Dive into Indexes

SYSTEM DESIGN

Database Indexes

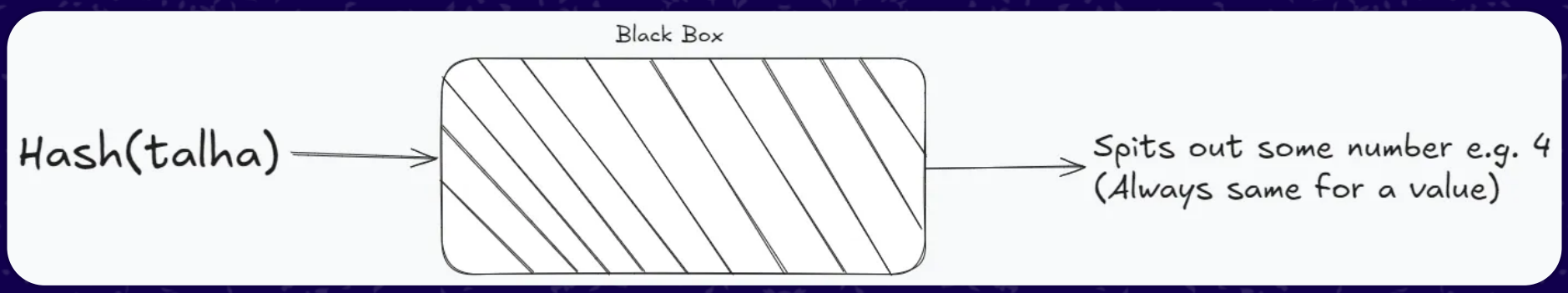
Talha Rizwan
@Toosterr

Indexes in Databases

Indexes are just a better way of reading/finding rows with specific value. They are widely used in large scale applications, but they come with the cost of slower writes as they have to perform multiple operations behind the scenes. Let's dive into some types of indexes that are widely used.

