LET'S LEARN

DATABASE MANAGEMENT SYSTEM (DBMS)

-By Riti Kumari

LET'S START WITH DBMS :)

Query Optimization

Whenever we want to improve the efficiency of a query, make it execute faster and consume fewer resources we use query optimization techniques.

Optimizing SQL queries is essential for improving the performance of database applications

1.Use Indexes Efficiently: Index the columns frequently used in WHERE, JOIN, ORDER BY, and GROUP BY clauses to speed up data retrieval.

2.Select Only Necessary Columns : Avoid SELECT *, specify only the columns you need in your query. This reduces the amount of data transferred and processed.

LET'S START WITH DBMS :)

Query Optimization

3. Optimize JOIN Operations : Choose the Right JOIN Type and use indexed columns in Join conditions.

4. Partition Large Tables: For very large tables, consider partitioning them to improve query performance. Partitioning allows the DBMS to scan only relevant partitions, reducing I/O and improving response times.

5. Cache results : Cache the results of frequently executed queries to avoid redundant calculations.

LET’S START WITH DBMS :)

Physical Storage And File Organization

Physical Storage

Storage device : For storing the data, there are different types of storage options available.

- Primary Storage -> Main memory and cache (fastest and expensive and as soon as the system leads to a power cut or a crash, the data also get lost.)

- Secondary Storage -> Flash Memory and Magnetic Disk Storage (non-volatile)save and store data permanently.

- Tertiary Storage-> Optical disk and Magnetic tape (used for data backup and storing large data offline)

LET'S START WITH DBMS :)

Physical Storage And File Organization

Databases store data in files on disk. Each table or index may correspond to one or more files. Each file has sequence of records.
Accessing data from RAM is much faster than accessing data from a hard disk or SSD. This is because RAM is designed for high-speed read and write operations. Data retrieval in RAM typically involves fetching data in nanoseconds.

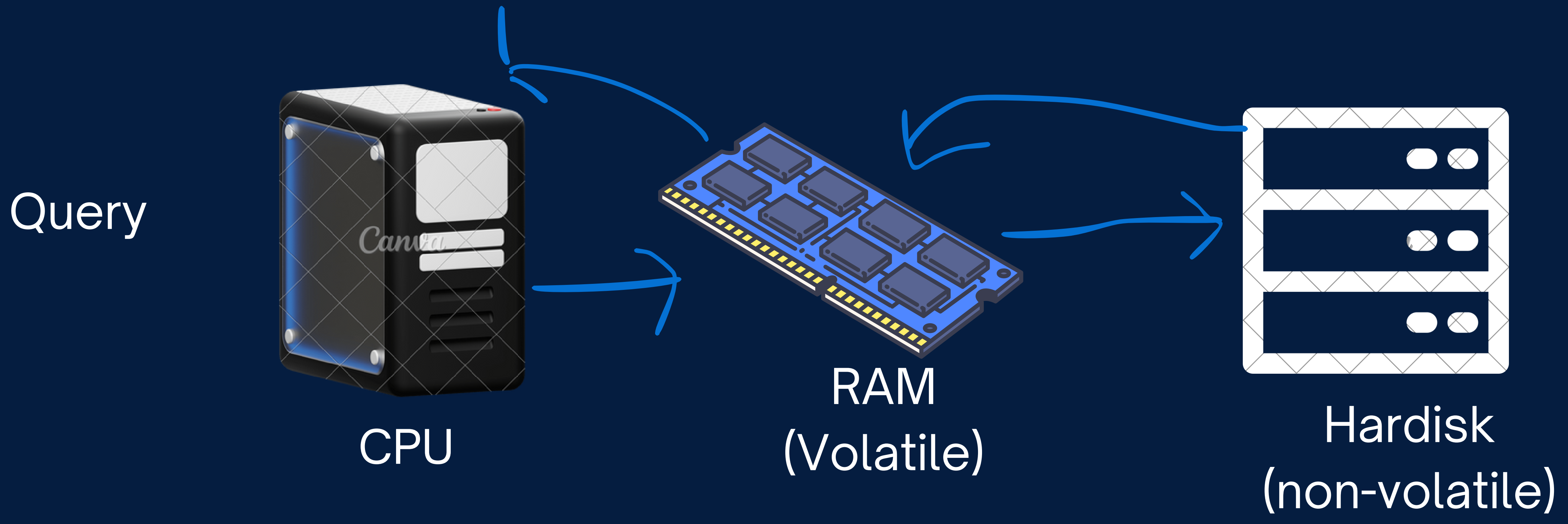
- **RAM** : Provides fast, volatile memory for quick data retrieval. Data is accessed via addressable locations and cache memory enhances speed.

- **Hard Disk:** Provides slower, non-volatile storage with mechanical components that impact access speed. Data retrieval involves moving the read/write head and waiting for disk rotation.

LET'S START WITH DBMS :)

Physical Storage And File Organization

When a program or application needs to access data, it first checks if the data is available in RAM. If the data is not in RAM, it must be loaded from a slower storage device like a hard disk or SSD. Index files in a Database Management System (DBMS) are stored on disk but are also managed in RAM to optimize performance



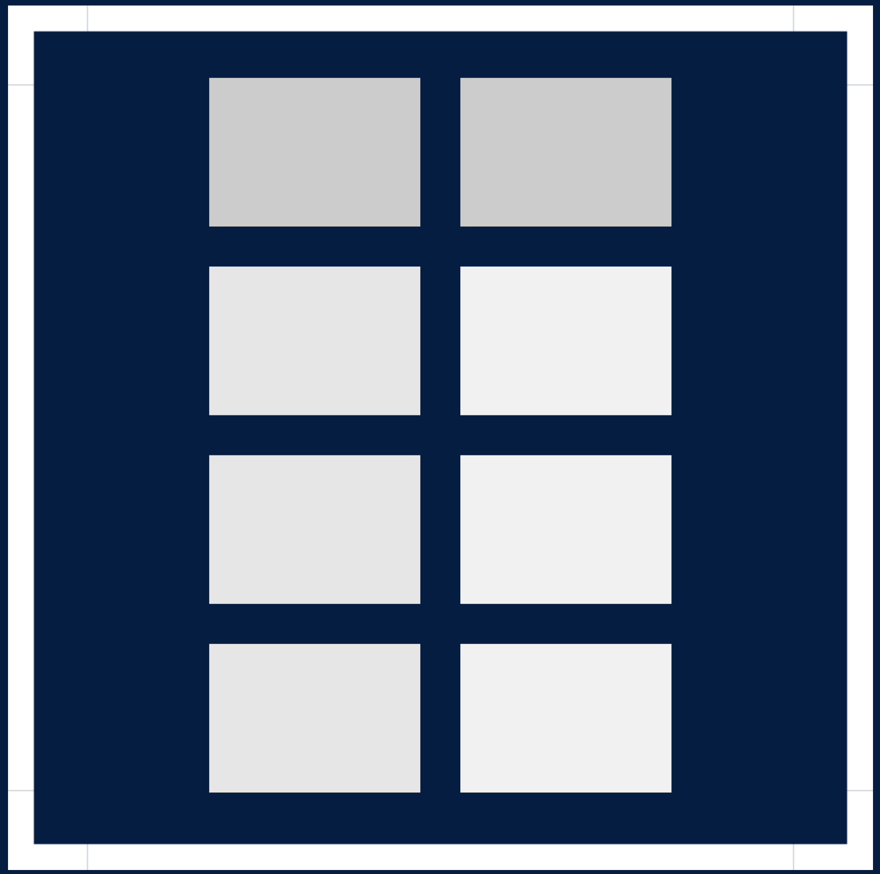
LET’S START WITH DBMS :)

Physical Storage And File Organization

How file is organized?

Rollno	Sname	SAge

DB



HARDISK

Files are allocated on the hard disk in contiguous blocks to reduce fragmentation.

No of records stored in each block= Size of block/size of record

Records can be stored in 2 ways-> sorted(seraching is fast) or unsorted(searching is slow, insertion is fast)

LET’S START WITH DBMS :)

Indexing and its types

Why do we need indexing?

Imagine you have a 1,000-page textbook and you want to serach a topic “xyz”.

- When indexing is not present-> You need to manually search for each term in the book, which could be frustrating and time-consuming.

- When indexing is present->The index allows you to quickly locate each topic. You can look up "xyz," find that it's covered on pages 448-490.This makes it more efficient and less time-consuming.

Indexing is a critical technique in database management that significantly improves the performance of query operations by minimizing the no of disk access. Index table is always sorted

LET'S START WITH DBMS :)

Indexing and its types

Why do we need indexing?

- In the same way how we have used indexing in book , it helps us to find specific records or data entries in a large table or dataset.

Search key	Data acccess
It is used to find the record	It contains the address where the data item is stored in memory for the provided search key

It helps in finding what a user is looking for

Index table

set of pointer holding address of a block

LET'S START WITH DBMS :)

Indexing and its types

How it helps?

- **Improved Query Performance:** Indexes allow the database management system (DBMS) to locate and retrieve data much more quickly than it could by scanning the entire table.

- **Indexes can be used to optimize queries that sort data (ORDER BY) or group data (GROUP BY) :** When an index is available on the columns being sorted or grouped, the DBMS can retrieve the data in the desired order directly from the index, eliminating the need for additional sorting operations.

- We have certain indexing methods like Dense(index entry for all the records) and Sparse(index entries for only a subset of records) index.

LET'S START WITH DBMS :)

Indexing and its types

Sparse Index: A sparse index is a type of database indexing technique where the index does not include an entry for every single record in the database. Instead, it contains entries for only some of the records, typically one entry per block (or page) of data in the storage. This makes sparse indexes smaller and faster to search through compared to dense indexes, which have an entry for every record.

It is used when we have an ordered data set.

EX: If a table has 1,000 rows divided into 100 blocks, and you create a sparse index, the index might only have 100 entries, with each entry pointing to the first record in each block.

Index table

Search key	data ref
1	B1
4	B2

Sparse Index
no of records in index
file=no of blocks

1 Ram 2 Shyam 3	**B1**
4 5 6	B2

disk

RollNo	Name
1	Ram
2	Shyam
3	Raghu
4	Riti
5	Raj
6	Rahul

How Sparse Indexing Works:

- **Primary Index:** Sparse indexes are often used with primary indexes, where the table is sorted on the indexed column. The sparse index only includes an entry for the first record in each block or page.

- **Searching:** When searching for a specific value, the database uses the sparse index to quickly locate the block where the record might reside. It then searches within the block to find the exact record.

Disadvantages of Sparse Indexes

- **Additional I/O Operations:** After using the index to find the correct block, an additional search within the block is required to find the specific record.

- **Less Efficient for Random Access:** If queries often need to retrieve random, non-sequential records, a dense index might be more efficient.

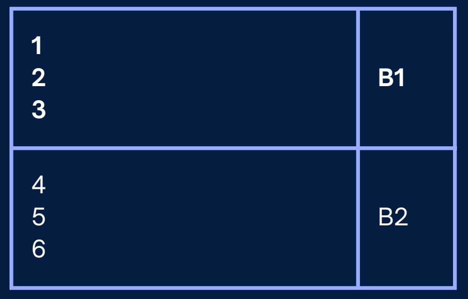
Use Cases:

- **Primary Indexing on Large Tables:** Sparse indexes are ideal for primary indexes on large tables where records are sequentially stored.
- **Range Queries:** Sparse indexes can be effective for range queries where the query needs to retrieve a range of records rather than a single record.

Index table

Search key	data ref
1	B1
4	B2

Sparse Index
no of records in index
file=no of blocks



disk

RollNo	Name
1	Ram
2	Shyam
3	Raghu
4	Riti
5	Raj
6	Rahul

LET’S START WITH DBMS :)

Indexing and its types

Dense Index: A dense index is a type of indexing technique used in databases where there is an index entry for every single record in the data file. This means that the index contains all the search keys and corresponding pointers (or addresses) to the actual data records, making it highly efficient for direct lookups.

It is used when we have an un-ordered data set.

EX : Consider a table with 1,000 rows, and you create a dense index on a non-primary key column. The dense index will have no ofunique non key records, each pointing to a specific row in the table.

These are useful in scenarios where random access to individual records is quiet frequent .

Search key	data ref
18	B1
19	B1
20	B2
21	B2

Index table

18 Ram 18. Shyam 19	B1
20 20 21	B2

Dense Index
no of records in index file=no of blocks unique non key records

Age	Name
18	Ram
18	Shyam
19	Raghu
20	Riti
20	Raj
21	Rahul

How Dense Indexing Works:

In a dense index, each index entry includes:

- **A search key** (the value of the indexed column).
- **A pointer** (or address) to the actual record in the data file.

When a query searches for a specific value, the database uses the dense index to quickly locate the corresponding record.

Disadvantages of Dense Indexes

- **Storage Intensive:** Requires significant storage space because it maintains an entry for every record in the table.
- **Maintenance Overhead:** Inserting, updating, or deleting records requires updating the dense index, which can be costly in terms of performance, especially for large tables.

