching.

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation

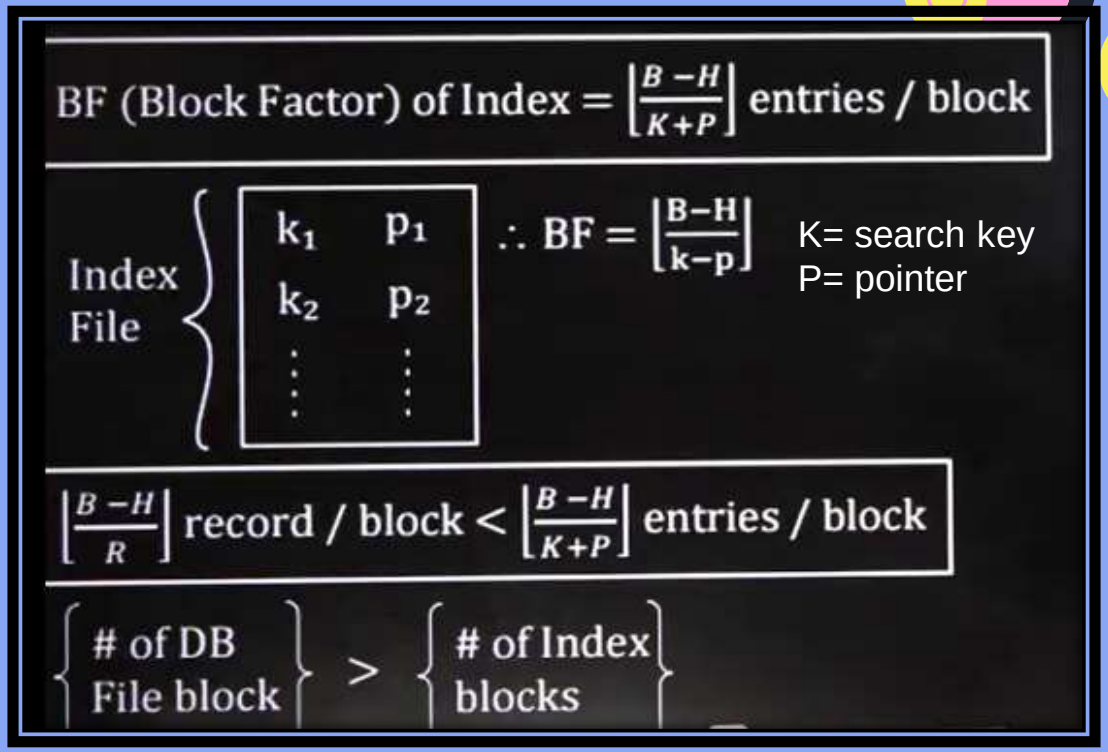


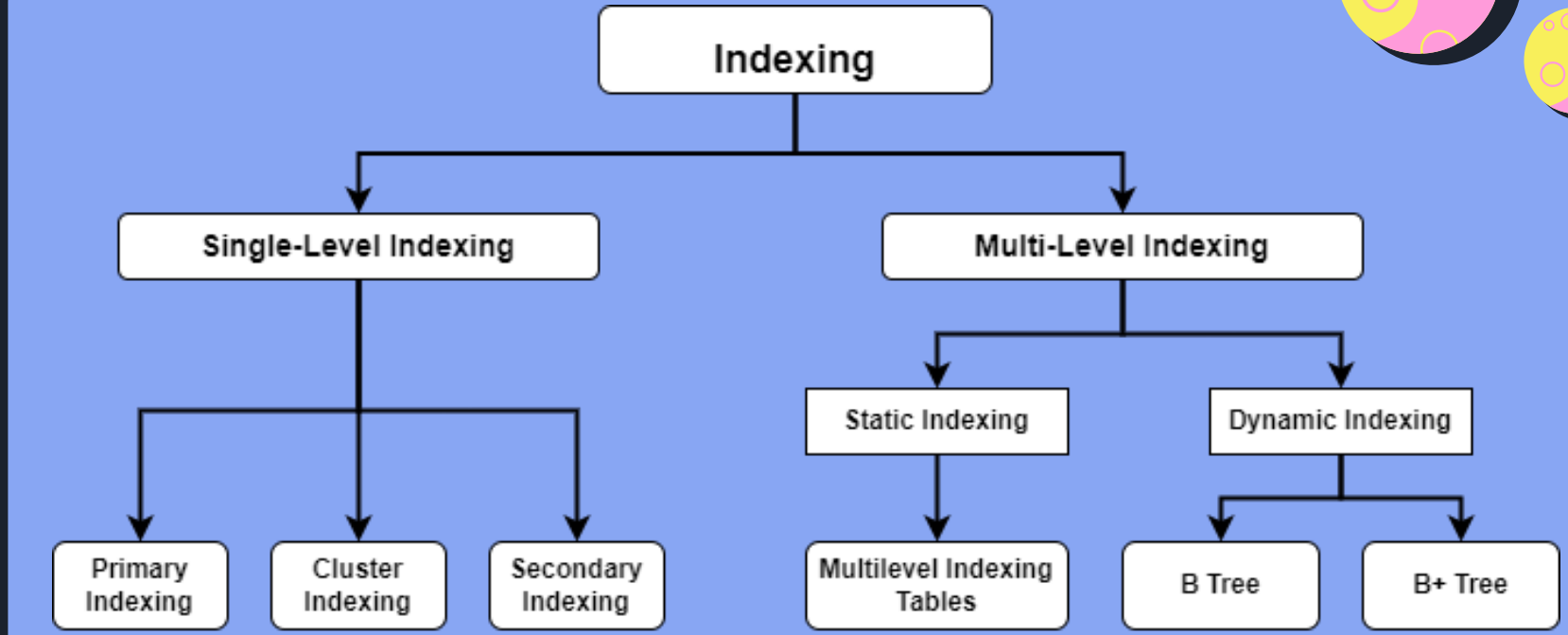
10101	10101	Srinivasan	Comp. Sci.	65000
12121	12121	Wu	Finance	90000
15151	15151	Mozart	Music	40000
22222	22222	Einstein	Physics	95000
32343	32343	El Said	History	60000
33456	33456	Gold	Physics	87000
45565	45565	Katz	Comp. Sci.	75000
58583	58583	Califieri	History	62000
76543	76543	Singh	Finance	80000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