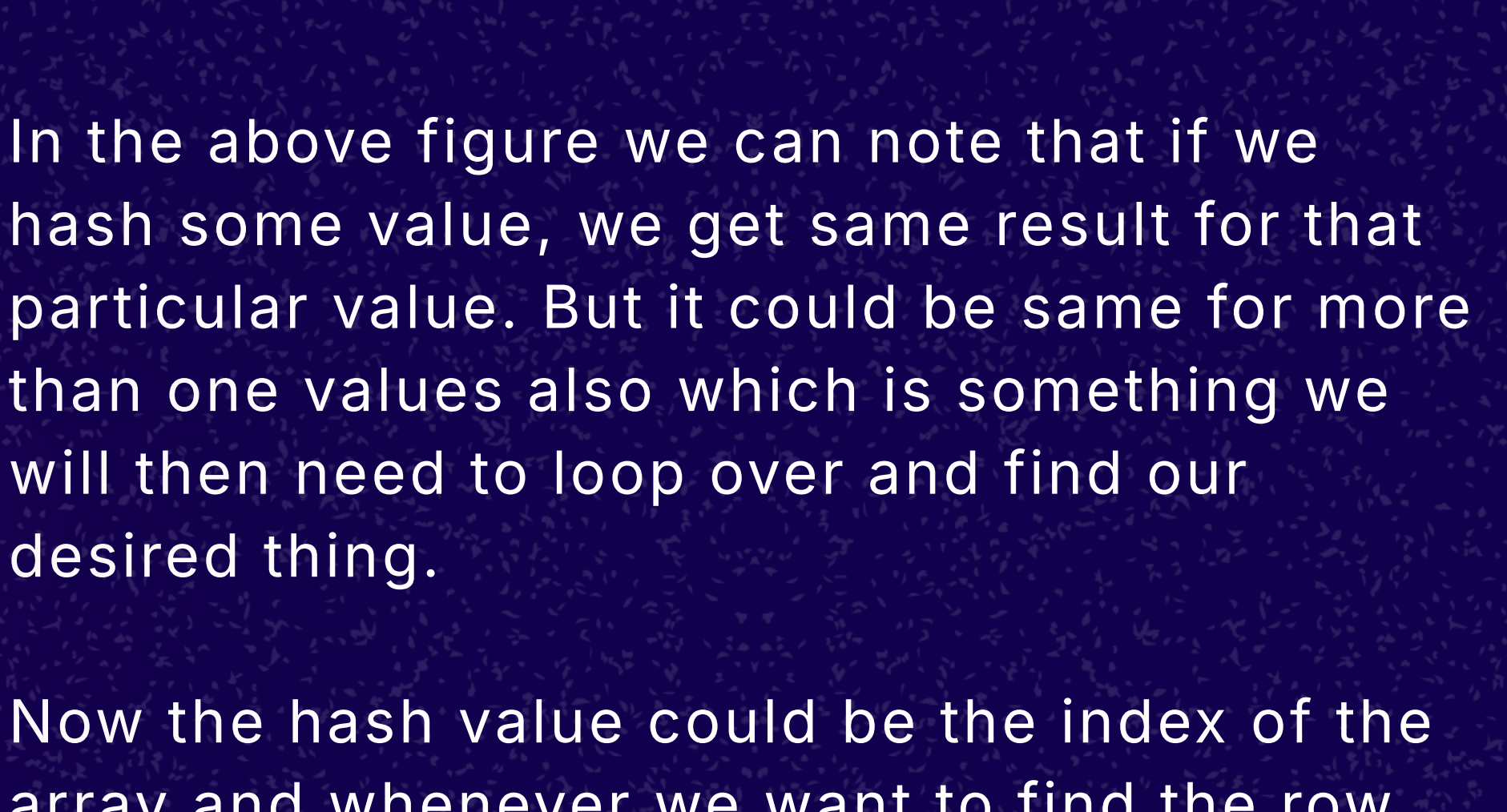
Hash Indexes

Hash index is based on hash map, and we know that in hash maps we have a function which is basically hidden from user but has the following functionality.



In the above figure we can note that if we hash some value, we get same result for that particular value. But it could be same for more than one values also which is something we will then need to loop over and find our desired thing.

Now the hash value could be the index of the array and whenever we want to find the row. We just use the hash function and get to know the location and access it. Making the read complexity equals to O(1)



Now for the issue when 2 or more values may get the same index value for that we can do the following:

- **Chaining:** We can store a linked list at that index and if more values with same hash comes, we can eventually grow our linked list.
- **Probing:** Store the other data in next possible index.

Although we may have linear lockup O(n) at these locations but still it is considered average to constant time if we use "Amortized Time Complexity" or "Load Factor".

Issue with Hash Index

- **Hash maps on hard disks are bad** because we are distributing data to multiple location making hard disk pointer to move a lot making it slower than usual. For this we use RAM for storing hash indexes, making this expensive and not durable because of RAM's volatile nature. and to tackle out the volatile nature we can use **Write Ahead Log (WHL)** which is on disk and sequential. On every DB startup we get all the logs and perform the operations on our hash map. This way we can replay all our operations to the hash map making it durable.
- **No Range Queries:** Hash maps are feasible for concurrent keys, but when we are not defining a concurrent key then we have to loop over the map and find our desired result making it O(n) for