Search key	data ref
18	B1
19	B1
20	B2
21	B2

Index table

18 Ram 18. Shyam 19	B1
20 20 21	B2

Dense Index
no of records in index file=no of blocks unique non key records

Age	Name
18	Ram
18	Shyam
19	Raghu
20	Riti
20	Raj
21	Rahul

Use Cases:

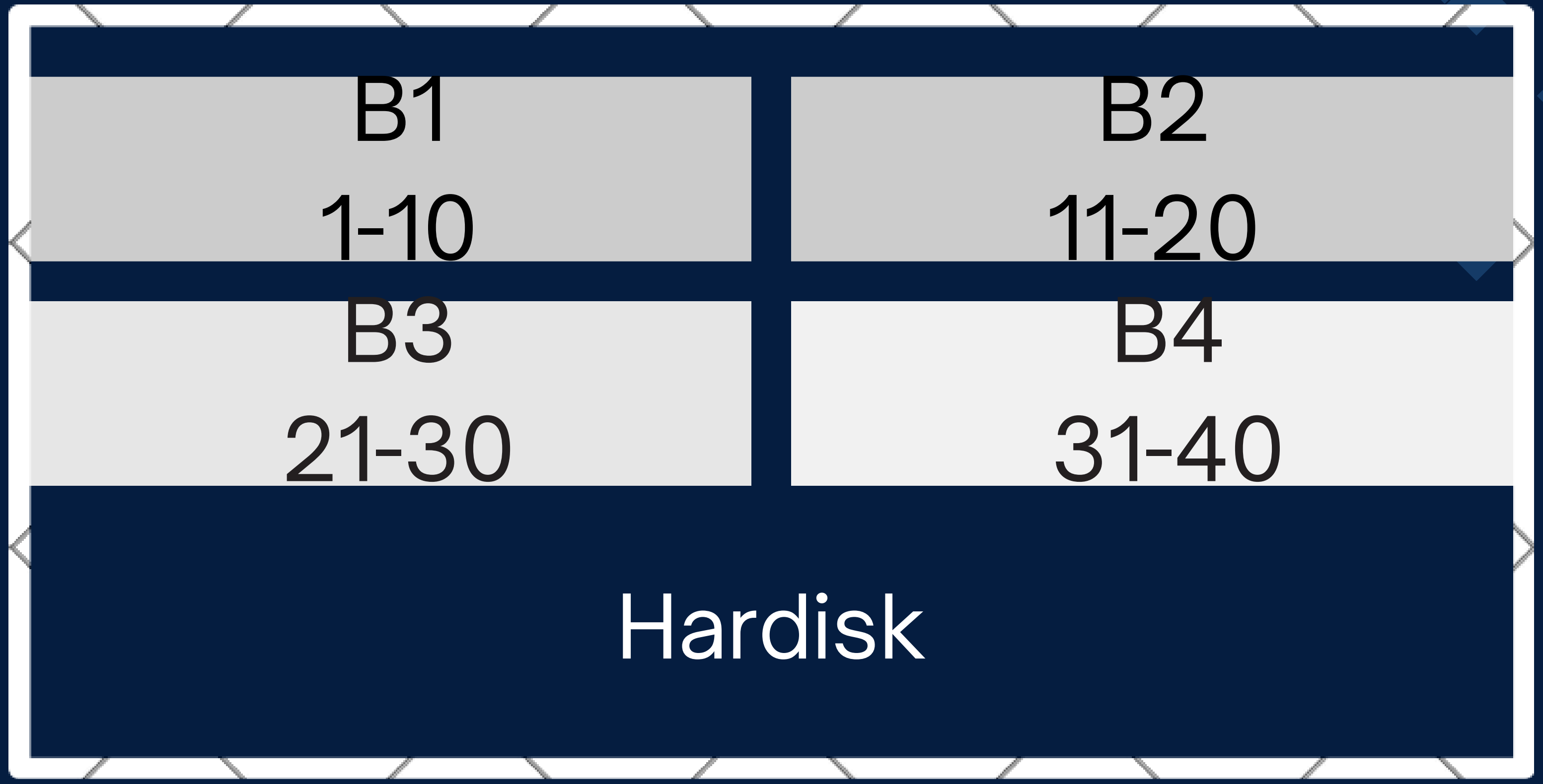
- **Primary Indexing:** Dense indexes are often used for primary indexing when the table is small to medium-sized, and quick access to individual records is required.
- **Exact Match Queries:** Ideal for situations where queries frequently request individual records based on an exact key match.

LET’S START WITH DBMS :)

Indexing and its types

Index table

Search key	data ref
1	B1
11	B2
21	B3



When a query is run, the database engine checks the index to find the pointers to the rows that contain the desired data. It then retrieves these rows directly, without scanning the entire table.

LET'S START WITH DBMS :)

Indexing and its types

How indexing works?

B1
1-10

B2
11-20

It takes a search key as input and then returns a collection of all the matching records. Now in index there are 2 columns, the first one stores a duplicate of the key attribute/ non-key attribute(serach key) from table while the second one stores pointer which hold the disk block address of the corresponding key-value.

LET'S START WITH DBMS :)

Indexing and its types

Indexing

Clustered Index

Primary index

When key= PK
order= sorted
Follows sparse indexing

Clustering Index

When key= non-key
order= sorted
Follows dense indexing

Multilevel Indexing

When there is one more level of indexing in the index table we use inner and outer index.
Mostly used for large files

Non clustered Index

Secondary index

When key= non-key/key
order= unsorted
Follows dense indexing

LET'S START WITH DBMS :)

Indexing and its types

Key data: Unique identifiers for each record. Often used to distinguish one record from another.
Non-key data: All other data in the record besides the key.

- **Single-Level Indexing** is straightforward and works well for smaller databases, offering a direct approach to speeding up query performance.
- **Multi-Level Indexing** is more complex but necessary for larger databases, as it effectively manages large indexes by creating additional layers of indexing, thereby improving data retrieval times.

LET’S START WITH DBMS :)

Primary Index

It enhances the efficiency of retrieving records by using their primary key values. It establishes an index structure that associates primary key values with disk block addresses. This index is composed of a sorted list of primary key values paired with their corresponding disk block pointers, thereby accelerating data retrieval by reducing search time. It is especially beneficial for queries based on primary keys. It is done on sorted data

Search key	data ref
1	B1
4	B2

It creates an index structure that maps primary key values to disk block addresses.
Index table

1 **2** **3**	**B1**
4 5 6	B2

Sparse Index
no of reocrds in index file=no of blocks

RollNo	Name
1	Ram
2	Shyam
3	Raghu
4	Riti
5	Raj
6	Rahul

LET’S START WITH DBMS :)

Primary Index

Example : In a Users table with a UserID column as the primary key, a primary index on UserID allows for quick retrieval of user records when you know the UserID

The primary index ensures that the database can immediately locate the row associated with a given UserID, making operations like SELECT, UPDATE, and DELETE highly efficient.

LET'S START WITH DBMS :)

Cluster Index

The cluster index is generally on a non- key attribute. The data in the table is typically ordered according to the order defined by the clustered index. Mostly each index entry points directly to a row. It is mostly used where we are using GROUP BY. It physically order records in a table based on the indexed columns.

Search key	data ref
18	B1
19	B1
20	B2
21	B2

Index table

18 **18** **19**	**B1**
20 20 21	B2

Dense Index
no of reocrds in index file=no of blocks unique non key records

Age	Name
18	Ram
18	Shyam
19	Raghu
20	Riti
20	Raj
21	Rahul

LET’S START WITH DBMS :)

Cluster Index

Example : In an Employees table, if you often need to produce lists of employees ordered by LastName, creating a clustered index on LastName can make these queries faster.

The clustered index keeps the rows in LastName order, reducing the need for the database to perform additional sorting when executing queries.

LET'S START WITH DBMS :)

Multilevel Index

A multilevel index is an advanced indexing technique used in database management systems to manage large indexes more efficiently. When a single-level index becomes too large to fit into memory, multilevel indexing helps by breaking down the index into multiple levels, reducing the number of disk I/O operations needed to search through the index

- First Level (Primary Index): The first level is the original index, where each entry points to a block of data or a page in the table.
- Second Level (Secondary Index): If the first level index is too large to fit in memory, a second-level index is created. This index points to blocks of the first-level index, effectively indexing the index itself.

LET’S START WITH DBMS :)

Multilevel Index

Eid	Pointer
E101	
E201	
E301	

Outer index

Eid	Pointer
E101	B1
E151	B2

Eid	Pointer
E201	B3
E251	B4

Eid	Pointer
E301	B5
E351	B6

Inner index

Eid	Data
E101 E102 . .	
E151 E152 . .	
E201 E202 . .	

LET'S START WITH DBMS :)

Secondary Index

Secondary indexing speeds up searches for non-key columns in a database. Unlike primary indexing, which uses the primary key, secondary indexing focuses on other columns. It creates an index that maps column values to the locations where the records are stored, making it faster to find specific records without scanning the entire table.

The data is mostly unsorted and it can be performed on both key and non-key attributes.

LET'S START WITH DBMS :)

Secondary Index

1. when serach key is key attribute and data is unsorted

Dense Index
no of reocrds in index file=no of blocks unique non key records

Search key	data ref
18	B1
20	B1
22	B2
24	B2
. . .	. . . .

Index table

Age **18** 20 45 67 68	**B1**
22 24 30	B2

LET'S START WITH DBMS :)

Secondary Index

2. when serach key is non- key attribute and data is unsorted

Dense Index
no of reocrds in index file=no of blocks unique non key records

Search key	data ref
18	B1
20	B1
22	B1->B2
24	B2
. . .	. . . .

Index table

Age **18** **18** 20 67 22	**B1**
22 24 30	B2

LET’S START WITH DBMS :)

Indexing and its types

- How to create indexes

Non-Clustered index:

(On one column)

```
CREATE INDEX index_name
ON table_name (column_name);
```

(On multiple column)

```
CREATE INDEX index_name
ON table_name (column_name1, column_name2,.....)
```

Clustered index:

Automatically created with the primary key. MySQL does not support explicit creation of additional clustered indexes.

- Remove an index

```
DROP INDEX  index_name ON table_name
```

LET’S START WITH DBMS :)

B and B+ trees

Clustered and non-clustered indexes are concepts that describe how data is stored and accessed in a database, but B-trees (and B+ trees) are the data structures that actually implement these indexes. Knowing B-trees gives you insight into how these indexes work under the hood.

Understanding B-trees helps you understand why certain queries perform well or poorly based on the structure of the index. For example, how a B-tree's balanced nature affects search times or why range queries are efficient with B-tree indexes

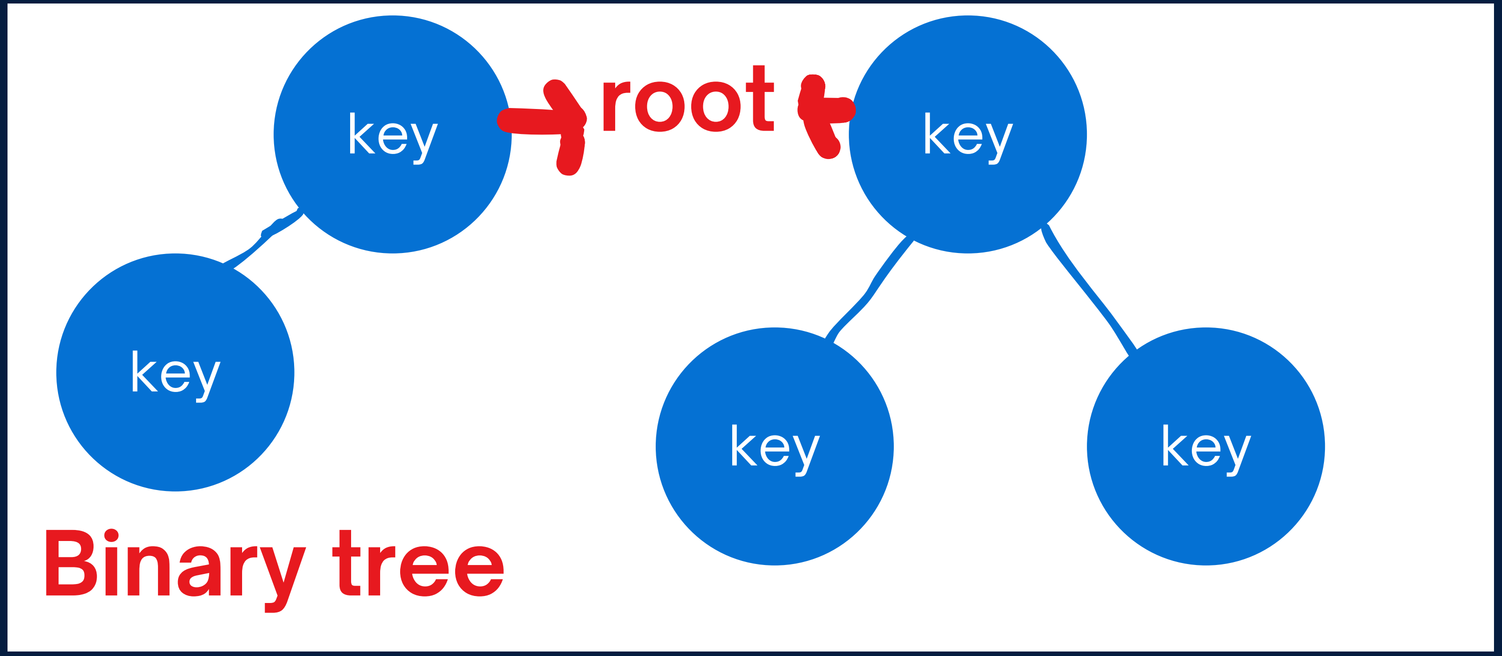
B-trees provide the balanced, efficient structure that makes these types of indexes performant, ensuring that operations like search, insert, delete, and update are done in logarithmic time (O(log n)). This makes B-tree indexing crucial for optimizing database queries, whether in clustered or non-clustered index scenarios.

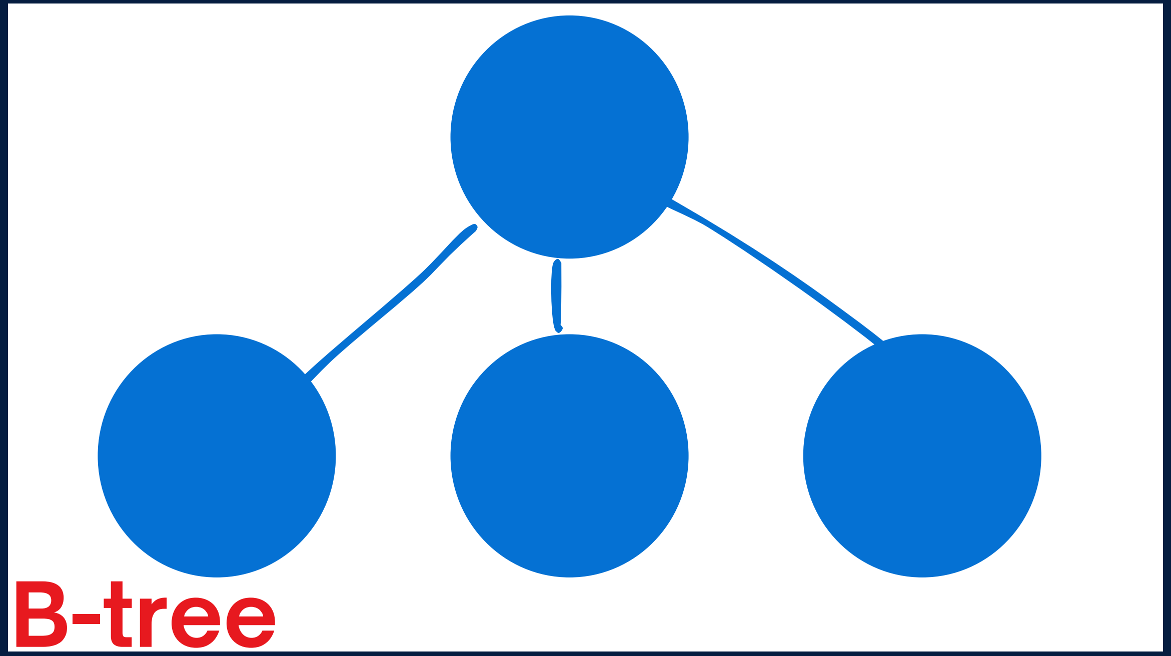
LET'S START WITH DBMS :)

B trees(M-way tree)

B-trees are self-balancing tree data structures that maintain sorted data and allow searches, sequential access, insertions, and deletions in logarithmic time. All leaf nodes are at the same level.