Example of Sparse Index Files

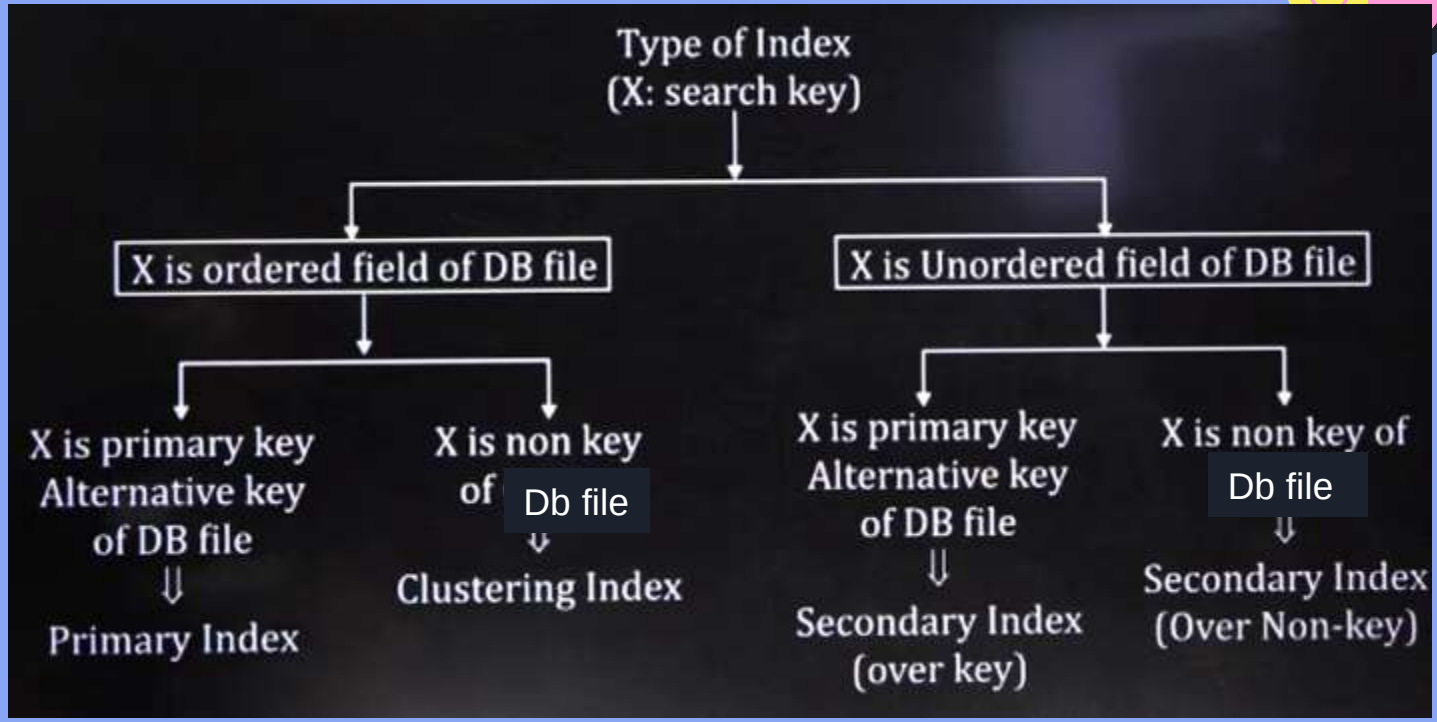


Index file :


BF (Block Factor) of Index = $\left\lfloor \frac{B-H}{K+P} \right\rfloor$ entries / block
Index File
k_1 p_1
k_2 p_2
$\therefore BF = \left\lfloor \frac{B-H}{k-p} \right\rfloor$
K= search key
P= pointer
$\left\lfloor \frac{B-H}{R} \right\rfloor$ record / block < $\left\lfloor \frac{B-H}{K+P} \right\rfloor$ entries / block
# of DB File block
>
# of Index blocks


Indexing
Single-Level Indexing
Multi-Level Indexing
Primary Indexing
Cluster Indexing
Secondary Indexing
Static Indexing
Dynamic Indexing