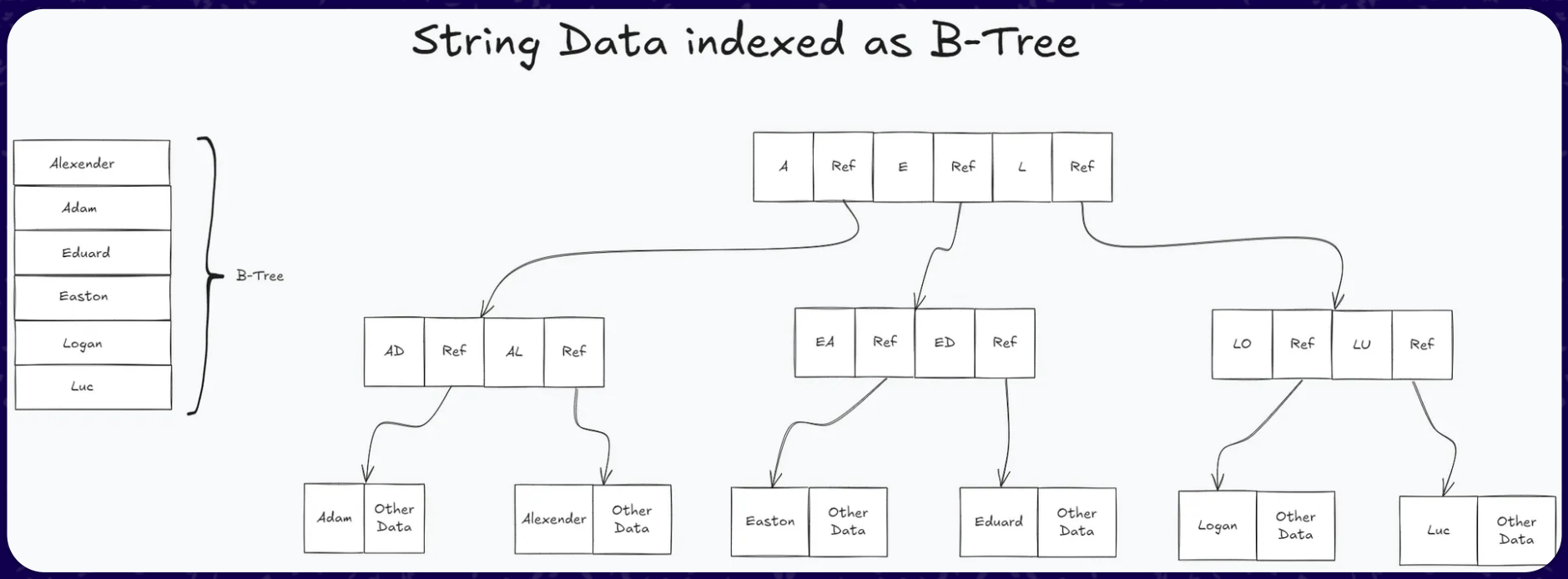
B-Tree Indexes (In Disk):

B-Tree index consist of a Tree structure having range of keys which ref to further keys which may be range of further keys and at root note we have the records the actual data.

Interesting fact about B-Tree is it always remain balanced (All left and right nodes may differ by at most 1 height)

Limitation We have every node a set and it have a fixed size approx. 8kb
Time Complexity: We are traversing tree which is balanced so we know that it has a time complexity of O(log n)

Let's visualize it and have better understanding:



Writes in B-Tree

If we add a new data it will go to its respective node but if the node does not have enough capacity it will delineate and make a new ref node to its parent node and a new leaf node will be added. And incase if the parent node does not have space it will go up and do the exact thing and if the root also don't have space then it will also split and make a new root.

Edge Case: What if during updating process system dies (e.g. someone spills coffee onto the server). We can again use Write ahead log to fix the inconstant state.

Pros/Cons of B-Tree

- It can handle range queries same as simple queries. As the nodes have refs to bunch of data.
- It has self-balancing ability
- No limit of data size.

- It is slower than hash as it is a tree instead of hash map.