In B trees you can have minimum 2 children and max x children.
Now B-tree is a generalisation of Binary Search trees. In BST every node can have atmost 2 children(0,1,2) and only one key

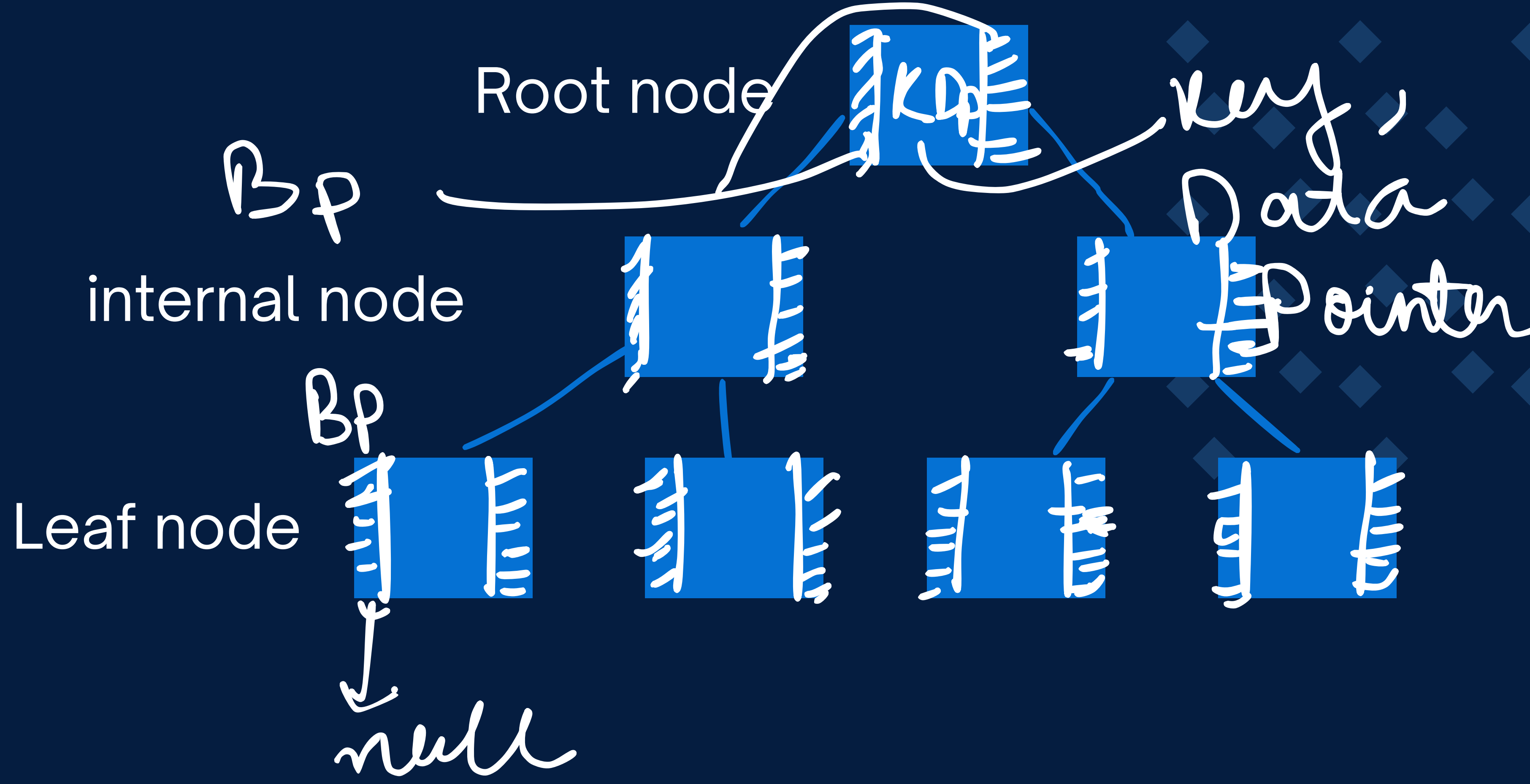




LET'S START WITH DBMS :)

B trees

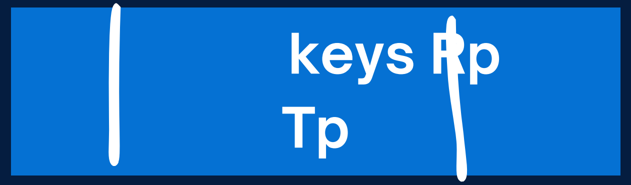
(Rp)Dp-> record/data pointer(where the record is present in secondary memory(disk)
(Bp)Tp-> block/tree pointer(links to the children nodes)



Structure of B-Tree:

- Nodes: A B-tree is composed of nodes, each containing keys and pointers (references) to child nodes. The keys within a node are sorted in ascending order.
- Root, Internal Nodes, and Leaves:

1. The root node is the topmost node in the tree.
2. Internal nodes contain keys and child pointers.
3. Leaf nodes contain keys and possibly pointers to records or other data.



- In indexing, each key in a B-tree node typically represents a value or range of values, and the associated pointer directs to a data block where records corresponding to the key(s) can be found.
- For example, in a database, the key might be a value in a column, and the pointer might direct to the location of a row or a set of rows in a table.

LET'S START WITH DBMS :)

B trees

A B-tree of order x can have:

- **The max no. of children**

For every node -> x i.e the order of tree.

- **The max no. of keys**

For every node -> x-1

- **The min no of keys**

a. root node -> 1

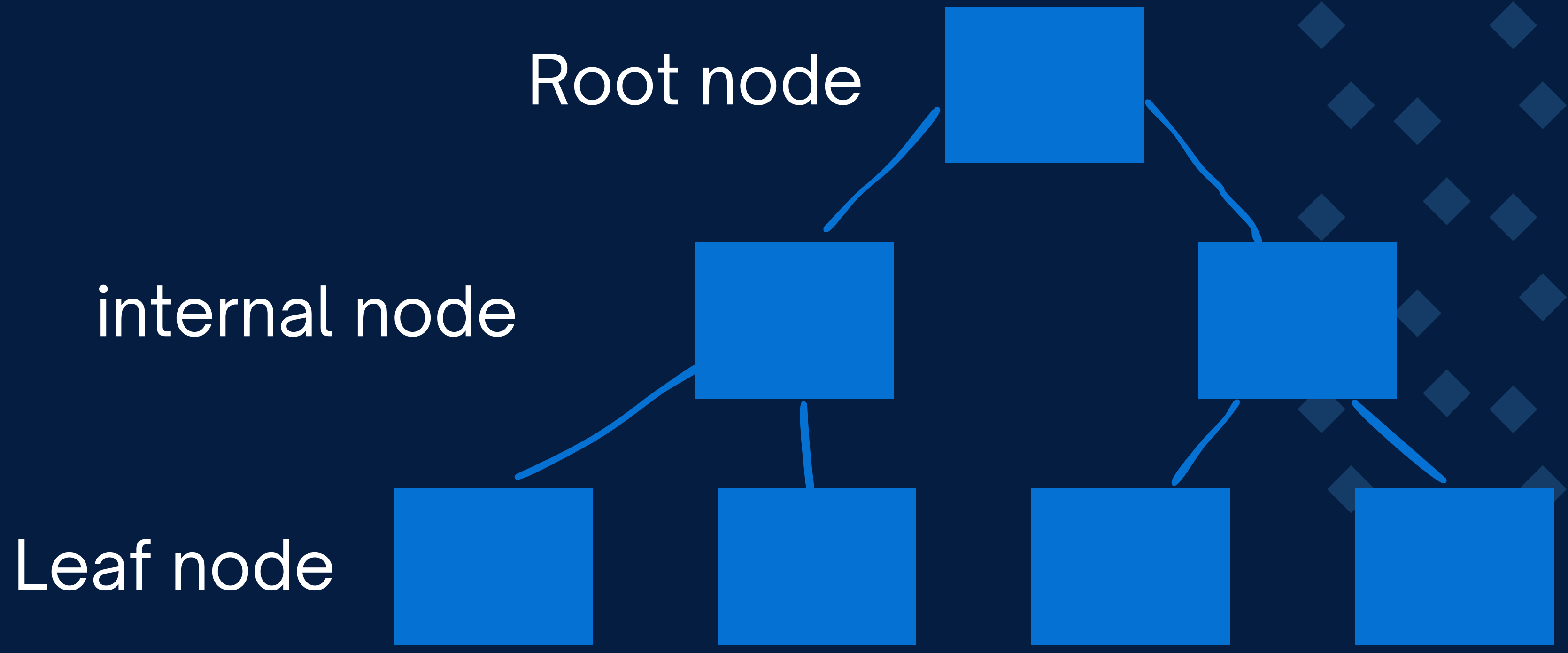
b. other nodes apart from root-> ceiling(m/2) -1

- **The min no. of children**

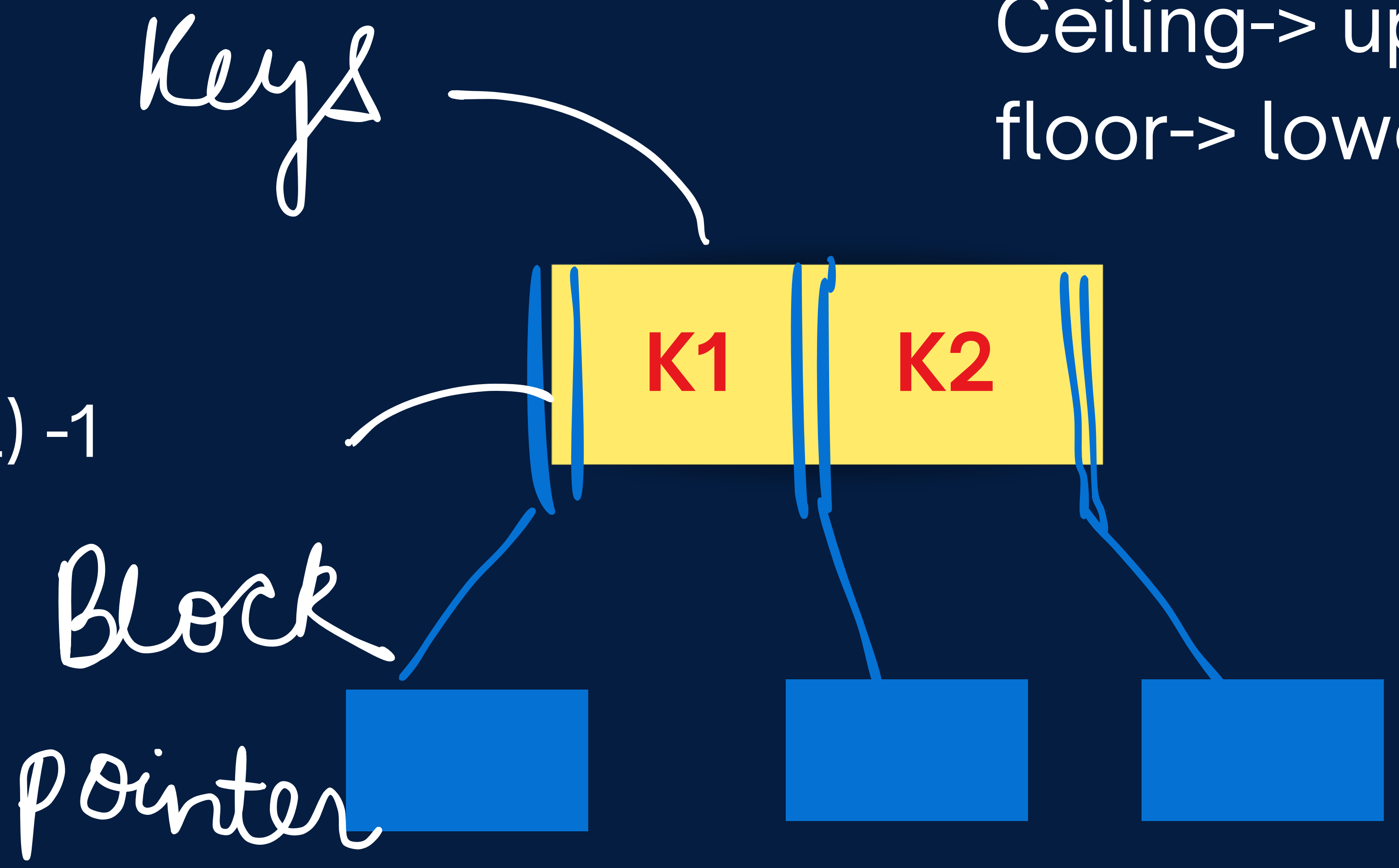
a. root node -> 2

b. Leaf node ->0

c. internal node-> ceiling(m/2)



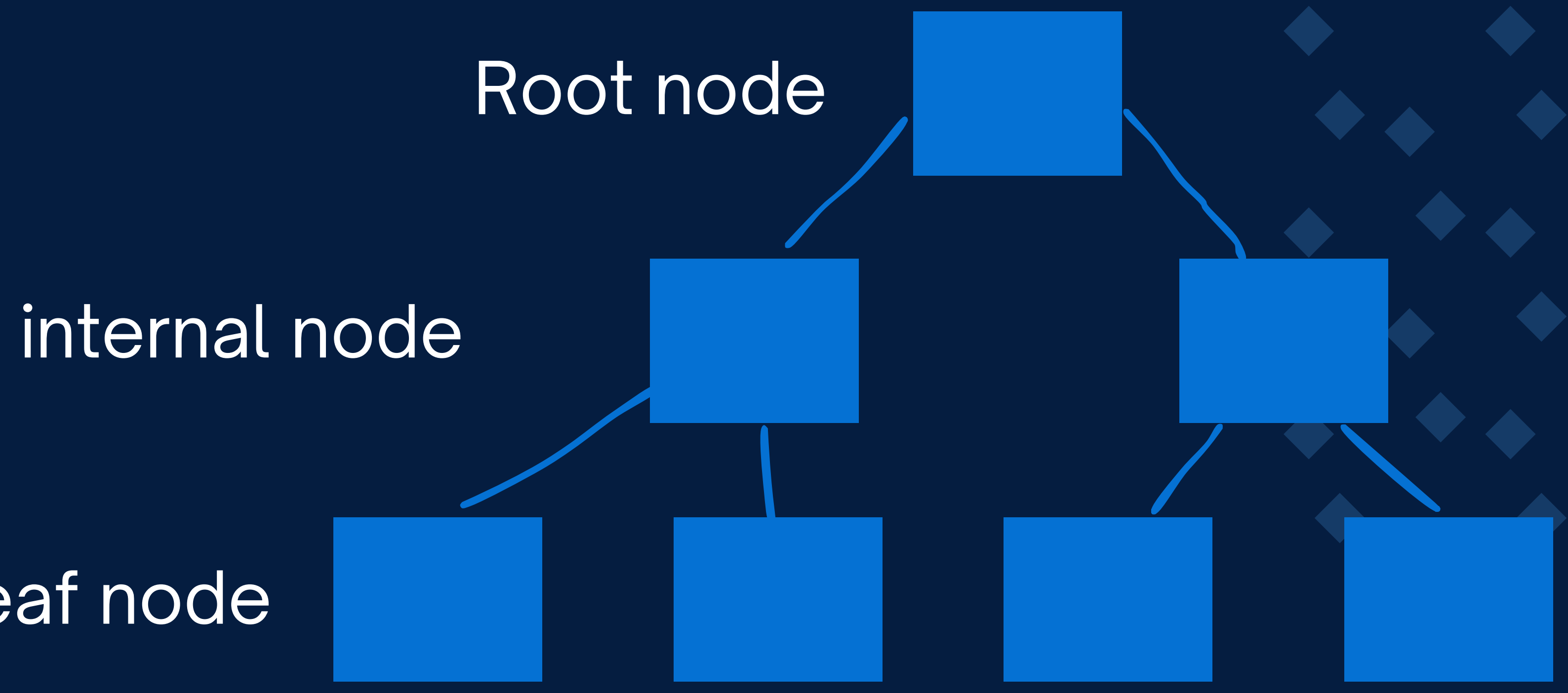
Ceiling-> upper value
floor-> lower value



Insertion in B-tree always happens from leaf node.