Multilevel Indexing Tables
B Tree
B+ Tree

Index file :


Type of Index
(X: search key)
X is ordered field of DB file
X is Unordered field of DB file
X is primary key
Alternative key
of DB file
⇓
Primary Index
X is non key
of
Db file
⇓
Clustering Index
X is primary key
Alternative key
of DB file
⇓
Secondary Index
(over key)
X is non key of
Db file
⇓
Secondary Index
(Over Non-key)

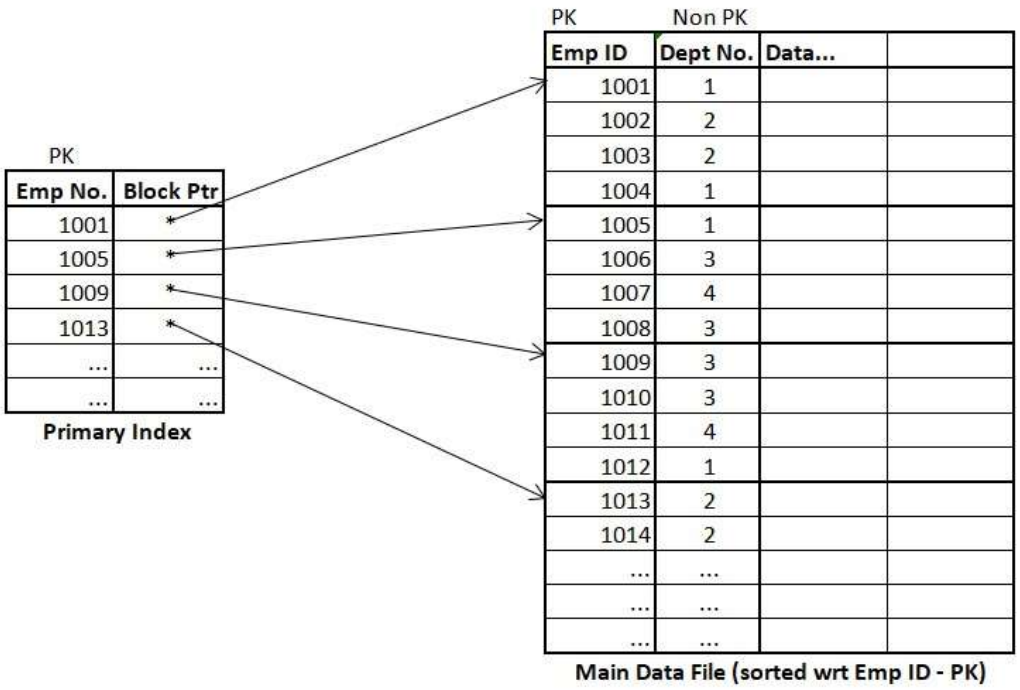
Note

- At most one Primary Index is possible per Relation(Table)
- More than one Secondary index is possible.
- In a Relation either Cluster index or Primary index is possible not both.