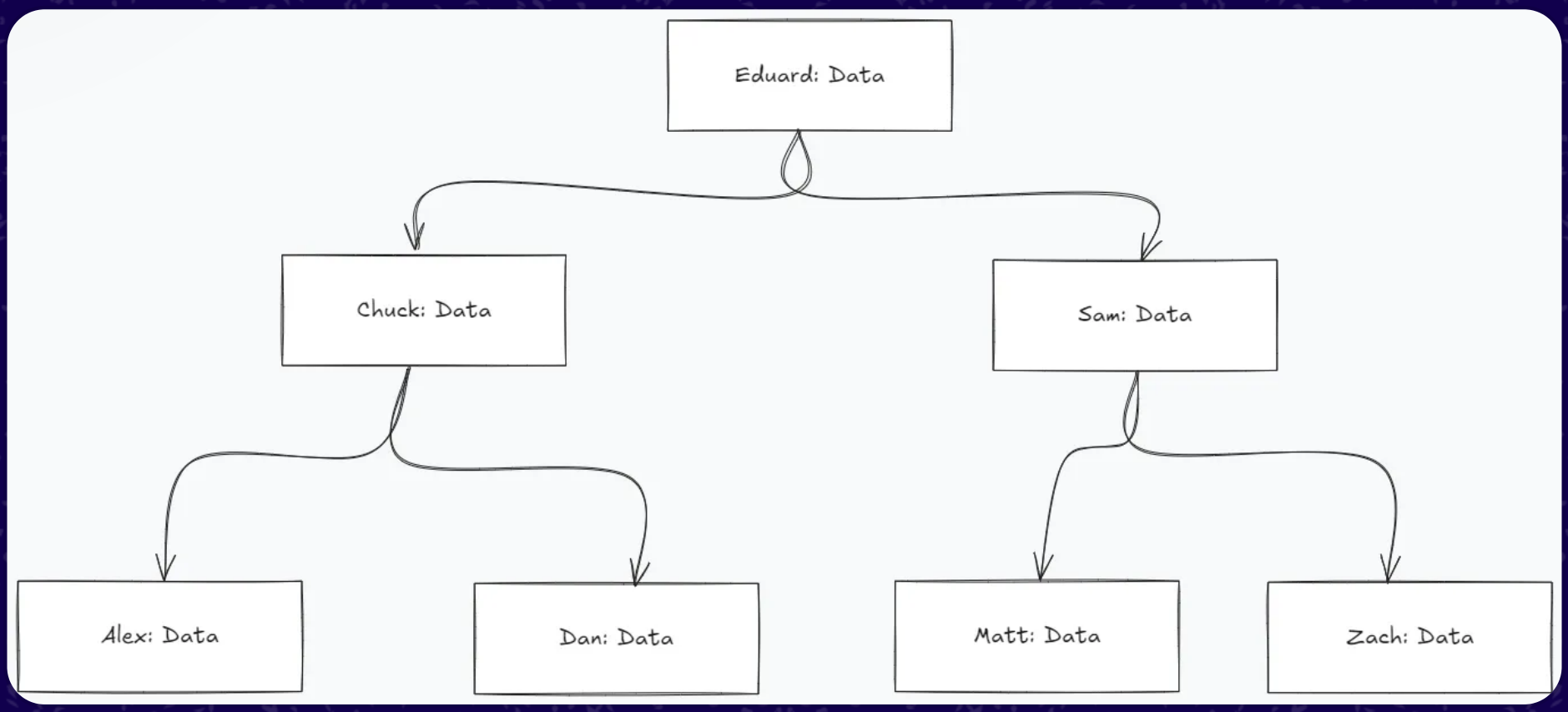
LSM Tree + SS Table Index

LSM tree could be any balanced binary search tree e.g. AVL Tree, Red Black Tree, B-Tree

LSM Tree index is made in memory so we have some speed going on already. All the reads and writes are O(log n) and In memory and fast. All the data stored in it is also sorted so we have

LSM tree it self is not durable as it is in memory. But we have a fix for it that we had for hash index the WAL(write ahead log).

LSM Tree Visualized:



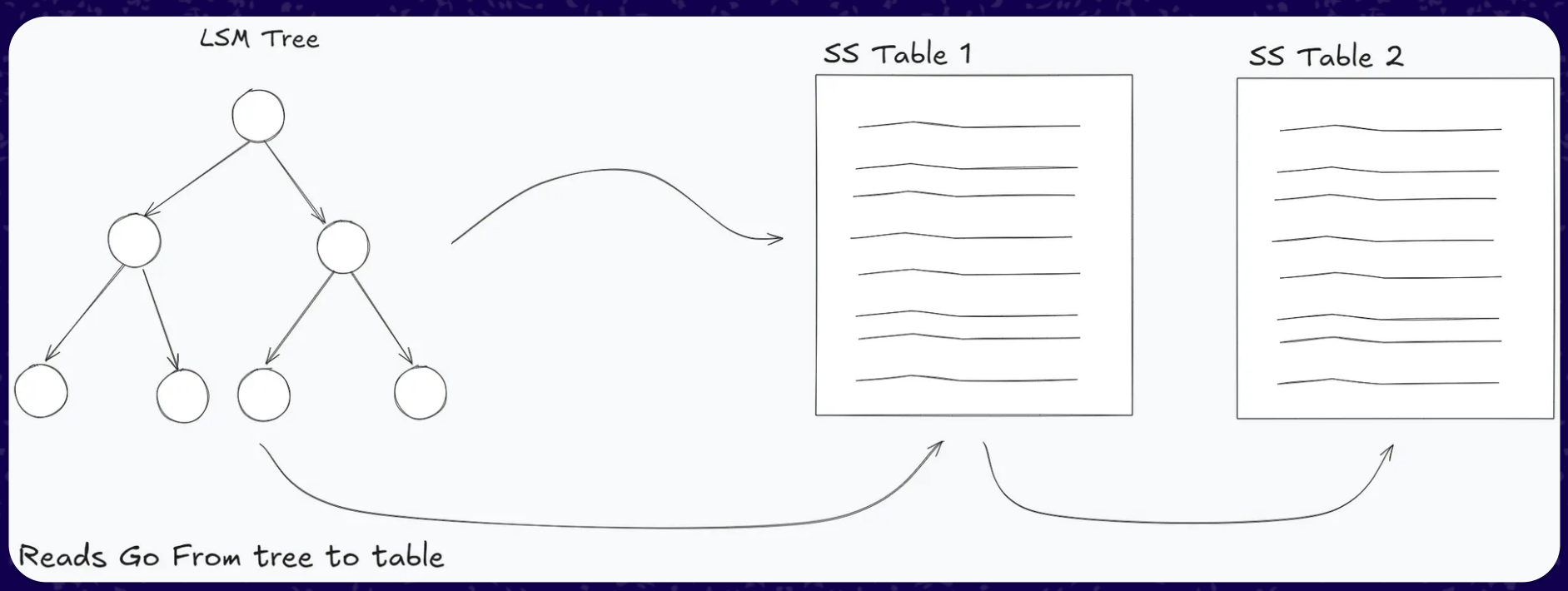
Space Issue In LSM

As we are storing our index in memory so we have a problem of storage how we can fix it? It basically works like the following:

- LSM Tree getting too big?
- Reset the tree purge all the data.
- Convert the purge data into SS Table

SS Table

SS tables are on disk, sorted and are immutable. and converting the LSM tree on to the SS table is not expensive either BST in order traversal can do it in O(n).



Other features

Reads: Reads are like in a order from LSM tree to the Nth SS table if not found in between. But as we said the tables are immutable then they could have duplicate data. So, we read and consider the most recent entry of that data.

Deletion: Deletions are handled using **Tomb Stone.**

Tomb Stone: Any value is considered as tomb stone value if it has a deleted flag set to true. So, on data deletion we mark it as tomb stone value and on read if the value is marked tomb stone, then we consider as deleted.

Time Complexity: Both LSM Tree and SS Tables have separate O(log n) complexity.

LSM Optimization

We can use different techniques to optimize this index such as using:

1. **Sparse Index**: From SS table put some of the most frequently used keys in sparse index with the reference to the address and as they are sorted, we can have binary search on that index.
2. **Bloom Filter**: Bloom filter is a unique type of data structure which stores hash of keys and when a new read comes it will tell if the key exists in it or not. We can use this on SS Tables and ensure if the table have the key or not.

Conclusion

Indexes enhance database read performance by providing quick access paths to data but at the cost of slower writes due to maintenance overhead. Common types include:

1. **Hash Indexes:** Fast (O(1) reads) for exact matches but struggle with range queries and are inefficient on disk due to scattered data placement.
2. **B-Tree Indexes:** Suitable for both exact and range queries with a balanced structure (O(log n) reads). They are durable but slower compared to hash indexes.

3. **LSM Tree + SS Table:** LSM Trees (in-memory) offer quick writes and are periodically merged into immutable SS Tables (on-disk), handling both reads and writes efficiently with O(log n) complexity. Optimizations like sparse indexes and bloom

Conclusion Cont.

Indexes boost read speed at the cost of slower writes. Hash indexes are quick for exact lookups, B-Trees handle range queries well, and LSM Trees balance performance with durability through SS Tables.

Extras

Range Queries:
Range queries are a very popular type of database quires which give a result between some range. Most of the time when we need more than one type of data. Most common use case of range quires are filters e.g. Show rows with names from A - C OR Show all my posts in last hour.

Balanced Binary Search Tree:
Every node in this tree is such that the node on the left has lower value and every node on the right has higher value making it in sorted form. With the node on right and left has the max height difference of 1 node.

Thanks for Reading

Talha Rizwan

@Toosterr