LET'S START WITH DBMS :)

B trees



To determine the order of a B-tree when the block size, block pointer size, and data pointer size are given :

$$m \times Pb + (m-1) \times (K + Pd) \leqslant B$$

B- Block size
m- Order of tree
Pb- Block pointer size
K- Key size
Pd- Data pointer size

For a B-tree node with m children:

- Number of Keys: A node with m children can have a maximum of m-1 keys.
- Number of Block Pointers: Each node has m block pointers (pointers to child nodes).
- Number of Data Pointers: Each key has an associated data pointer, so there are m-1 data pointers.

LET'S START WITH DBMS :)

B trees

Let's say you have the following:

- Block size (B) = 1024 bytes
- Block pointer size (Pb) = 8 bytes
- Data pointer size (Pd) = 12 bytes
- Key size (K) = 16 bytes

Find the order of the tree.

Formula to find order : $m \times Pb + (m-1) \times (K+Pd) \leqslant B$
$mx8+(m-1)x(16+12) <= 1024$

So, the order of the B-tree is m = 29

To determine the order of a B-tree when the block size, block pointer size, and data pointer size are given :

$m \times Pb + (m-1) \times (K+Pd) \leqslant B$

B- Block size
m- Order of tree
Pb- Block pointer size
K- Key size
Pd- Data pointer size

LET'S START WITH DBMS :)

Insertion in B-TREE

Steps to insert values in B-tree

1.Start at the root and recursively move down the tree to find the appropriate leaf node where the new value should be inserted.

2. Insert the value into the leaf node in sorted order. If the leaf node has fewer than the maximum allowed keys (order - 1), this step is simple.

3. If the leaf node contains the maximum number of keys after the insertion, it causes an overflow.
Split the Node:
Divide the node into two nodes. The middle key (median) is pushed up to the parent node.

- The left half of the original node stays in place, while the right half forms a new node.
- Insert the Median into the Parent:
- If the parent node also overflows after this insertion, recursively split the parent node and propagate the median up the tree.

4.If the root node overflows (which can happen if it already has the maximum number of keys), split it into two nodes, and the median becomes the new root. This increases the height of the B-tree by one.

LET'S START WITH DBMS :)

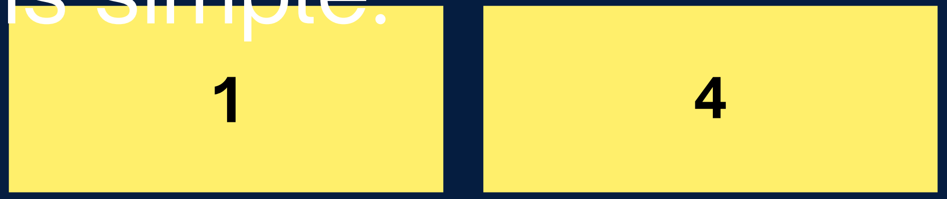
Insertion in B-TREE

Create a B-tree of Order 3 and insert values from 1,4,6,8,10,12,14,16

Max children= order of tree=3
Max keys node can have=order-1=3-1=2

Insertion in B-tree happens from leaf node and values are also inserted in sorted order. The element in left of root would be less than root and the element in right would be greater than root

Step 1: Start at the root and recursively move down the tree to find the appropriate leaf node where the new value should be inserted. Insert the value into the leaf node in sorted order. If the leaf node has fewer than the maximum allowed keys (order - 1), this step is simple.



A B-tree of order x can have:

- Every node will have max x children i.e the order of tree.
- Every node can have max (x-1) keys
- The min no of keys

a. root node -> 1
b. other nodes apart from root-> ceiling(m/2) -1

- Min children

a. root node -> 2
b. Leaf node ->0
c. internal node-> ceiling(m/2)

LET'S START WITH DBMS :)

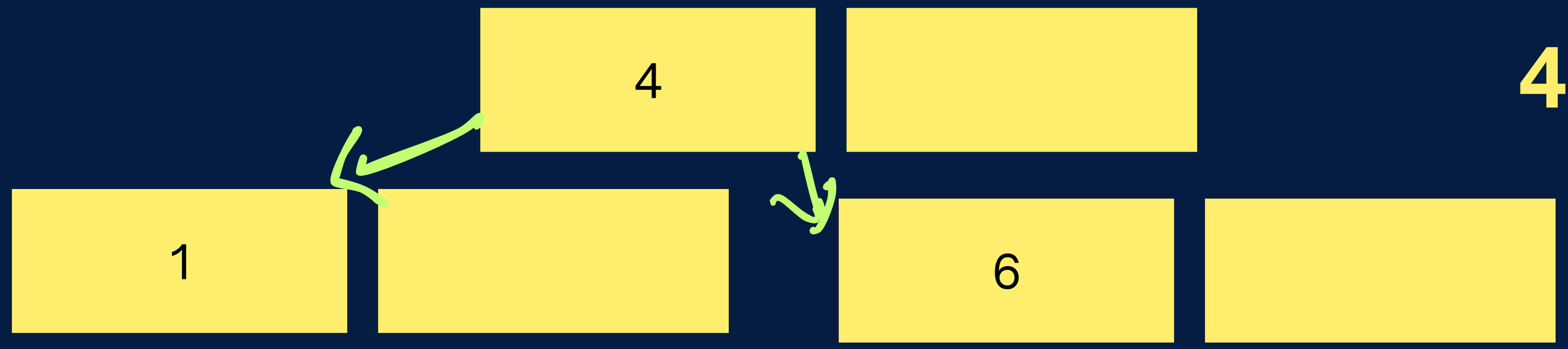
Max children= order of tree=3
Max keys node can have=order-1=3-1=2

Insertion in B-TREE

Create a B-tree of Order 3 and insert values from 1,4,6,8,10,12,14,16

Step 2: If the leaf node contains the maximum number of keys after the insertion, it causes an overflow.
Split the Node: Divide the node into two nodes. The middle key (median) is pushed up to the parent node.

- The left half of the original node stays in place, while the right half forms a new node.
- Insert the Median into the Parent:
- If the parent node also overflows after this insertion, recursively split the parent node and propagate the median up the tree.



4 (left- 1 , right- 6)

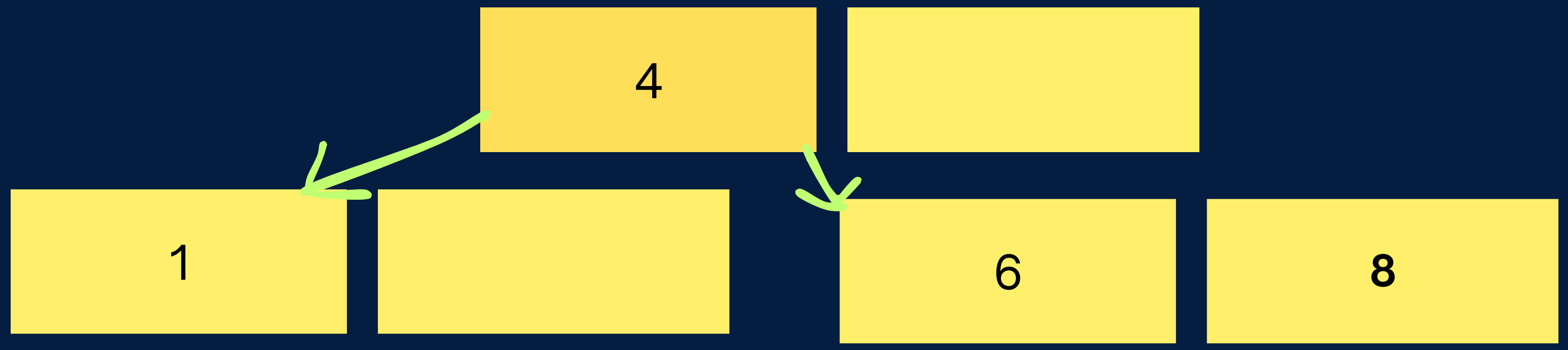
Max children= order of tree=3

Max keys node can have=order-1=3-1=2

LET'S START WITH DBMS :)

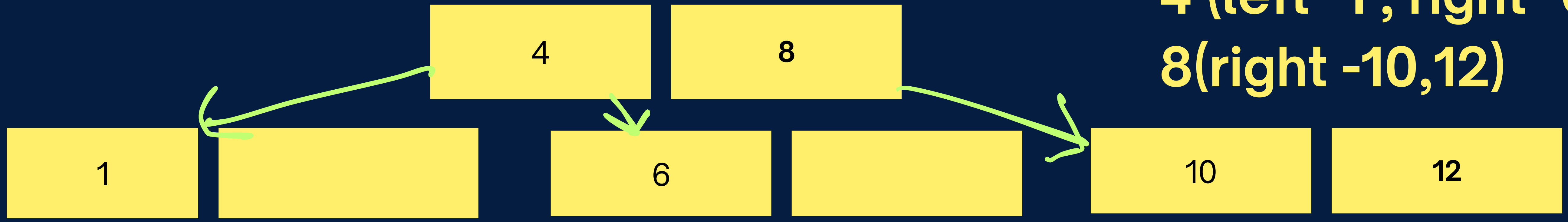
Insertion in B-TREE

Create a B-tree of Order 3 and insert values from 1,4,6,8,10,12,14,16



4 (left- 1 , right- 6,8)

Follow step-2 again



4 (left- 1 , right- 6)

8(right -10,12)

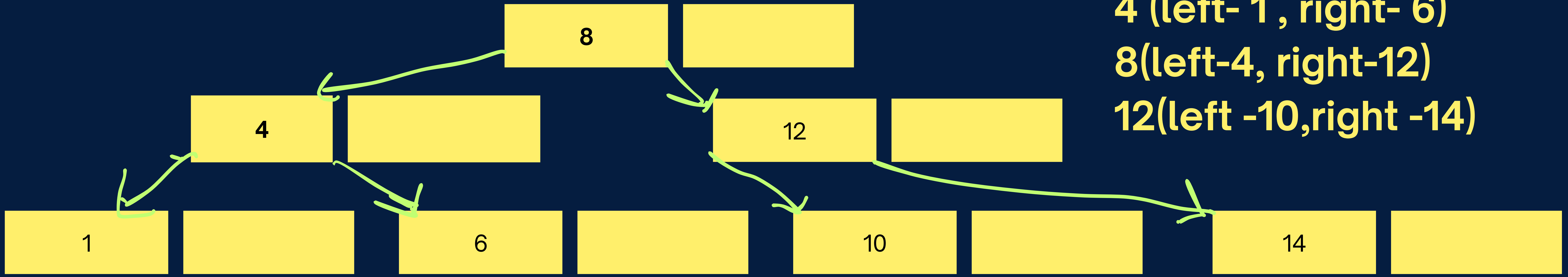
Max children= order of tree=3
Max keys node can have=order-1=3-1=2

LET'S START WITH DBMS :)

Insertion in B-TREE

Create a B-tree of Order 3 and insert values from 1,4,6,8,10,12,14,16

Step 3: If the root node overflows (which can happen if it already has the maximum number of keys), split it into two nodes, and the median becomes the new root. This increases the height of the B-tree by one.



4 (left- 1 , right- 6)
8(left-4, right-12)
12(left -10,right -14)

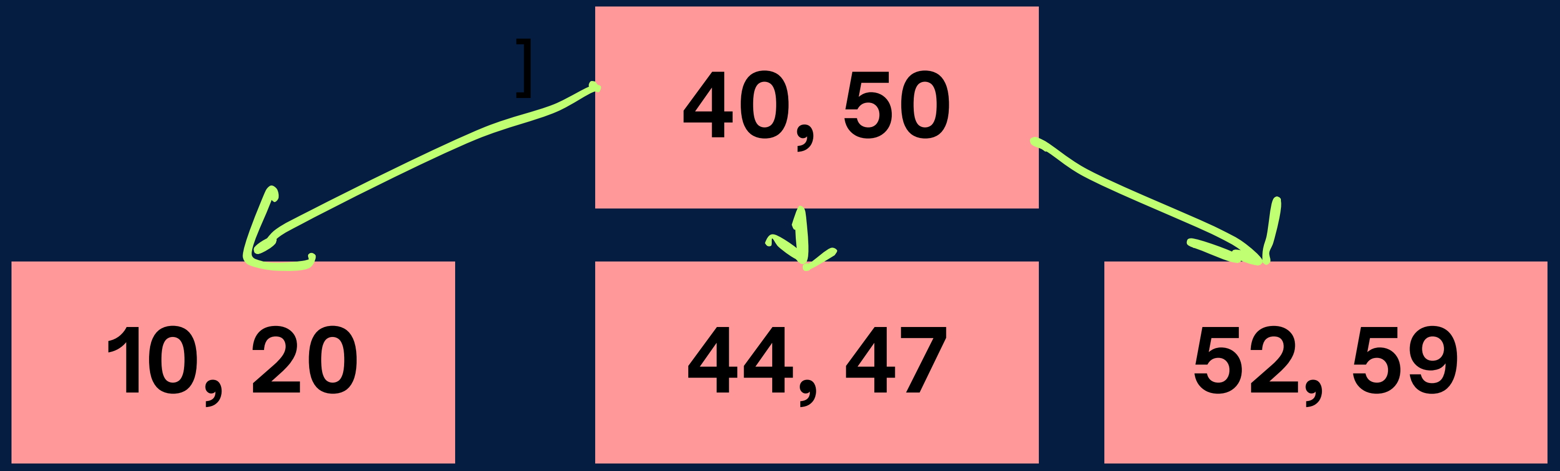
LET'S START WITH DBMS :)

Deletion in B-TREE

maximum of (m-1)=3 keys
and minimum of (ceil(m/2)-1)=1 key

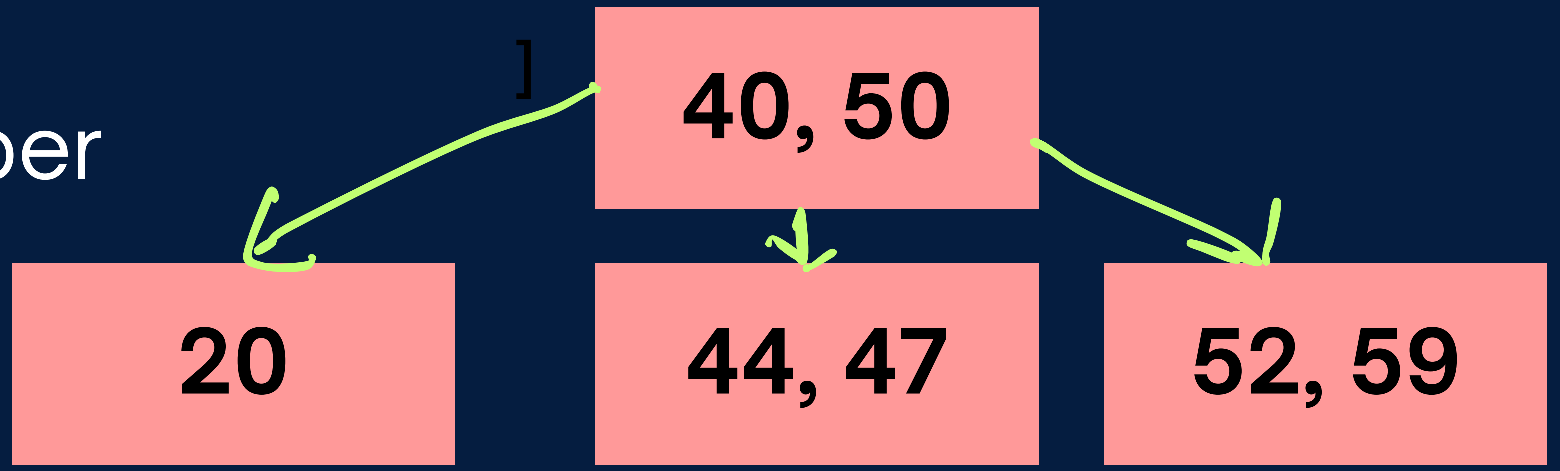
Consider a B-tree of order 4

Step 1: Begin at the root and recursively move down the tree to find the node that contains the key to be deleted.