Primary Indexing

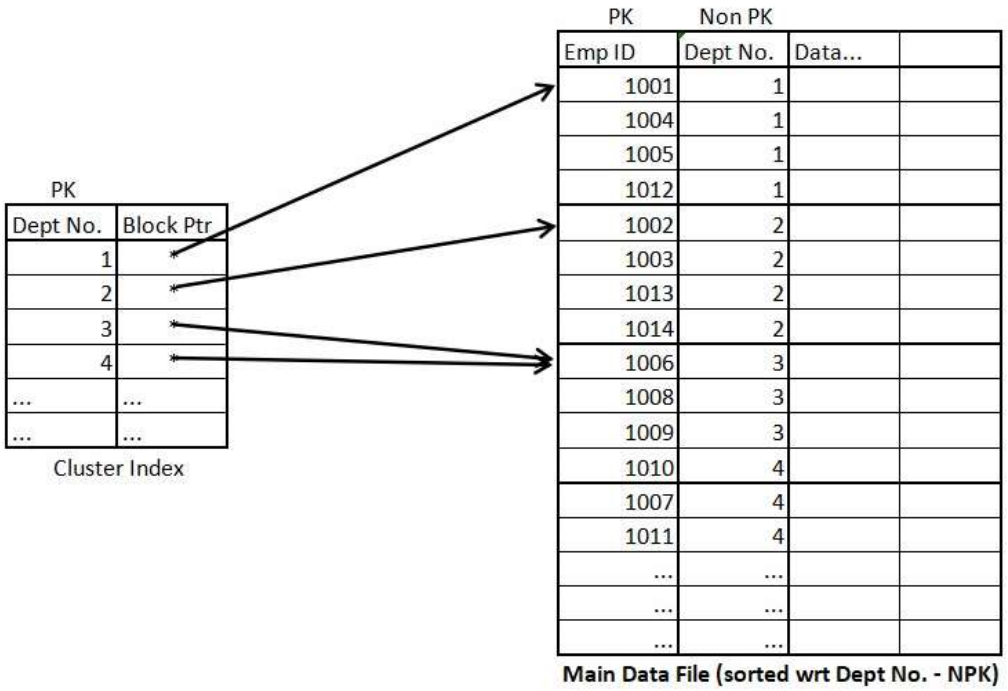
- **Based on the primary key** of a table.
- One **index entry per data block**, pointing to the **first record** (anchor) in each block.
- **Sorted** on the primary key.
- Index is **dense** or **sparse**.
- Also called **clustering index** *if the data is stored in the same order as the index key.*

Example: Index on StudentID (Primary Key).


PK
Emp No.
Block Ptr
1001
1005
1009
1013
Primary Index
PK
Non PK
Emp ID
Dept No.
Data...
1001 1
1002 2
1003 2
1004 1
1005 1
1006 3
1007 4
1008 3
1009 3
1010 3
1011 4
1012 1
1013 2
1014 2
Main Data File (sorted wrt Emp ID - PK)

Clustering Indexing

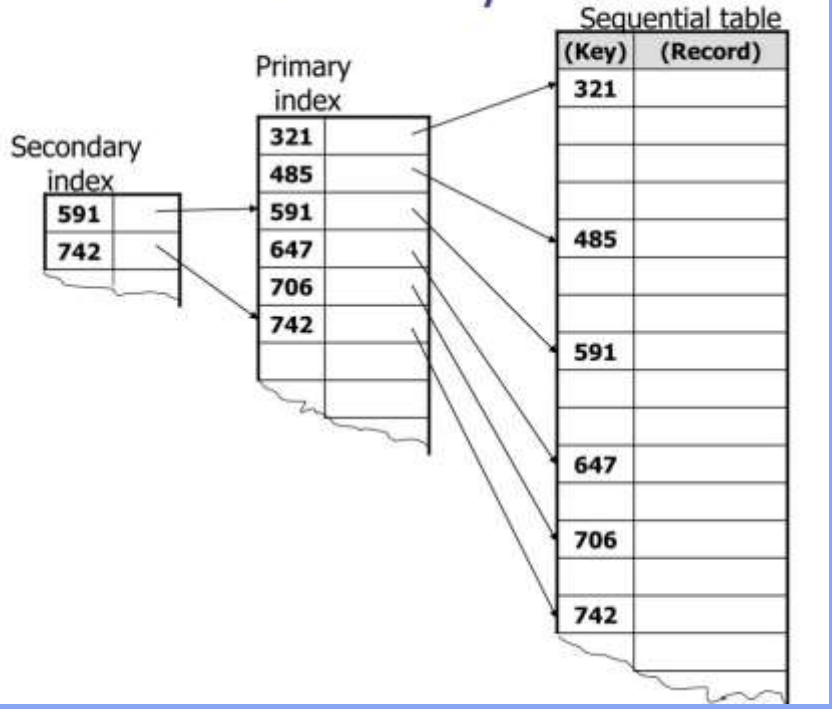
- Data **not uniquely identified** by the index field.
- Index built on **non-primary key column**.
- Records **physically stored** in the same order as the **clustering field**.
- One index entry per **cluster** or **group of similar records**.
- *Example:* Index on Department where multiple employees are from the same department.