Step 2 :
Case 1: The Key is in a Leaf Node **(Delete 10)**

- Simply remove the key from the leaf node.
- If the node still has the minimum required number of keys (i.e., at least ceil(order/2) - 1 keys), the deletion is complete.



LET'S START WITH DBMS :)

Deletion in B-TREE

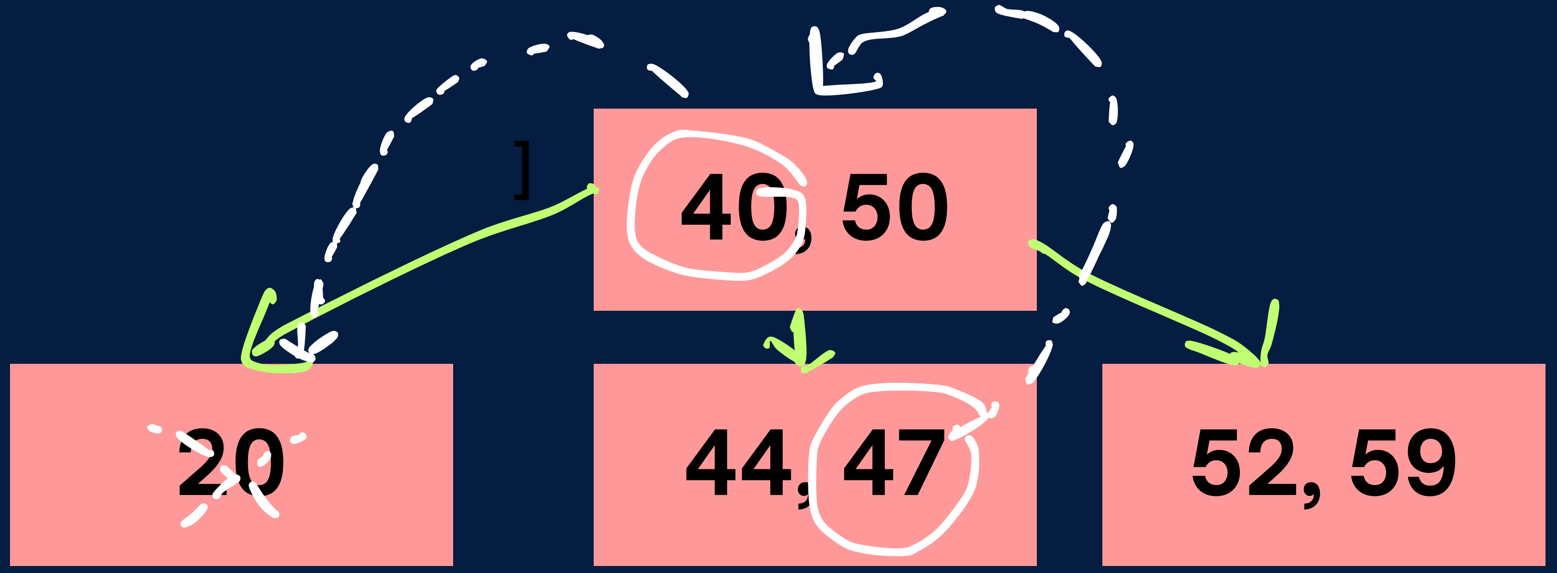
Consider a B-tree of order 4

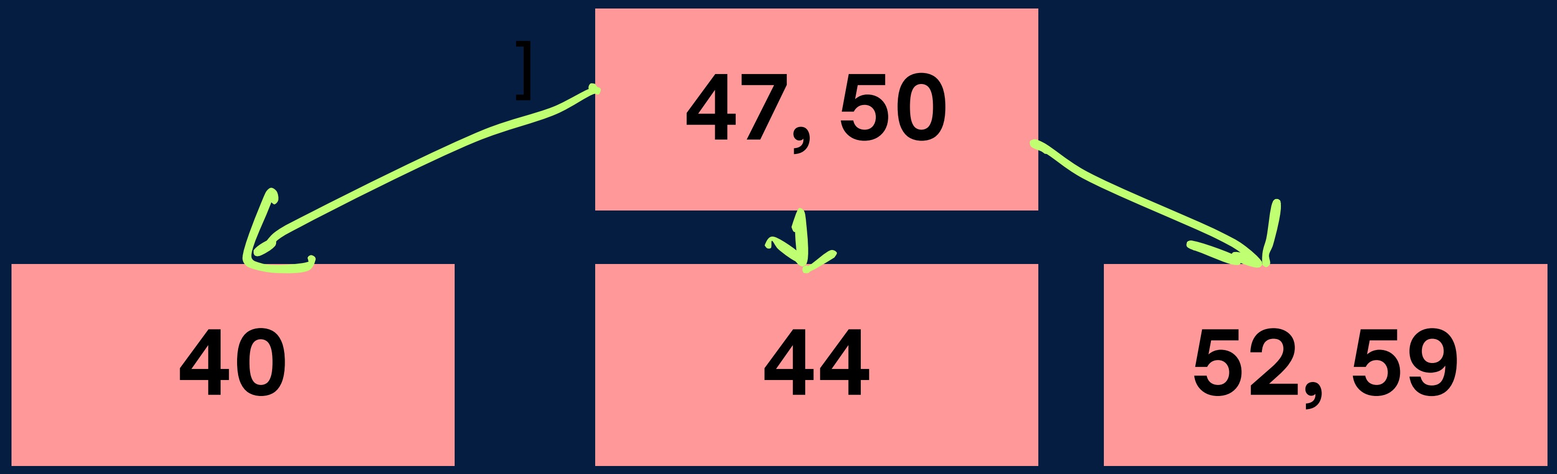
maximum of (m-1)=3 keys
and minimum of (ceil(m/2)-1)=1 key

Step 2 :
Case 2: The Key is in a Leaf Node **(Delete 20)**

- If the deletion causes the node to have fewer than the minimum number of keys, proceed to the Borrowing or Merging step.

1. If the node has a sibling with more than the minimum number of keys, you can borrow a key from this sibling. The parent key between the node and the sibling moves down to the node, and a key from the sibling moves up to the parent.





LET'S START WITH DBMS :)

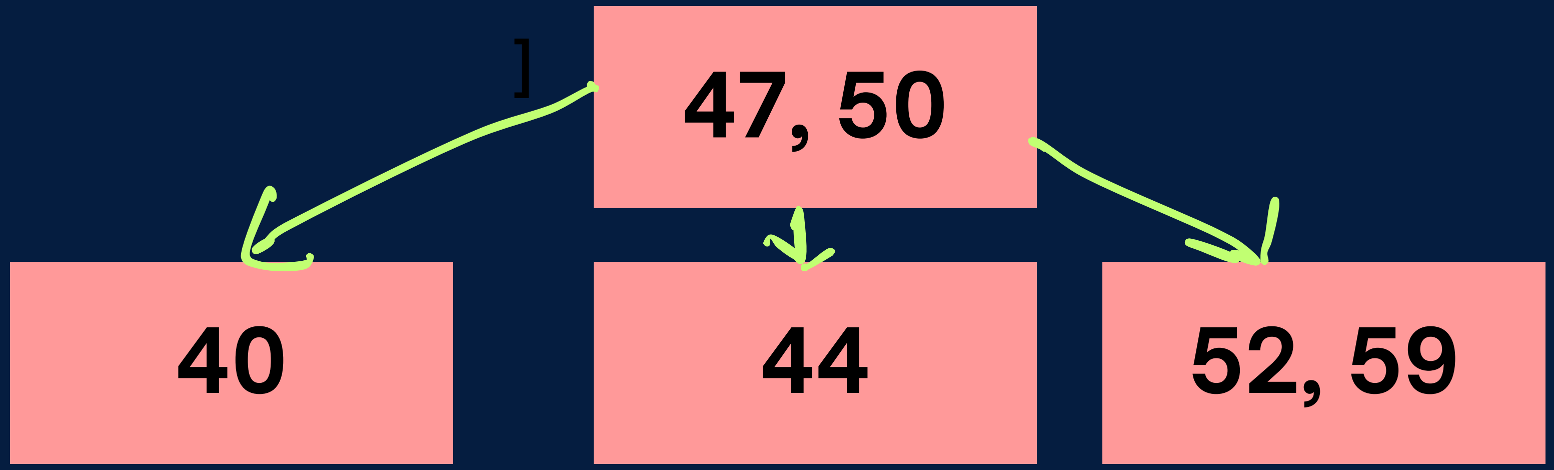
Deletion in B-TREE

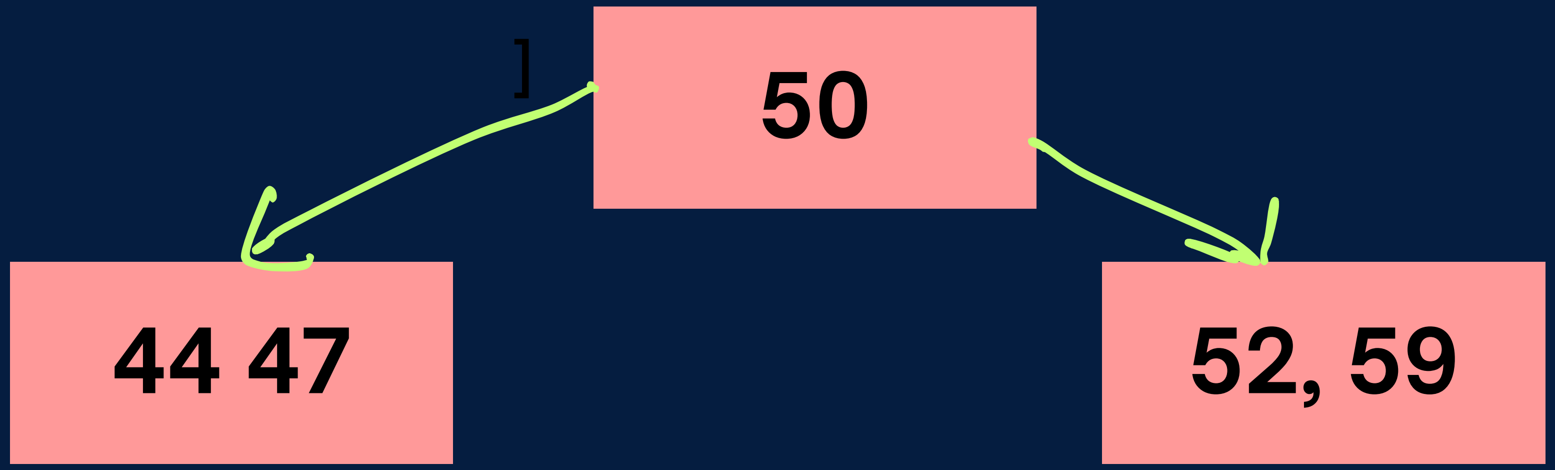
Consider a B-tree of order 4

Step 2 :
Case 2: The Key is in a Leaf Node **(Delete 40)**

2. If borrowing is not possible (i.e., the sibling also has the minimum number of keys), merge the node with a sibling. The key from the parent that separates the two nodes moves down into the newly merged node.If this causes the parent to have too few keys, repeat the borrowing or merging process at the parent level.

maximum of (m-1)=3 keys
and minimum of (ceil(m/2)-1)=1 key

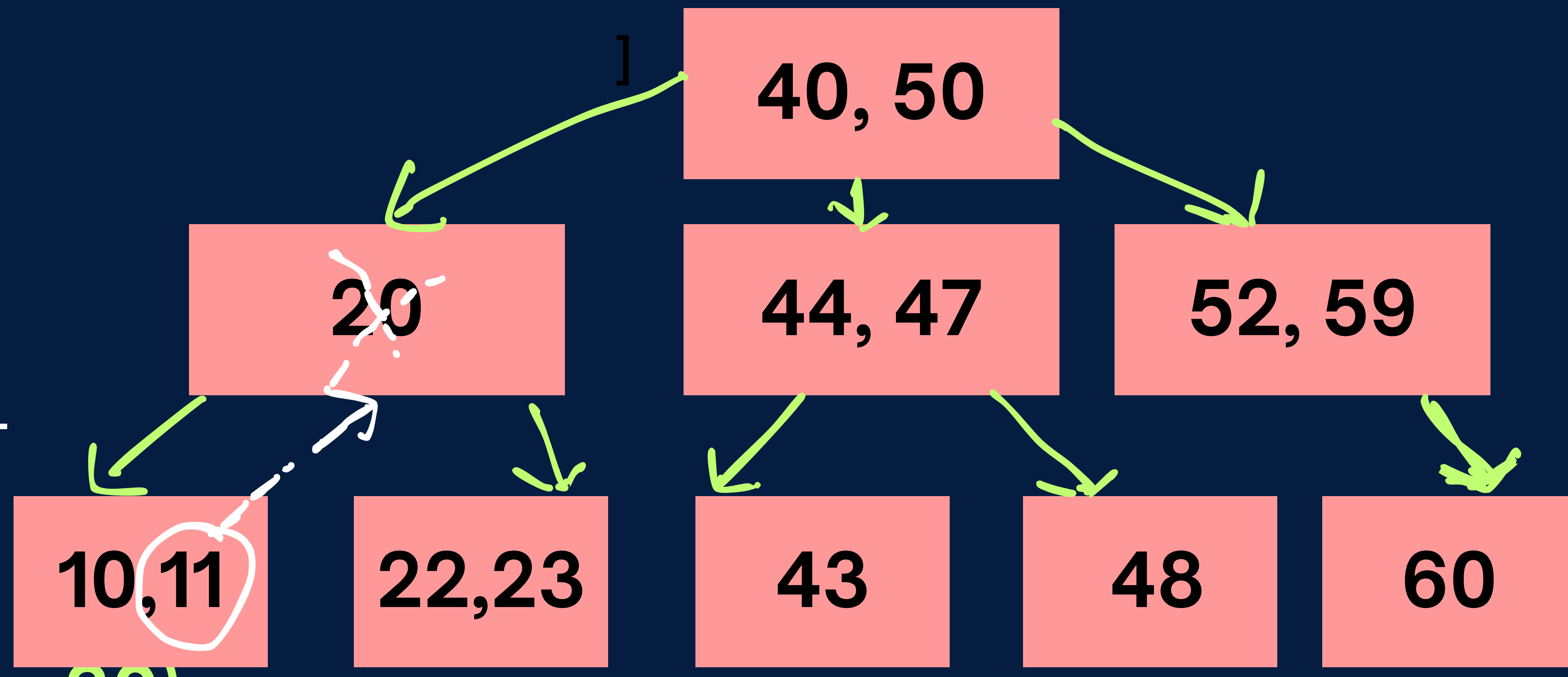




LET'S START WITH DBMS :)

Deletion in B-TREE

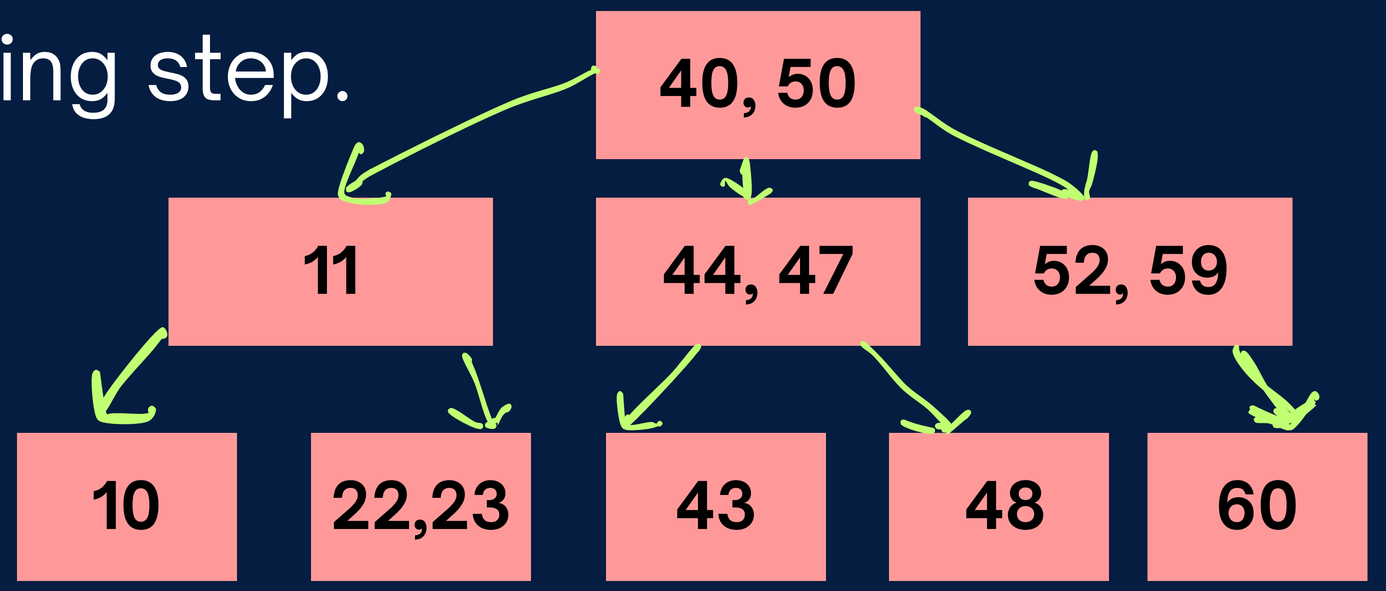
Steps on how we can delete elements in B-tree



Step 2:

Case 2: The Key is in an Internal Node **(Delete 20)**

- Replace the key with its predecessor (the largest key in the left subtree) or its successor (the smallest key in the right subtree).
- Delete the predecessor or successor key from the corresponding subtree.
- If this causes an underflow (i.e., a node has too few keys), proceed to the Borrowing or Merging step.



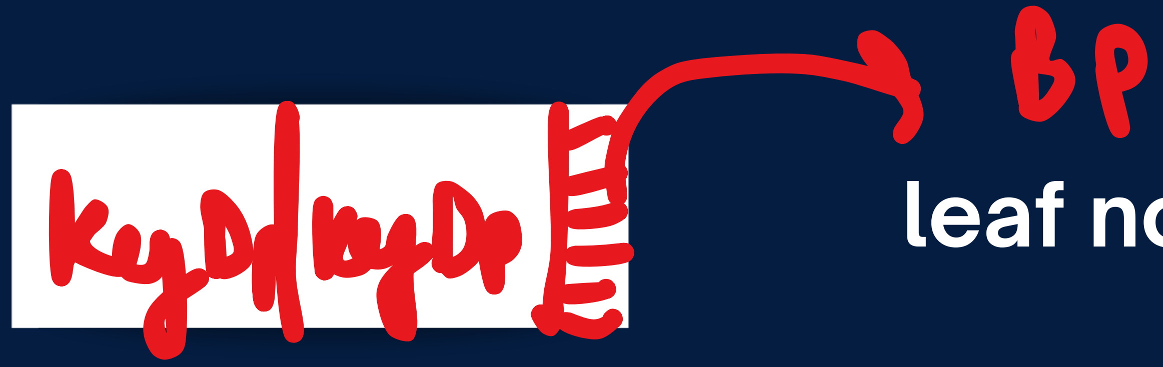
LET'S START WITH DBMS :)

B+ Tree

A B+ tree is an extension of the B-tree and is commonly used in databases and file systems to maintain sorted data and allow for efficient insertion, deletion, and search operations. B+ tree is a balanced tree, meaning all leaf nodes are at the same level

The key difference between a B+ tree and a B-tree lies in how they store data and how leaf nodes are structured.

1. In a B+ tree, all actual data (or references to data) are stored in the leaf nodes. Internal nodes only store keys

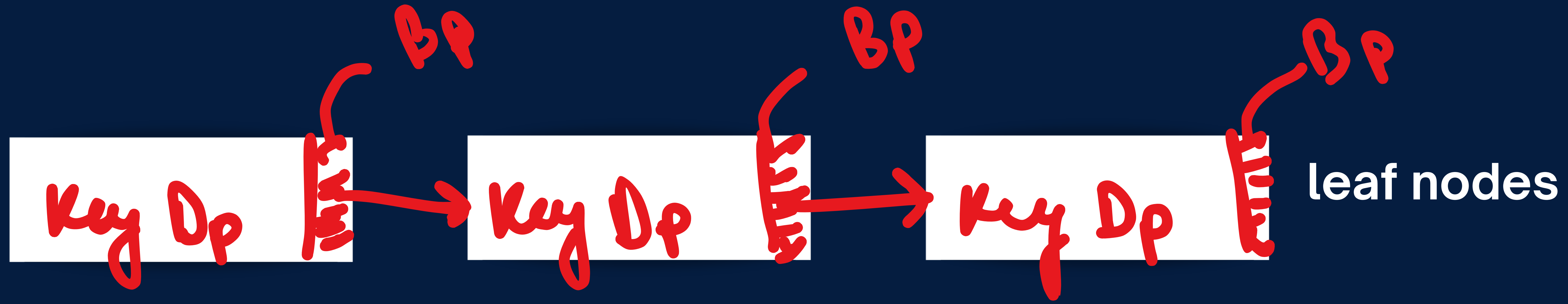


leaf node

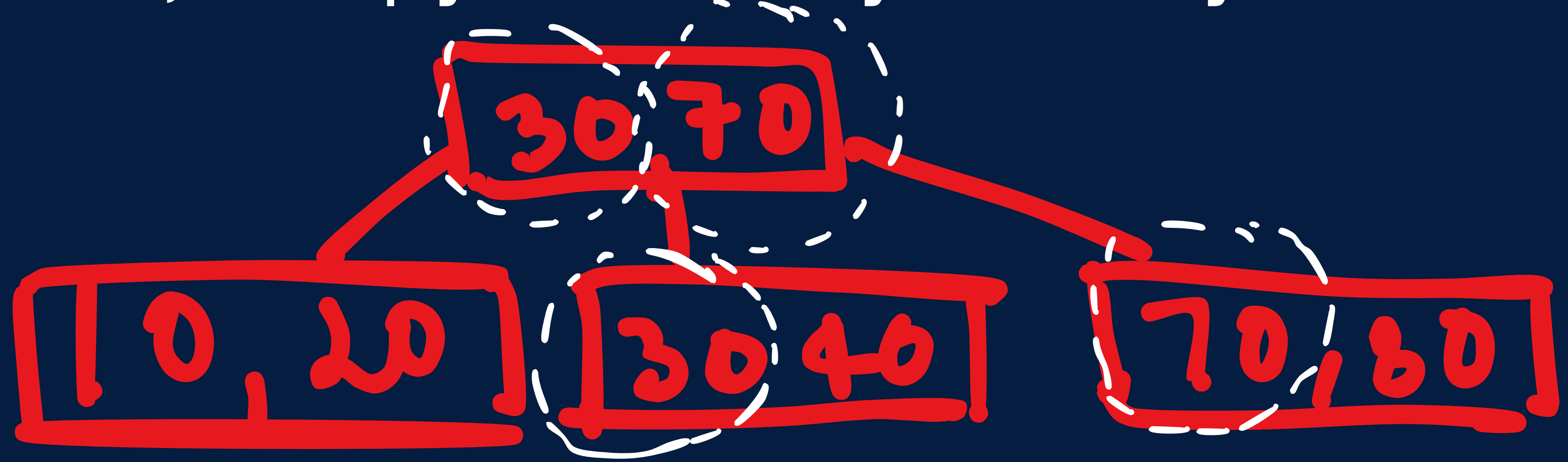
LET'S START WITH DBMS :)

B+ Tree

2. In a B+ tree, Leaf nodes are linked together in a linked list fashion, allowing for efficient sequential access, one leaf node will have only 1 Bp.



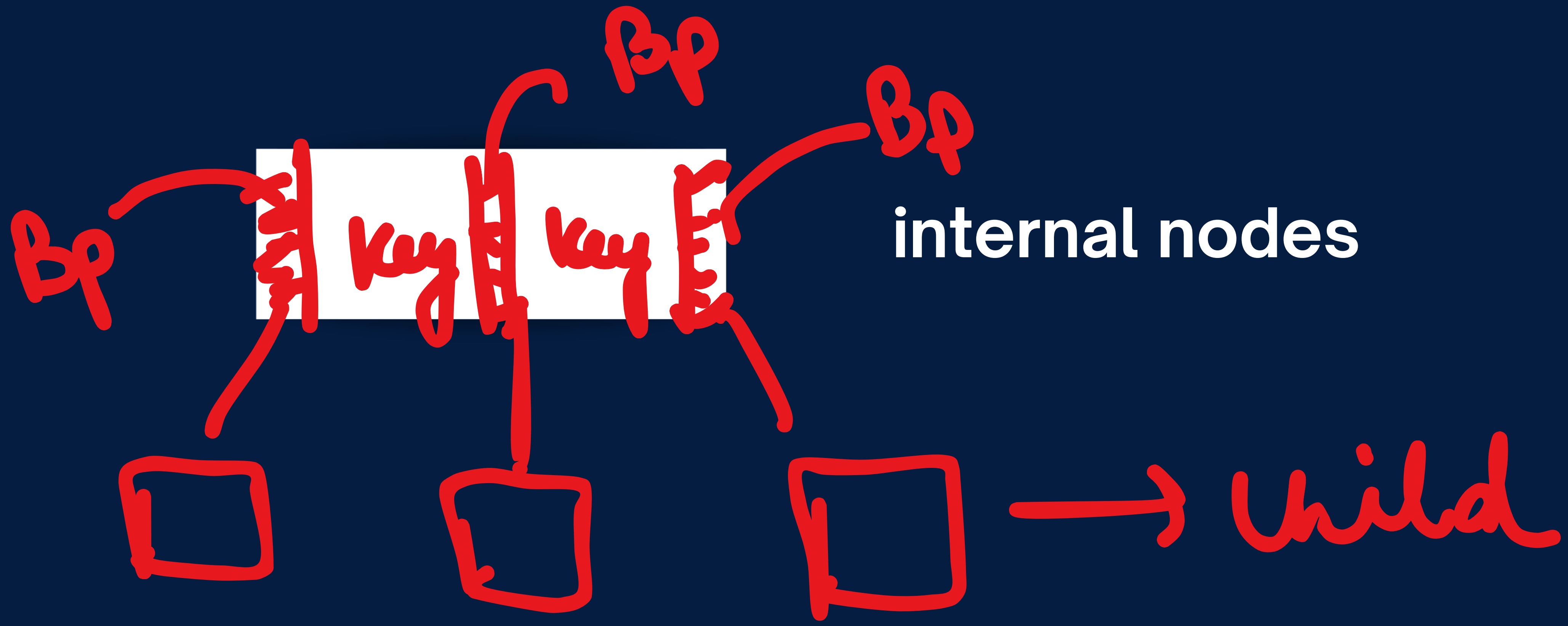
During inserting value in B+ tree, a copy of the key is always stored in the leaf node.