PK
Dept No.
Block Ptr
1
2
3
4
...
...
Cluster Index
PK
Non PK
Emp ID
Dept No.
Data...
1001
1004
1005
1012
1002
1003
1013
1014
1006
1008
1009
1010
1007
1011
Main Data File (sorted wrt Dept No. - NPK)

Secondary Indexing

- Index on **non-primary key attributes**.
- Data is **not sorted** by this attribute.
- Index can be **dense** and may have **multiple entries for same value**.
- Useful for **frequently searched columns** that are not primary keys.
- *Example:* Index on City in a Customer table.

A secondary index

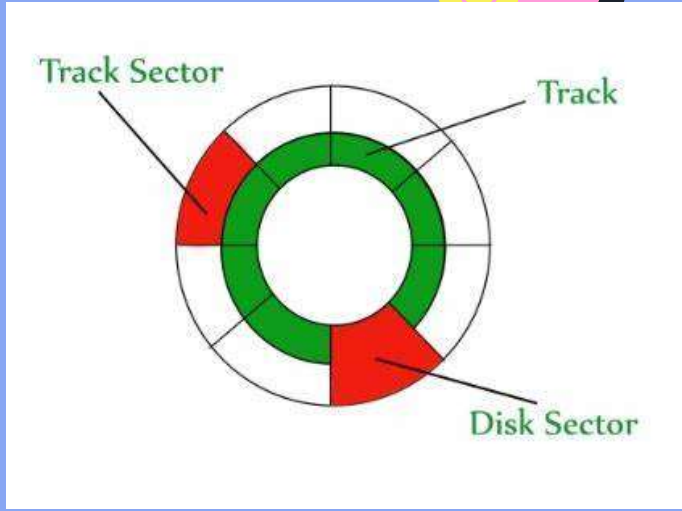


RAM :-

- Temporary(volatile) memory
- Expensive
- Limited.

Hard Disk :-

- Persistence
- Cheap
- Take more time(I/O operation)



- Intersection of Track and Sector give us a file block (or Disk Sector). (typically 4KB)
- Whenever we need to process any data we need to take the entire file block from Hard disk to RAM. (Because C.P.U can access data stored in R.A.M)

Time Taken to find 1 million Records. (Rough value)

(All values are on assumptions and for understanding the concept.)

Let say each record is an entry which takes 400 bytes.

1 Block => 4KB data (can be stored)

- 4000 / 400
- i.e. 1 block can store, 10 records / entry.

Now 1 million records,

10^6 Records =>

No of blocks = $10^6/10 = 10^5$ blocks

$= 10^5$ millisecond [Let say an I/O operation in HDD takes 1 millisecond]

= 100 second (Huge) ☹

- Each **record** = **400 bytes**
- **1 file block** (or disk block) = **4 KB** = **4000 bytes** (approx.)
- So, **1 block can hold:**

$$\frac{4000}{400} = 10 \text{ records per block}$$

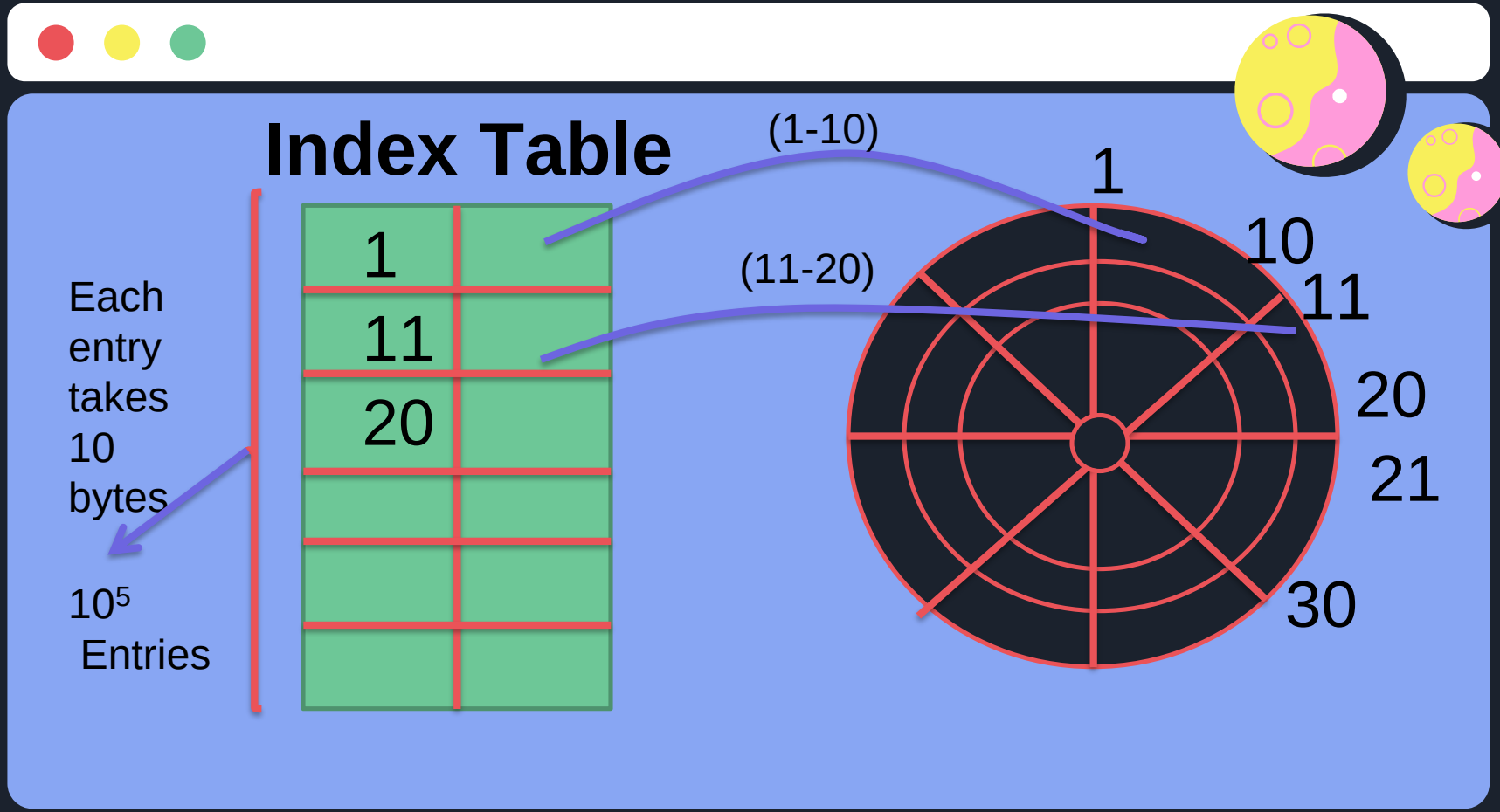
📊 For 1 Million Records (= 10^6 records):

- Total number of blocks needed:

$$\frac{10^6}{10} = 10^5 = 100{,}000 \text{ blocks}$$

- If **each I/O operation** (reading one block from HDD) takes **1 millisecond**:

$$100{,}000 \text{ blocks} \times 1 \text{ ms/block} = 100{,}000 \text{ ms} = 100 \text{ seconds}$$


Index Table
Each entry takes 10 bytes
10^5 Entries
1
11
20
(1-10)
(11-20)
1
10
11
20
21
30

1 block can store = 4000 bytes
Each entry takes 10 bytes
Each block can store 400 entries of table .