LET'S START WITH DBMS :)

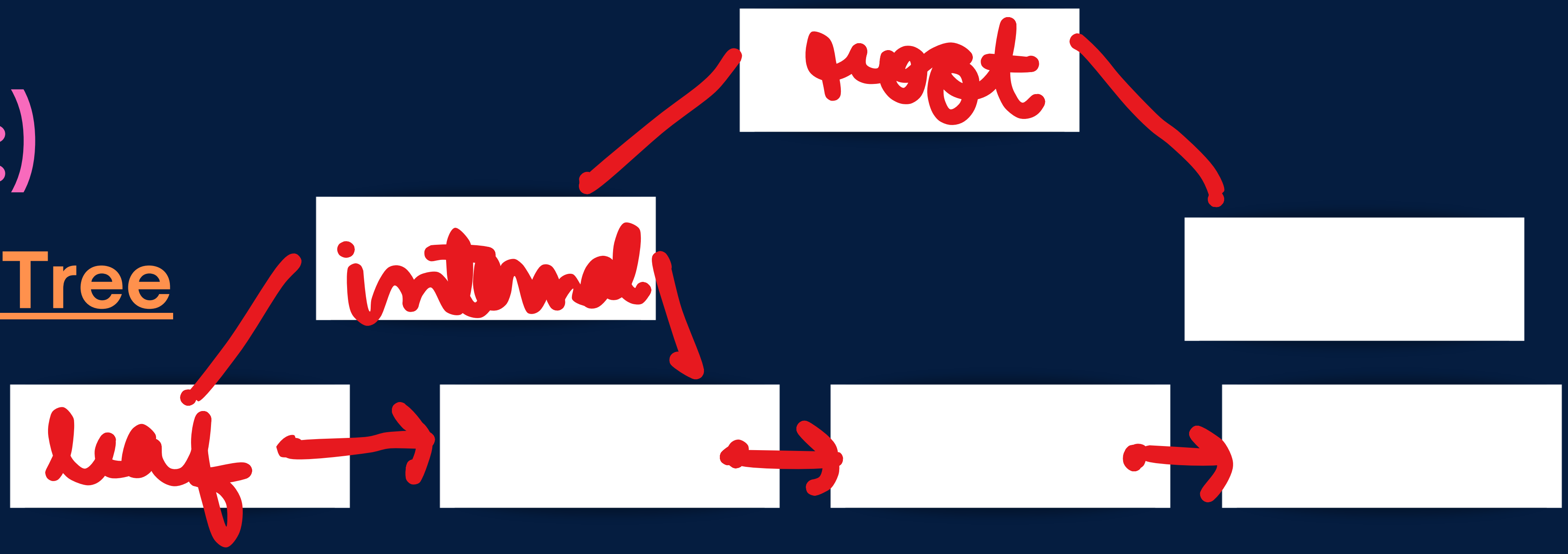
B+ Tree

3. Internal nodes do not store data pointers, only keys and child pointers. This allows more keys to be stored in each internal node, leading to a lower height and more efficient operations.



LET'S START WITH DBMS :)

B+ Tree



Structure of a B+ Tree:

- Root Node:
    - The top node of the B+ tree, which points to the first level of internal nodes or directly to leaf nodes if the tree has only one level.
- Internal Nodes:
    - These nodes contain only keys and pointers to child nodes. They guide the search process down to the correct leaf node.
- Leaf Nodes:
    - These nodes contain keys and data pointers. Each leaf node stores a pointer to the next leaf node, enabling quick traversal of records.

LET'S START WITH DBMS :)

B+ Tree

Advantages :

1. B+ trees have a balanced structure, meaning all leaf nodes are at the same level. This balance ensures that search operations require logarithmic time relative to the number of keys, making it very efficient even for large datasets.
2. The structure of the B+ tree allows for direct access to data. Since the internal nodes contain only keys, searching for a specific value can be done quickly by navigating through the tree down to the leaf node where the data is stored.

LET'S START WITH DBMS :)

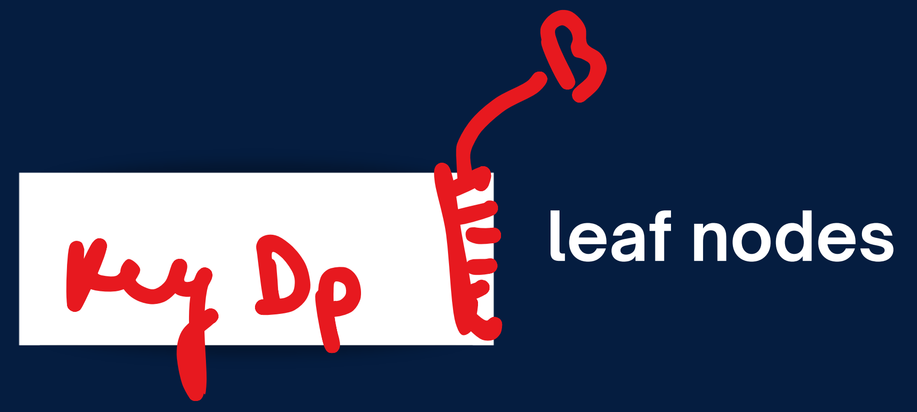
B- Block size
m- Order of tree
Pb- Block pointer size
K- Key size
Pd- Data pointer size

B+ Tree

Order of B+ tree

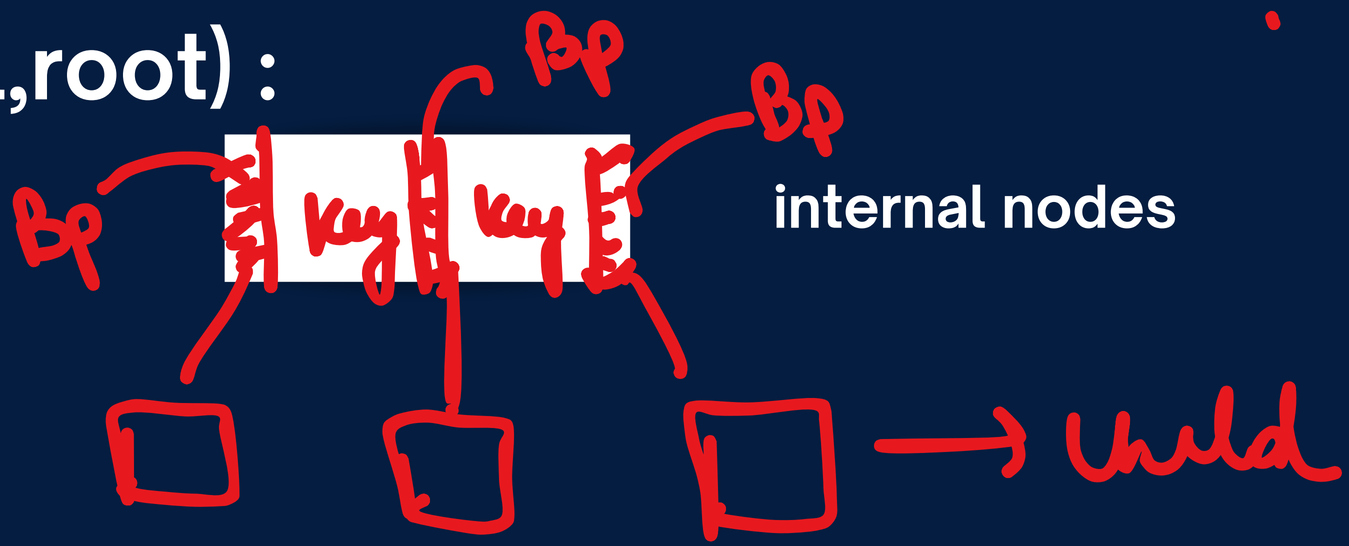
Order of Leaf node :

1xPb+M(k+Pd)<=B



Order of non-Leaf node(internal,root) :

mxPb+(m-1)k<=B



LET’S START WITH DBMS :)

Difference between B and B+ Tree

- B-Tree Example: Think of a B-Tree as a directory structure on your computer where folders (nodes) contain both names (keys) and pointers to subfolders or files (child pointers).

- B+ Tree Example: Imagine a library catalog where all book records are listed in a sorted, linked list (leaf nodes), and the internal nodes only contain pointers to guide you to the right part of the catalog.

LET’S START WITH DBMS :)

Difference between B and B+ Tree

Case	B tree	B+ tree
Data Storage	keys and associated pointers to data are stored in all nodes (internal and leaf nodes)	all actual data is stored only in the leaf nodes.Internal nodes contain only keys and pointers to child nodes
Leaf Node Linking	There is no inherent linked structure between the leaf nodes	Leaf nodes are linked together in a linked list. This linked structure allows for efficient sequential access, making range queries faster and easier
Search Performance	Access times can be slower for certain types of queries because you may need to traverse multiple levels of the tree to find the data.	Leaf nodes are linked, allowing for efficient sequential access and range queries.
Space Utilization	Since data is stored throughout the tree, B-trees might have lower space utilization in the nodes.	B+ trees typically have better space utilization because internal nodes are used only for keys, allowing more keys to be stored per node.

LET’S START WITH DBMS :)

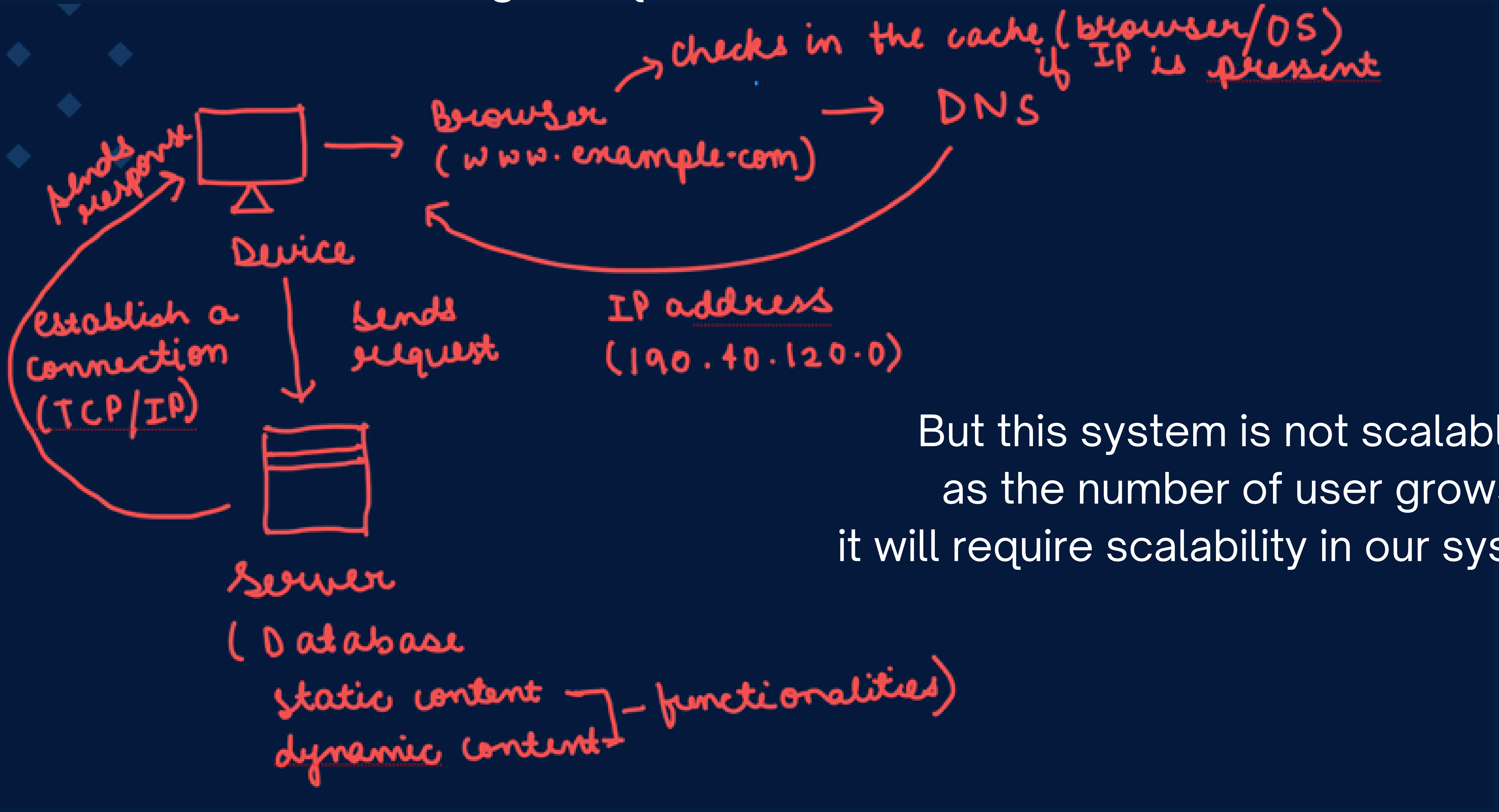
Scaling in Databases

Process of user sending a request to access data on the internet

LET'S START WITH DBMS :)

Scaling in Databases

Process of user sending a request to access data on the internet



But this system is not scalable
as the number of user grows
it will require scalability in our systems

LET’S START WITH DBMS :)

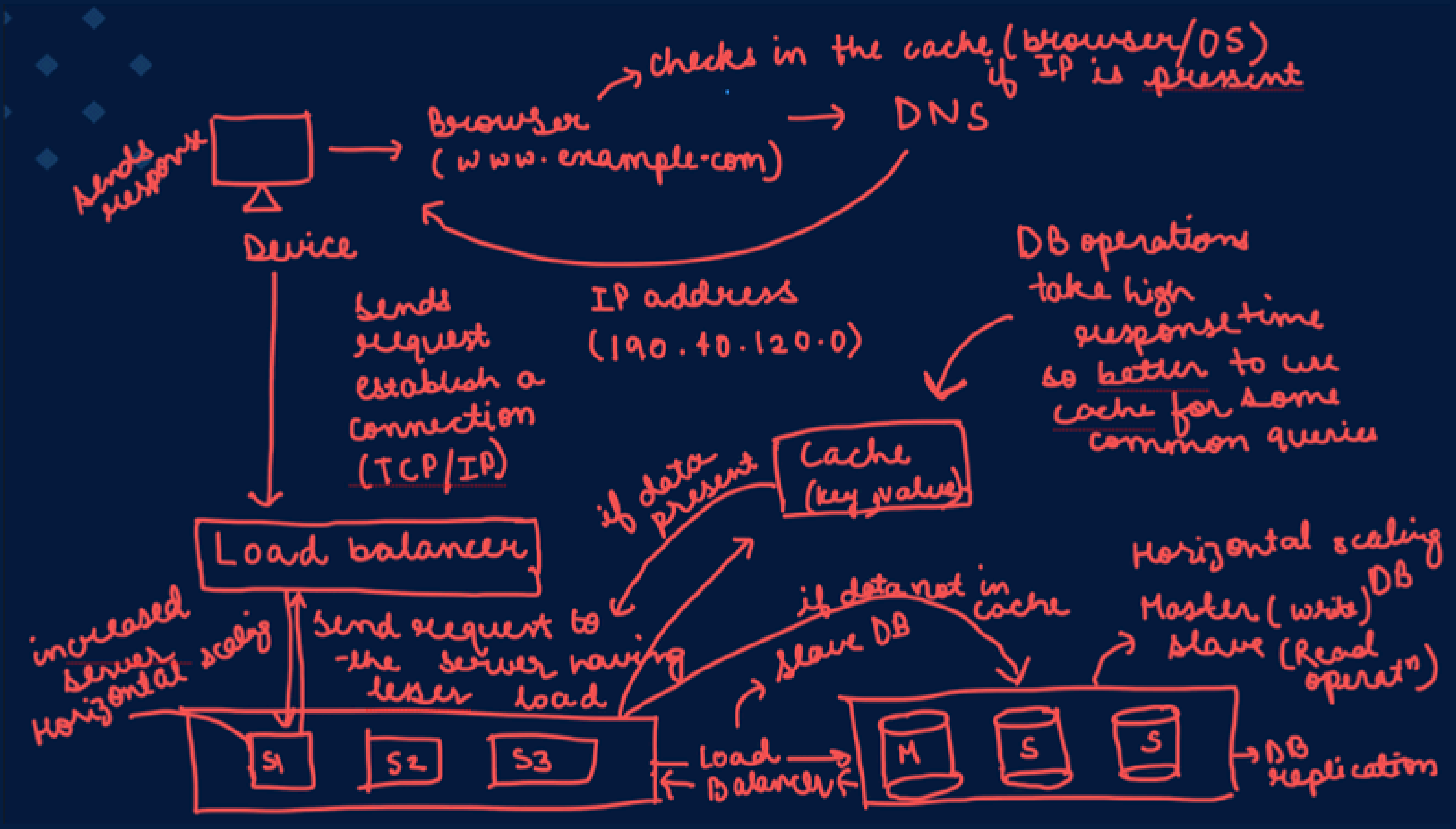
Scaling in Databases

How to scale a Database?

LET’S START WITH DBMS :)

Scaling in Databases

How to scale a Database?



LET'S START WITH DBMS :)

Scaling in Databases

User Journey

1. User Action: Sending a Request
2. Browser Parses the URL
3. DNS Lookup
4. Establishing a Connection
5. Sending the HTTP Request
6. Server Processing
7. Browser Receives the Response
8. Displaying the Web Page
9. Caching

LET'S START WITH DBMS :)

Scaling in Databases

The user sends a request by entering a URL or clicking a link. The browser then translates the domain name into an IP address through DNS lookup, establishes a connection with the server, sends an HTTP request, and receives a response.

The browser then processes this response, rendering the webpage and displaying it to the user. The process involves multiple layers of networking, data transfer, and client-server interaction, all working together to deliver the desired content to the user's screen.

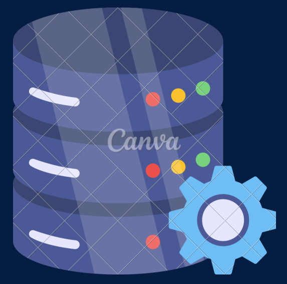
LET'S START WITH DBMS :)

Scaling in Databases

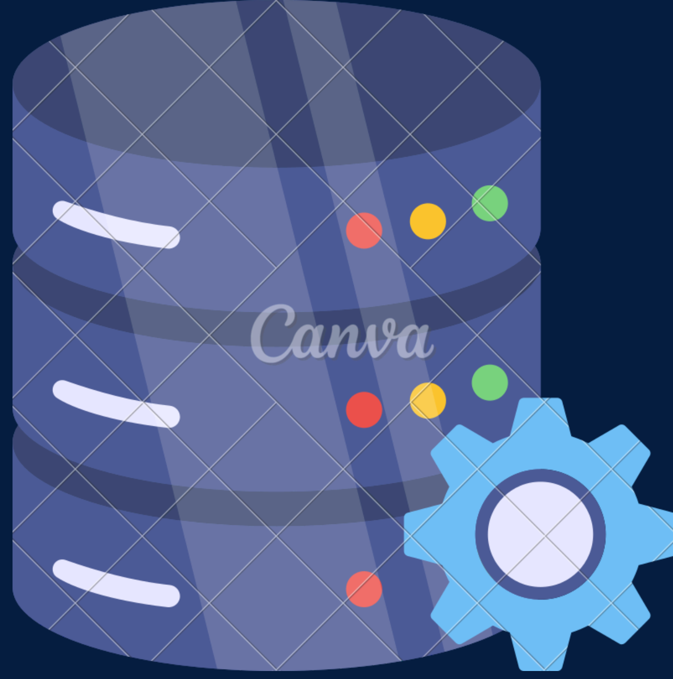
Scaling in databases is a crucial topic in development, mostly when applications grow and the volume of data increases.

- Vertical Scaling (Scaling Up)
- Horizontal Scaling (Scaling Out)
- Database Partitioning
- Replication
- Sharding
- Caching
- Load Balancing

LET'S START WITH DBMS :)



Increase the capacity of instances RAM,CPU



Scaling in Databases

Vertical Scaling (Scaling Up)

Vertical scaling involves increasing the capacity of a single database server.

- **Adding More Resources:** Upgrading the server by adding more RAM, faster processors, or larger and quicker storage. This helps the server handle more requests and store more data.

- **Improving Performance:** Tweaking the database settings, optimizing queries, and indexing tables to make better use of the hardware.

- **Upgrading to a Stronger Machine:** Moving the database to a more powerful server, like switching from a regular server to a high-performance cloud instance.

LET’S START WITH DBMS :)

Scaling in Databases

Vertical Scaling (Scaling Up)

Advantages:

- Easy to set up because it usually doesn't need changes to the app or database structure.
- Simple to manage since everything is kept on one server.

Disadvantages:

- Limited by the hardware's maximum capacity.
- Expensive, as high-end hardware can be very costly.
- Single point of failure: if the server fails, the entire database goes offline.

LET'S START WITH DBMS :)

Scaling in Databases

Horizontal Scaling (Scaling Out):

Horizontal scaling involves spreading the database workload across multiple servers. This can be done in different ways:

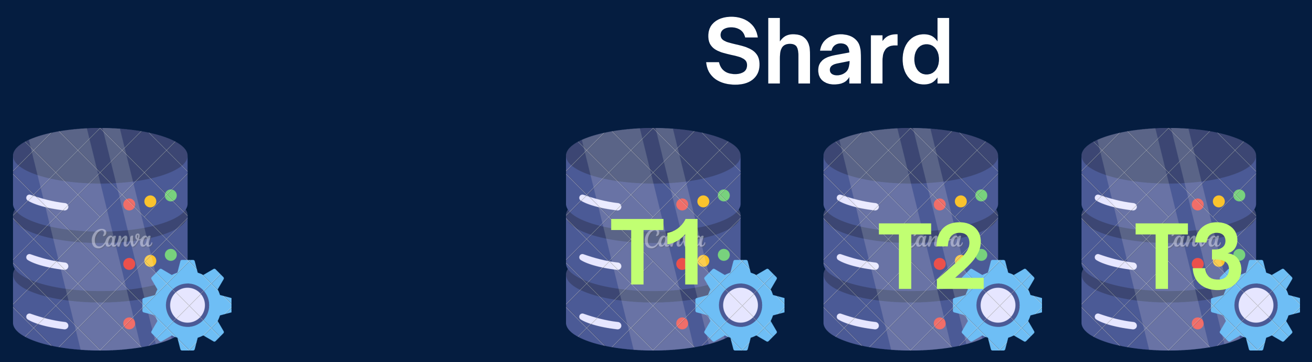
a. Database Replication:
Replication is the process of copying data from one main server (master) to one or more backup servers (slaves).

- **Master-Slave Replication:** The master server manages all write operations, while read operations are distributed among the slave servers. This setup improves read performance and provides redundancy.

- **Multi-Master Replication:** Multiple servers handle both read and write operations, increasing availability and write capacity. However, it also introduces challenges in managing conflicts.

LET'S START WITH DBMS :)

Scaling in Databases



Divide the data between multiple tables in diff db instances

Horizontal Scaling (Scaling Out):

b. **Database Sharding**

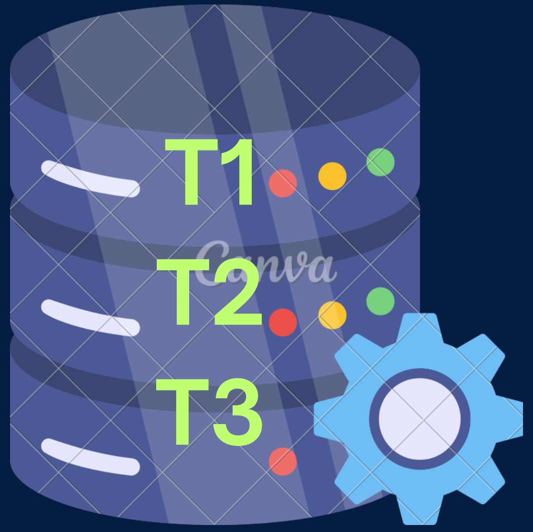
Sharding involves partitioning the database into smaller, more manageable pieces (shards), with each shard stored on a separate server.

- Range-Based Sharding: Data is divided based on ranges of a particular key, such as date or user ID.
- Hash-Based Sharding: A hash function is applied to a key (e.g., user ID) to determine which shard the data should go to.

LET'S START WITH DBMS :)

Scaling in Databases

- **Caching**: Implement caching layers (e.g., Redis, Memcached) to store frequently accessed data in memory, reducing the load on the database.
- **Load Balancing**: Distribute database queries across multiple servers using load balancers to prevent any single server from becoming a bottleneck
- **Partitioning:** Divide large tables into smaller partitions that can be queried independently, improving query performance.



Partitioning : Divide the data into multiple table in same DB instance

LET'S START WITH DBMS :)

RBAC(Role-based access control)

RBAC is a security system that controls who can access a database or system based on their role in the organization.

Instead of giving permissions directly to each user, the system creates roles with specific permissions, and users are assigned to these roles.

For example, a database administrator might create roles like DB_Read for reading data and DB_Write for modifying data. Only people with the right role can read or change the information.

LET'S START WITH DBMS :)

RBAC(Role-based access control)

- **Roles:** A set of permissions associated with a job function (e.g., admin, manager, developer)
- **Permissions:** The rights granted to perform certain actions (e.g., read, write, delete).
- **Users:** Individuals who are assigned one or more roles.

Advantages:

- Makes it easier to manage permissions, especially in big systems.
- Lowers the risk of misuse by making sure users only have access to what they need for their role.
- Improves security and helps keep rules consistent across the system.

LET'S START WITH DBMS :)

<u>Encryption</u>

Encryption in Databases refers to the process of converting data into a secure format that can only be read or accessed by someone with the correct decryption key. This helps protect sensitive information from unauthorized access
For example, an MNC might use encryption to protect records in their database. Even if a hacker gains access to the database, they wouldn't be able to read the data without the decryption key.

Types of Encryption
1. At-Rest Encryption
2. In-Transit Encryption
3. Column-Level Encryption
4. Full Disk Encryption:

LET'S START WITH DBMS :)

Encryption

- **At-Rest Encryption:** At-rest encryption protects data stored in the database by encrypting it when it is saved to disk. This means that if someone gains access to the physical storage or the database files, they won't be able to read the data without the encryption key.

- Example: A company encrypts employee salary records stored in their database. If someone accesses the database files directly, they see encrypted data like xYz1234!@# instead of the actual salary figures.

LET'S START WITH DBMS :)

Encryption

- **In-Transit Encryption:** In-transit encryption secures data as it moves between the database and applications or users. This is typically done using secure communication protocols like SSL/TLS to prevent eavesdropping or tampering during data transmission.

- Example: When you send an email, in-transit encryption ensures that the email content is scrambled while traveling to the recipient's server, so even if intercepted, it appears as random characters like qW9rTy!@ and is unreadable.

LET'S START WITH DBMS :)

Encryption

- **Column-Level Encryption:** Column-level encryption encrypts specific columns within a database. This is used when only certain pieces of data, like credit card numbers or social security numbers, need to be protected, leaving the rest of the data unencrypted for easier access.

- Example: A database holds user information with a column for Social Security Numbers (SSNs). This column is encrypted, so an SSN like 123-45-6789 might be stored as aBcXyZ!@3 in the database.

LET’S START WITH DBMS :)

Encryption

- **Full Disk Encryption**: Full disk encryption encrypts the entire disk where the database is stored. This ensures that all data on the disk is protected, not just the database but also any files, logs, or backups stored on that disk.

- Example: A laptop used by an employee is encrypted. If the laptop is stolen, the entire disk is protected, and the thief would see only encrypted data like r9T!@2d3Xy instead of readable files.

LET’S START WITH DBMS :)

Encryption

Advantages

- It ensures that sensitive information remains secure, even if the database is breached.
- Helps organizations meet regulatory requirements for data security, such as GDPR, HIPAA
- Limits who can view or modify encrypted data to those with the appropriate decryption keys.

Challenges

- Encryption can slow down database operations because of the additional processing required.
- If decryption keys are lost, the data may become permanently inaccessible.

LET’S START WITH DBMS :)

Data masking techniques

Data masking means hiding or changing sensitive information so it’s protected from unauthorized access. This allows the data to be used for things like testing, analytics, or training without exposing the real data. The goal is to create data that looks real but isn’t actually the original information.

For example : A bank needs to train its machine learning models to detect fraudulent transactions. To protect customers' personal information, the bank uses data masking to replace real account numbers and transaction details with fake but realistic-looking data. This way, the models can be trained effectively without risking exposure of actual customer data.

LET'S START WITH DBMS :)

Data masking techniques

Common Data Masking Techniques

Substitution:

- Replacing sensitive data with random but realistic-looking values. For example, swapping real names with random names from a list.

- Example: A company's employee database contains Social Security Numbers (SSNs). To mask these, the company replaces real SSNs with randomly generated fake SSNs. So, 123-45-6789 might be substituted with 987-65-4321.

LET’S START WITH DBMS :)

Data masking techniques

Common Data Masking Techniques

Shuffling:

- Rearranging the values within a column so that the data remains realistic but is not linked to the original records.
- Example: A hospital has a list of patients with their birth dates. To protect privacy, the hospital shuffles the birth dates among the patients. For instance, Patient A's birth date of 01/15/1980 might be swapped with Patient B's birth date of 07/22/1985.

LET’S START WITH DBMS :)

Data masking techniques

Common Data Masking Techniques

Encryption:

- Encrypting data so that it is unreadable without the decryption key. This can be used as a form of masking if the data doesn't need to be human-readable.
- Example: A retail company encrypts credit card numbers in its database. A number like 4111 1111 1111 1111 might be encrypted to something like Ae34Bf98Gh65Jk21, which can’t be read without the decryption key.

LET'S START WITH DBMS :)

Data masking techniques

Common Data Masking Techniques

Nulling Out:

- Replacing sensitive data with null values or placeholders like "XXXX". This removes the original data but may reduce the usefulness of the dataset.
- Example: An HR department wants to share employee records for analysis but needs to hide salary information. They replace the salary field with null values or placeholders like N/A or XXXX.

LET'S START WITH DBMS :)

Data masking techniques

Common Data Masking Techniques

Number Variance:

- Altering numeric data by adding or subtracting a random value within a certain range. For example, changing salary figures slightly to mask the exact amounts

- Example: A finance department needs to protect the exact sales figures but still wants to share data for trend analysis. They slightly alter the figures by adding or subtracting a small random amount. For example, a sales figure of $10,000 might be changed to $10,020 or $9,980.

LET'S START WITH DBMS :)

Data masking techniques

Common Data Masking Techniques

Tokenization:

- Tokenization is a technique used to replace sensitive data with non-sensitive equivalents, called "tokens." These tokens can be stored and processed without exposing the original sensitive data, while maintaining a mapping to the original values when needed.

- Example: Imagine a database that stores customer credit card information. To protect this sensitive data, tokenization can be applied. Each original credit card number is replaced with a unique token (e.g., TKN_XYZ123). These tokens have no meaningful relationship to the actual credit card numbers.