How many block are needed to store this table

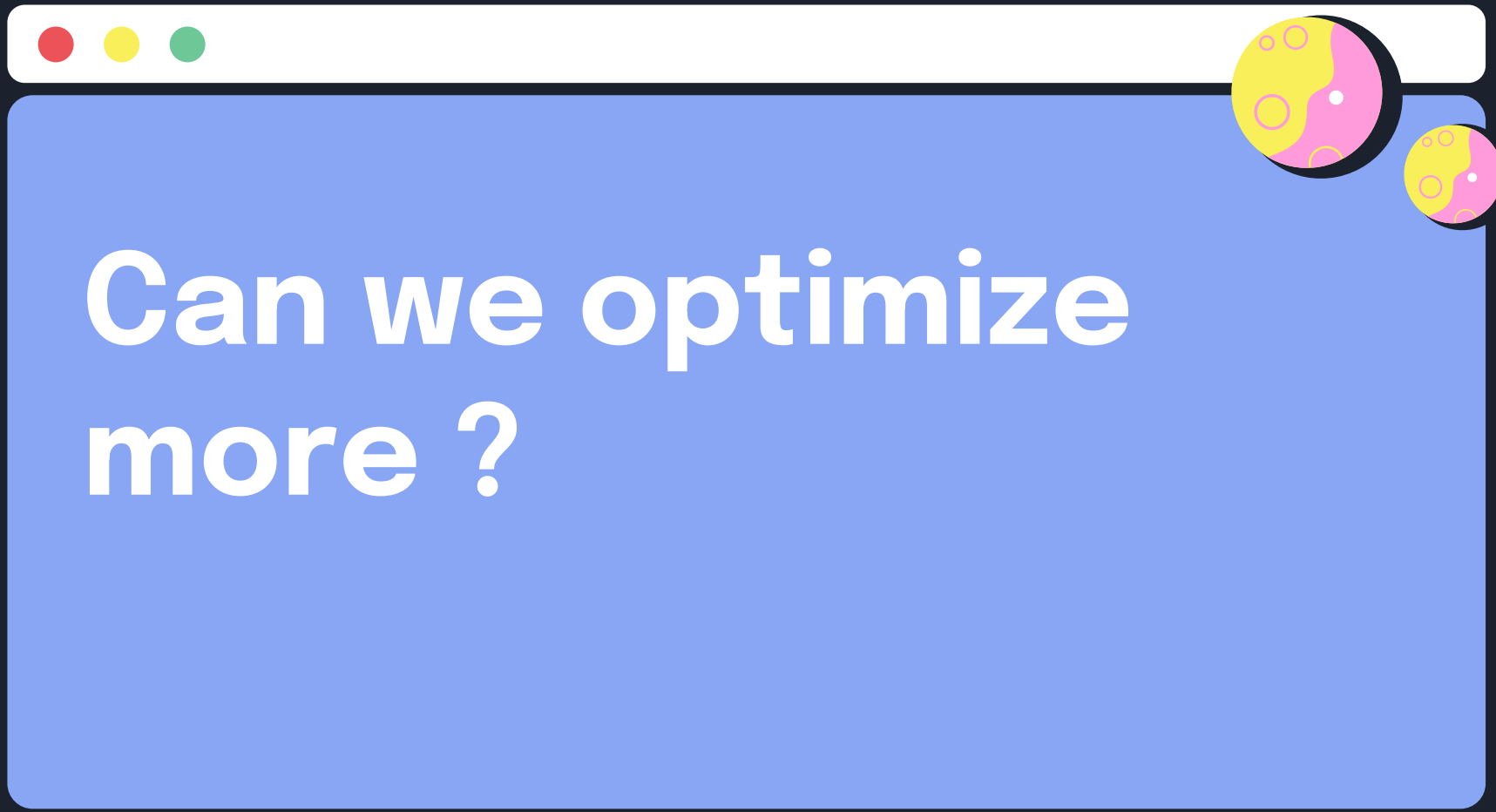
No. of block => 10^5 / 400

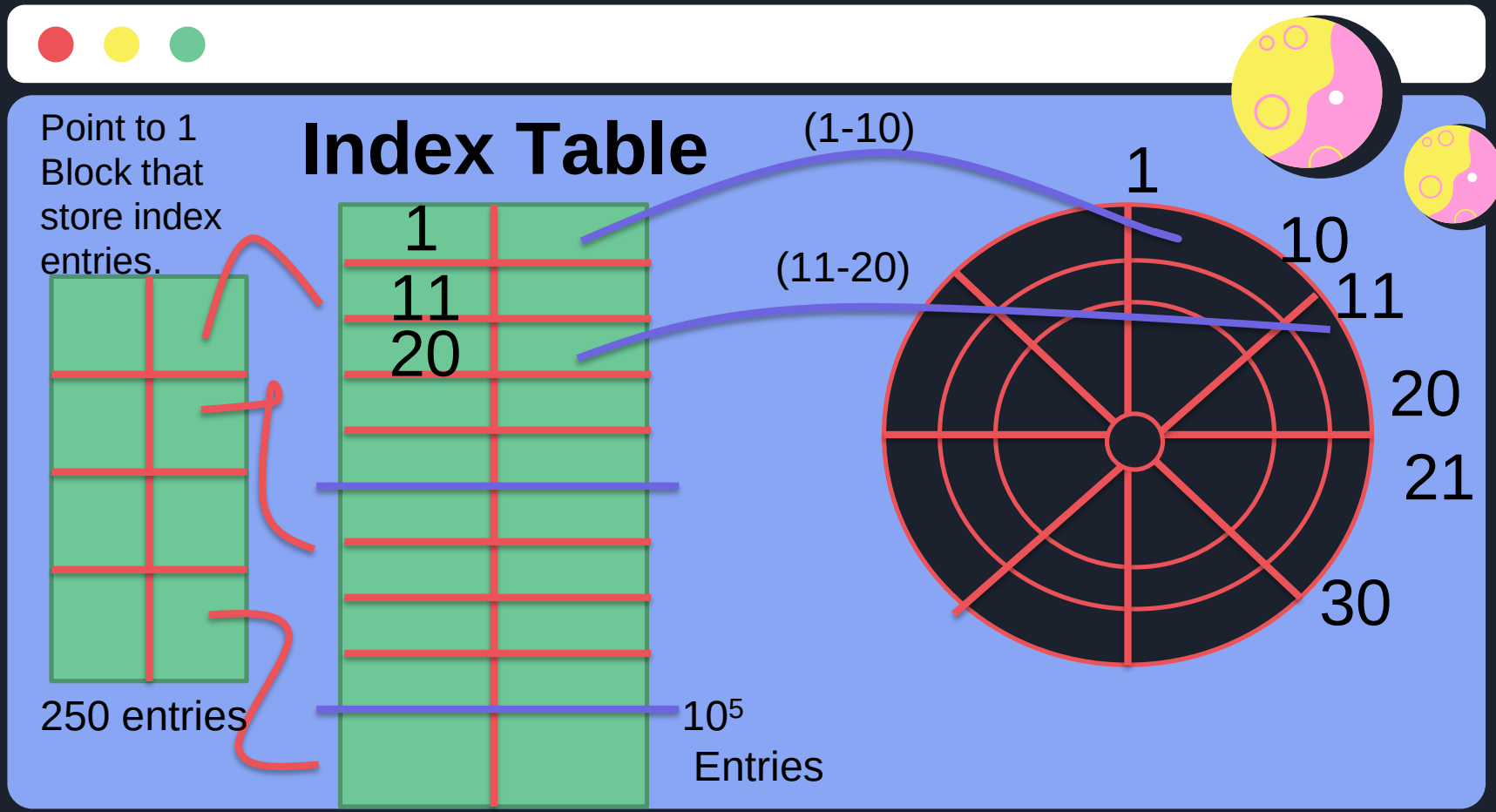
=> 250 blocks of disk

And, another 1ms to read the corresponding data block / file block .

=> 250 ms (0.25 seconds)

Optimized


Can we optimize more ?


Point to 1 Block that store index entries.
Index Table
1
11
20
(1-10)
(11-20)
1
10
11
20
21
30
250 entries
10^5 Entries

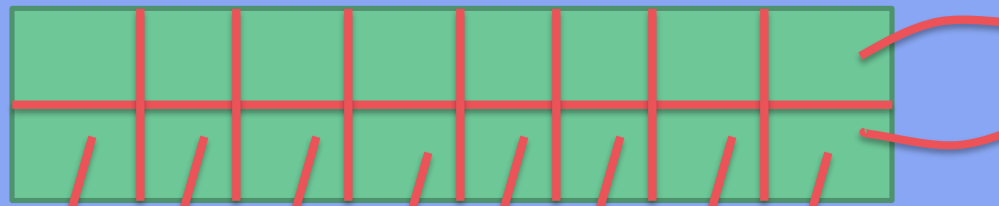
250 entries * 10 bytes = 2500 bytes [<4kb]
3 I/O operation taking 3 ms [0.003 seconds]

Finally done !

Multi-level Indexing

Multi-level Indexing

This is how
B-tree looks.



Keys

Child pointer

Nodes

Pointer

Search , Insert , Delete

- Every I/O operation is expensive.

Most common DB operations

1 million records = $\log(10^6) = 20$
20 I/O operations with Hard disk for search , insert and delete.

Database	Operation		
	Search	Insert	Delete
Balance BST	20	20	20

Database	Operation		
Un-Balance BST	Search	Insert	Delete
	10^6	10^6	10^6

Database	Operation		
	Search	Insert	Delete
Sorted Array	20	10^6	10^6

Tricky :- Shifting of elements.

Database	Operation		
	Search	Insert	Delete
B-tree (n-way Tree)	3	3	3

Thank-You

Thank-You